Jul. 08, 2013

Files (/files/) IO (/io/)

System & OS (/systems-programming/)

Reading and Writing Files in Python

Overview

When you're working with Python, you don't need to import a library in order to read and write files. It's handled natively in the language, albeit in a unique manner.

The first thing you'll need to do is use Python's built-in ***open*** function to get a ***file object***.

The ***open*** function opens a file. It's simple.

When you use the ***open*** function, it returns something called a ***file object***. ***File objects*** contain methods and attributes that can be used to collect information about the file you opened. They can also be used to manipulate said file.

For example, the ***mode*** attribute of a ***file object*** tells you which mode a file was opened in. And the ***name*** attribute tells you the name of the file that the ***file object*** has opened.

You must understand that a ***file*** and ***file object*** are two wholly separate – yet related – things.

File Types

What you may know as a file is slightly different in Python.

In Windows, for example, a file can be any item manipulated, edited or created by the user/OS. That means files can be images, text documents, executables, and much more. Most files are organized by keeping them in individual folders.

In Python, a file is categorized as either text or binary, and the difference between the two file types is important.

Text files are structured as a sequence of lines, where each line includes a sequence of characters. This is what you know as code or syntax.

Each line is terminated with a special character, called the EOL or **End of Line** character. There are several types, but the most common is the comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun.

A backslash character can also be used, and it tells the interpreter that the next character – following the slash – should be treated as a new line. This character is useful when you don't want to start a new line in the text itself but in the code.

A binary file is any type of file that is not a text file. Because of their nature, binary files can only be processed by an application that know or understand the file's structure. In other words, they must be applications that can read and interpret binary.

Open ( ) Function

In order to open a file for writing or use in Python, you must rely on the built-in ***open ()*** function.

As explained above, ***open ( )*** will return a file object, so it is most commonly used with two arguments.

An argument is nothing more than a value that has been provided to a function, which is relayed when you call it. So, for instance, if we declare the name of a file as "Test File," that name would be considered an argument.

The syntax to open a file object in Python is:

```
file_object  = open("filename", "mode") where file_object is the variable to a
dd the file object.
```

The second argument you see – ***mode*** – tells the interpreter and developer which way the file will be used.

Mode

Including a mode argument is optional because a default value of '***r***' will be assumed if it is omitted. The '***r***' value stands for read mode, which is just one of many.

The modes are:

- '***r***' – Read mode which is used when the file is only being read
- '***w***' – Write mode which is used to edit and write new information to the file (any existing files with the same name will be erased when this mode is activated)
- '***a***' – Appending mode, which is used to add new data to the end of the file; that is new information is automatically amended to the end
- '***r+***' – Special read and write mode, which is used to handle both actions when working with a file

So, let's take a look at a quick example.

```
F = open(“workfile”,”w”)
Print f
```

This snippet opens the file named “workfile” in writing mode so that we can make changes to it. The current information stored within the file is also displayed – or printed – for us to view.

Once this has been done, you can move on to call the objects functions. The two most common functions are read and write.

Create a text file

To get more familiar with text files in Python, let’s create our own and do some additional exercises.

Using a simple text editor, let’s create a file. You can name it anything you like, and it’s better to use something you’ll identify with.

For the purpose of this tutorial, however, we are going to call it “testfile.txt”.

Just create the file and leave it blank.

To manipulate the file, write the following in your Python environment (you can copy and paste if you’d like):

```
file = open(“testfile.txt”,”w”)

file.write(“Hello World”)
file.write(“This is our new text file”)
file.write(“and this is another line.”)
file.write(“Why? Because we can.”)

file.close()
```

Naturally, if you open the text file – or look at it – using Python you will see only the text we told the interpreter to add.

```
$ cat testfile.txt
Hello World
This is our new text file
and this is another line.
Why? Because we can.
```

Reading a Text File in Python

There are actually a number of ways to read a text file in Python, not just one.

If you need to extract a string that contains all characters in the file, you can use the following method:

```
file.read()
```

The full code to work with this method will look something like this:

```
file = open("testfile.text", "r")
print file.read()
```

The output of that command will display all the text inside the file, the same text we told the interpreter to add earlier. There's no need to write it all out again, but if you must know, everything will be shown except for the "$ cat testfile.txt" line.

Another way to read a file is to call a certain number of characters.

For example, with the following code the interpreter will read the first five characters of stored data and return it as a string:

```
file = open("testfile.txt", "r")

print file.read(5)
```

Notice how we're using the same *file.read()* method, only this time we specify the number of characters to process?

The output for this will look like:

```
Hello
```

If you want to read a file line by line – as opposed to pulling the content of the entire file at once – then you use the *readline()* function.

Why would you use something like this?

Let's say you only want to see the first line of the file – or the third. You would execute the *readline()* function as many times as possible to get the data you were looking for.

Each time you run the method, it will return a string of characters that contains a single line of information from the file.

```
file = open("testfile.txt", "r")
print file.readline():
```

This would return the first line of the file, like so:

```
Hello World
```

If we wanted to return only the third line in the file, we would use this:

```
file = open("testfile.txt", "r")
print file.readline(3):
```

But what if we wanted to return every line in the file, properly separated? You would use the same function, only in a new form. This is called the *file.readlines()* function.

```
file = open("testfile.txt", "r")
print file.readlines()
```

The output you would get from this is:

```
['Hello World', 'This is our new text file', 'and this is another line.', 'Wh
y? Because we can.']
```

Notice how each line is separated accordingly? Note that this is not the ideal way to show users the content in a file. But it's great when you want to collect information quickly for personal use during development or recall.

