Python Chases Monkey

An Introduction to Python Programming

Jason M. Pittman

Chapter 8

Python Lays Eggs

Introduction

There are a hundred and one reasons why reading and writing files are important in programming. Dealing with files will feel like we have crossed a major milestone. In fact, we have crossed a programming rubicon because either our program is crunching enough logics to warrant reading input from a file or writing output to something other than the screen. Further, reading from and writing to files increases our computational power tenfold.

Consider ther finer points of working with files. We can now save our computation to a persistent format. Moreover, we can read someone else's results. We can format and reformat. We can integrate. Honestly, the possibilites are well over one hundered and one.

Now, I realize that pythons do not lay eggs. However, the resulting visual works better as a metaphor for reading and writing to files compared to *python gives birth.* Let's go with it for the purposes of narrative. So…the entire jungle is at pause; the trees, all the pythons and monkeys and leaves alike. Everything is organized for the great chase…

Learning Objectives

1. Explain fundamental aspects of file structure and file locations.
2. Read data to files
3. Write data to files

Understanding files

Although files are simple to look at, there actually is a lot going on under the hood so to speak. Thus, there are four of things for us to understand up front so that we work with files properly. As easy as file input and output can be, so too is wrecking filesystem objects when we're not careful.

File structure

Put simply, there are three sections to a file. The one section we see is **data**. It would be fair to think of the *data* section as the content. It would also be fine to think of the data or content as the filling in the sandwich. Thus, the other two sections are relatively above and below the content.

Above the data is the **header** section. Header structure is something best discussed elsewhere but we should know that file type, content type, and other metadata exist in the header. Because of the sensitive nature of these metadata, let's just agree to not monkey with file header contents. Fair.

The last section comes after the data although in some sense it is part of the data. The **EOF** section is an indicator that we have programmatically gotten to... yes, you guessed it, the end of a file. As luck (or computer science) would have it, the EOF is not something we can mistakenly wreck. EOF is implicitly, invisibly always present whether we have zero data or infinite data. We'll come back to this section later.

File location

Naturally, knowing what a file is and what the internal structure of a file looks like can help us understand better how to work with them. However, one thing we need to be very careful about is *where* files reside. This point is more complex than we might think because of two factors.

The first factor is *full* versus *relative* pathing.

Stemming from the first factor, we have a second consideration when it comes to file location. More than anything, how we handle file location dictates the portability of our source code. Portability in this context means our code functions (as expected) across various operating systems. Thankfully, there are just two types of operating systems to deal with: Windows and Unix or its many, many variants. Further, the difference between the two is subtle, just `/` versus `\`.

We may be tempted to construct a file path as follows:

Example 1

```
(1) folder = "dir/subdir/"
(2) file = folder + "some_file.txt"
```

This is wrong and for the love of every monkey and python don't do it. First, line one (a) assumes the `dir` is in the same location as the program and (b) assumes we are running the program in Unix. Instead, we should do the following:

Example 2

```
from pathlib import Path

folder = Path("dir/subdir/")
file = folder / "some_file.txt"
```

The beauty of using the `pathlib` module is that slashes are dynamically converted to the correct operating system type. Indeed, `pathlib` is powerful and provides a host of useful methods. Exploring these is beyond the scope of this book but I strongly recommend reading the official documentation along with some healthy trial-and-error on your own!

File line ending

Another thing based on the type of operating system is file line endings. While `EOF` is a universal file structure concept that is always present, the demarcation of the end of a line within a file is conceptually universal but practically different. Moreover, the line ending is only present in conjunction with a present line of data.

Again, we have a subtle but important difference based on the underlying operating system. Whereas Windows uses `\r\n`, Unix and its offshoots uses `\r` at the end of a line followed by `\n` on a new line. Get used to that last term by the way, **newline**.

File encoding

Back to our **data** section then where we can now talk about the real nature of a file. It is still true that a file is a persisted data structure yes. However, we can more deeply think of a file as a container for *bytes*. Deeper still, we can define the format of content by declaring an encoding scheme for our data-as-bytes. Chiefly, we use `UTF-8` encoding which gives us *unicode* and *ascii* formats. For now, if we need to specifyg we should go with `UTF-8`.

