Interacting with the File System

John M. Morrison

May 3, 2016

Contents

1 Introduction

In this chapter we shall introduce some new modules and features that will make Python a far more useful applications programming tool. You will gain command of your file system and be able to manipulate files and obtain file metadata without opening files.

We begin by introducing raw strings, which are a real convenience when working with fileIO, especially on the Windoze side. We then show a couple of useful features in the `sys` module.

We then will study the `os` and `os.path` modules, and make ample use of the `sys` module.

1.1 A Helpful Tool: Raw Strings

Python supports a version of strings called *raw strings.* To make a raw string literal, just prepend with an `r`. When Python encounters a raw string, all backslashes are read literally. No special meaning is given them by the language. This interactive session shows how it works.

```
>>> path = 'C:\nasty\mean\ugly'
>>> print (path)
C:
asty\mean\ugly
>>> path = r'C:\nasty\mean\ugly'
>>> print (path)
C:\nasty\mean\ugly
>>>
```

Notice that in the raw string, the `\n` did not expand to a newline; it was a literal backslash-n. This is a great convenience when dealing with file paths in Windoze.

Warning! You may not end a raw string with a `\`. This causes the close-quote to be escaped to a literal character and causes a string-delimiter leak. Think for a moment: there is an easy work-around for this!

2 Using `os` for basic File System Functions

This library gives you a means for seeing into your file system. We will dicuss some of its methods. You can also use it to modify your file system and to change the `cwd` of your Python process.

Let us begin by seeing our `cwd` and the files in it. We will also list the contents of the root directory. Note the presence of dotfiles. A list comprehension gets rid of the dotfiles easily if you don't want to see them.

```
>>> os.getcwd()
'/Users/morrison/newEdition'
>>> os.listdir()
['c3.aux', 'c3.log', 'c3.out', 'c3.pdf', 'c3.tex', 'c3.toc',
'c3core.tex', 'files.tex', 'filesCore.tex']
>>> q = os.listdir("/")
>>> q
['.DocumentRevisions-V100', '.DS_Store', '.file', '.fseventsd',
'.hotfiles.btree', '.OSInstallerMessages', '.Spotlight-V100',
'.Trashes', '.vol', 'Android', 'Applications', 'bin', 'cores',
'dev', 'Developer', 'etc', 'home', 'Incompatible Software',
'installer.failurerequests', 'libpeerconnection.log',
'Library', 'net', 'Network', 'opt', 'private', 'sbin',
'System', 'tmp', 'User Guides And Information', 'Users', 'usr',
'var', 'Volumes']
>>> [k for k in q if (not k.startswith("."))]
['Android', 'Applications', 'bin', 'cores', 'dev', 'Developer',
'etc', 'home', 'Incompatible Software',
'installer.failurerequests', 'libpeerconnection.log',
'Library', 'net', 'Network', 'opt', 'private', 'sbin',
'System', 'tmp', 'User Guides And Information', 'Users', 'usr',
'var', 'Volumes']
>>>
```

The command `os.listdir()` can be given any valid path in your system; you see here that it listed the contents of the root directory when asked. It, by default, uses your `cwd`. The list you see is a list of strings containing the names of files present in the dirctory.

You can change the directory of your Python process by using `os.chdir()`; you must specify a valid absolute or relative path for it. Here we show it at work, going to the parent directory, then going to an absolute path.

```
>>> os.getcwd()
'/Users/morrison/newEdition'
>>> os.chdir("..")
>>> os.getcwd()
'/Users/morrison'
>>> os.chdir("/Users/morrison/book")
>>> os.getcwd()
'/Users/morrison/book'
>>>
```

You can also change permissions using Python. When doing so, make sure to use the `0o` prefix so Python will see your code as an octal permission string.

```
>>> import os
>>> os.chmod("fooment", 0o777)
>>>
```