Looping over a file object

When you want to read – or return – all the lines from a file in a more memory efficient, and fast manner, you can use the loop over method. The advantage to using this method is that the related code is both simple and easy to read.

```
file = open("testfile.txt", "r")
for line in file:
print line,
```

This will return:

```
Hello World
This is our new text file
and this is another line.
Why? Because we can.
```

See how much simpler that is than the previous methods?

Using the File Write Method

One thing you'll notice about the file write method is that it only requires a single parameter, which is the string you want to be written.

This method is used to add information or content to an existing file. To start a new line after you write data to the file, you can add an EOL character.

```
file = open("testfile.txt", "w")

file.write("This is a test")
file.write("To add more lines.")

file.close()
```

Obviously, this will amend our current file to include the two new lines of text. There's no need to show output.

Closing a File

When you're done working, you can use the ***fh.close()*** command to end things. What this does is close the file completely, terminating resources in use, in turn freeing them up for the system to deploy elsewhere.

It's important to understand that when you use the ***fh.close()*** method, any further attempts to use the file object will fail.

Notice how we have used this in several of our examples to end interaction with a file? This is good practice.

File Handling in the Real World

To help you better understand some of the methods discussed here, we're going to offer a few examples of them being used in the real world. Feel free to copy the code and try it out for yourself in a Python interpreter (make sure you have any named files created and accessible first).

Opening a text file:

```
fh = open("hello.txt", "r")
```

Reading a text file:

```
Fh = open("hello.txt", "r")
print fh.read()
```

To read a text file one line at a time:

```
fh = open("hello.text", "r")
print fh.readline()
```

To read a list of lines in a text file:

```
fh = open("hello.txt", "r")
print fh.readlines()
```

To write new content or text to a file:

```
fh = open("hello.txt", "w")

fh.write("Put the text you want to add here")
fh.write("and more lines if need be.")

fh.close()
```

You can also use this to write multiple lines to a file at once:

```
fh = open("hello.txt","w")
lines_of_text = ["One line of text here", "and another line here", "and yet an
other here", "and so on and so forth"]
fh.writelines(lines_of_text)
fh.close()
```

To append a file:

```
fh = open("hello.txt", "a")
fh.write("We Meet Again World")
fh.close
```

To close a file completely when you are done:

```
fh = open("hello.txt", "r")
print fh.read()
fh.close()
```

With Statement

You can also work with file objects using the with statement. It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it's good practice to use the with statement where applicable.

One bonus of using this method is that any files opened will be closed automatically after you are done. This leaves less to worry about during cleanup.

To use the with statement to open a file:

```
with open("filename") as file:
```

Now that you understand how to call this statement, let's take a look at a few examples.

```
with open("testfile.txt") as file:
data = file.read()
do something with data
```

You can also call upon other methods while using this statement. For instance, you can do something like loop over a file object:

```
with open("testfile.txt") as f:
for line in f:
print line,
```

You'll also notice that in the above example we didn't use the "***file.close()***" method because the with statement will automatically call that for us upon execution. It really makes things a lot easier, doesn't it?

Using the With Statement in the Real World

To better understand the with statement, let's take a look at some real world examples just like we did with the file handling functions.

To write to a file using the with statement:

```
with open("hello.txt", "w") as f:
f.write("Hello World")
```

To read a file line by line, output into a list:

```
with open("hello.txt") as f:
data = f.readlines()
```

This will take all of the text or content from the "hello.txt" file and store it into a string called "data".

Splitting Lines in a Text File

As a final example, let's explore a unique function that allows you to split the lines taken from a text file. What this is designed to do, is split the string contained in variable data whenever the interpreter encounters a space character.

But just because we are going to use it to split lines after a space character, doesn't mean that's the only way. You can actually split your text using any character you wish - such as a colon, for instance.

The code to do this (also using a with statement) is:

```
with open("hello.text", "r") as f:
data = f.readlines()

for line in data:
words = line.split()
print words
```

If you wanted to use a colon instead of a space to split your text, you would simply change line.split() to line.split(":").

The output for this will be:

```
[“hello”, “world”, “how”, “are”, “you”, “today?”]
[“today”, “is”, “Saturday”]
```

The reason the words are presented in this manner is because they are stored – and returned – as an array. Be sure to remember this when working with the split function.

More Reading

```
Official Python Documentation - Reading and Writing Files (http://docs.python.
org/2/tutorial/inputoutput.html#reading-and-writing-files)
Python File Handling Cheat Sheet (../../../../cheatsheet/python-file-handlin
g/)
Non-Programmer's Tutorial for Python 3 (http://en.wikibooks.org/wiki/Non-Progr
ammer's_Tutorial_for_Python_3/)
Beginning Python
 (http://chryswoods.com/beginning_python/)
```

Recommended Python Training – DataCamp (https://www.datacamp.com/courses/tech:python?tap_a=5644-dce66f&tap_s=75426-9cf8ad)

For Python training (https://www.datacamp.com/courses/tech:python?tap_a=5644-dce66f&tap_s=75426-9cf8ad), our top recommendation is DataCamp.

Datacamp (https://www.datacamp.com/courses/tech:python?tap_a=5644-dce66f&tap_s=75426-9cf8ad) provides online interactive courses that combine interactive coding challenges with videos from top instructors in the field.

Datacamp has beginner to advanced Python training that programmers of all levels benefit from.