Reading from Files

The next phase in our Python file handling kung-fu learning is right in front of us now that we have a working knowledge of files. That's right, it is time to open and *read*. Call ahead - the `open()` method is the same we'll use when we start *writing* to files as well.

The first steps in reading data from a file is identical to handling the file location. In **Example 3**, we employ the `pathlib` technique again and add on a simple call to the built-in `open()` method as shown on line six. Technically, we've not so much opened the file as we have opened a file handle attached to the indicated file.

Example 3

```
(1) from pathlib import Path
(2)
(3) folder = Path("dir/subdir/")
(4) file = folder / "some_file.txt"
(5)
(6) f = open(file)
```

An important lesson to pick up now is that anytime we *open* something, the last step after we are done is the *closing* that something. There are two ways to close the file handle we just opened.

Example 4

```
(1) from pathlib import Path
(2)
(3) folder = Path("dir/subdir/")
(4) file = folder / "some_file.txt"
(5)
(6) f = open(file)
(7) try:
(8)     # read here
(9) finally:
(10) f.close()
```

There is a much more compact, more *Pythonic* method at our disposal. To wit:

Example 5

```
(1) with open(file, 'r') as f:
(2)     # read here
```

The `with` keyword is convenient because it combines the `try-finally` logic, including **exception** handling, along with the calling of the `close()` method.

The other small detail we added is the `'r'`. This is the signal to open the file handle in read-only mode. If we are interested in only reading data, this is the safe option. Later, when writing we will experience a few other common access mode options.

Once the file handle `f` is established as in **Example 5**, we can read data. Here we commonly call one of two methods: `read()` or `readlines()`. The difference is `read()`

can take a size in bytes and return that much data (e.g., `read(size=5)`). By the way, calling `read()` without a parameter will consume the whole data section.

Example 6

```
with open(file, 'r') as f:
    f.read()
```

Alternatively, we can call `readlines()` without a parameter to read each line of data (hence the earlier discussion about line endings!) into a **List**. Passing a parameter will cause `readlines()` to read that many characters.

Example 7

```
with open(file, 'r') as f:
    f.readlines()
```

Notice how we don't have to look for the `EOF`? Technically, Python will return an empty `str` when the EOF is reached. Implicitly, this will cause a **definite iteration** to break. That's pretty cool as we don't have to explicitly handle the condition.

Writing to Files

Here's the payoff- writing data to a file. Again, we open the file handle identically to how we did with reading. Well, except for a couple of details.

First, we have to use a different file access mode in the `open()` call. There are a handful of options here but essentially we have to decide whether we want to overwrite an existing file of the same name or append to it if it exists. The former is achieved with `w` or `w+`. The latter- append mode- is achieved with `a` or `a+`. In both situations, the `+` gives us read-write capability.

Second, we call the `write` method instead of `read`. That should be expected I suppose. Check this out:

Example 7

```
with open(file, 'a+') as f:
    f.write("Append this line")
```

Further, as with reading, we also have a `writelines()` method option. Instead of passing a string, with `writelines()` we pass a *sequence*. The difference between sequence and string being that the sequence doesn't implicitly include a line ending. Believe it or not, there are proper uses cases. However, I suspect we'll often just use `write()`.

Excercises

1. Create an example program that implements the logic in **Example 2** but replace the `/` on line three with `\`. What happens when running the program?
2. Implement a plaintext file `pythons.txt` with `Peetie`, `Piotr`, and `Paulina` as data on three lines. Write a simple program to read the contents using `read(size=5)`. Do the results match your expected outcome? If no, explain what happens with the program.
3. Develop a program to write the string `Prez` in append access mode (not append plus read!) to the `pythons.txt` file.

Questions

1. How would you programmatically read `EOF`?
2. What is the **efficiency** difference between **Example 4** and **Example 5**?
3. How does calling the `.readlines()` method work if we open the file handle in `w` access mode?
4. What line ending would you include in a `writelines(seq)` call?