Python allows you to get rid of stuff, too.

```
>>> os.listdir()
['.filesCore.tex.swp', 'c3.aux', 'c3.log', 'c3.out', 'c3.pdf',
'c3.tex', 'c3.toc', 'c3core.tex', 'files.aux', 'files.log',
'files.out', 'files.pdf', 'files.tex', 'files.toc',
'filesCore.tex', 'fooment']
>>> os.remove("fooment")
>>> os.mkdir("garbage")
>>> os.listdir()
['.filesCore.tex.swp', 'c3.aux', 'c3.log', 'c3.out', 'c3.pdf',
'c3.tex', 'c3.toc', 'c3core.tex', 'files.aux', 'files.log',
'files.out', 'files.pdf', 'files.tex', 'files.toc',
'filesCore.tex', 'garbage']
>>> os.remove("garbage")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
PermissionError: [Errno 1] Operation not permitted: 'garbage'
>>> os.rmdir("garbage")
>>> os.listdir()
['.filesCore.tex.swp', 'c3.aux', 'c3.log', 'c3.out', 'c3.pdf',
'c3.tex', 'c3.toc', 'c3core.tex', 'files.aux', 'files.log',
'files.out', 'files.pdf', 'files.tex', 'files.toc',
'filesCore.tex']
>>>
```

3 Using `os` to Get File Metadata

4 Python File IO

File operations in Python are handled by the built-in function `open`. This function opens a pipe between your program and a file. You can specify the file using a relative or absolute path. It is an error to attempt to do operations on a file or into a directory for which you do not have permission. For example, you can open a file for reading if you have read permissions. You cannot open a file for writing if you lack have write permission.

4.1 File Opening Modes

The open function has two arguments: a filename (a string) and a mode (string). We will concentrate on three modes for opening files. We show them in the table here.

Python File Open Modes	
Mode	Explanation
a	This mode will open a file for writing and append that which you write to the end of the file. If the file does not exist, it is created for you. It is an error for you to attempt to write in a directory in which you do not have write permissions. It is an error to attempt to append to a file for which you lack write permissions.
w	This mode will open a file for writing and overwrite the file you specify if it exists and it will create it otherwise. Be warned: *opening an existing file for writing means it will be clobbered.* It is an error for you to attempt to write to a file in a directory in which you do not have write permissions.
r	This mode will open a file for reading; it is an error to attempt to read a nonexistent file or to read from a file for which you do not possess read permissions.

Usage In what follows the file name and mode are both strings. The syntax for opening a file is as follows.

```
filePipe = open(filename, mode)
```

4.2 Writing to a File

You will often see the name `outFilePipe` for a file to write or append to and `inFilePipe` for a file we are reading into our programs. Think of the pipe as a one–way connection between your program and the file. The `write` method sends the characters you write down the pipe and into the file. The `read` method will hoover up the characters from a file into your program. We shall begin by working with the write mode.

```
outFilePipe = open("out.txt", "w")
outFilePipe.write("Hello")
outFilePipe.write("World")
outFilePipe.close()
```

When you open the resulting file `out.txt`, you will see that it contains this text.

```
HelloWorld
```

Notice that there is no newline placed by the write method. If you want newlines, you must put them yourself! The `write` method just pushes the new bytes into the file. We now revise our program as follows

```
outFilePipe = open("out.txt", "w")
outFilePipe.write("Hello\n")
outFilePipe.write("World\n")
outFilePipe.close()
```

and you will now have the newlines you wanted. Do not fail to close the file or it may never write and your work will be lost!

Danger! If you open a file for writing which already exists, it is clobbered. Later, when we learn about the `os` module, we will see how to prevent this. Recall that such clobbering can occur in recipient files when invoking the `mv` and `cp` commands. This feature is present because the destructive overwrite is often a desired side effect of the command.

4.3 Reading from a File

Suppose we have a file named `in.txt` with this Haiku stored in it.

```
I know am an innie
I therefore collect lint daily
It is my lot in life.
```

Now we show how to open it for reading using the `"r"` mode.

```
inFilePipe = open("in.txt", "r")
stuff = inFilePipe.read();
inFilePipe.close()
print (stuff)
print ("len(stuff) = " , len(stuff))
```

The `read()` method reads the file in as one giant string. This string will contain the newlines necessary to reconstruct the original structure of the file.

```
$ python read.py
I know am an innie
I therefore collect lint daily
It is my lot in life.

len(stuff) =  72
```

If you count the characters with spaces, you will come up short characters. Do not forget that there are invisible `'\n'`s that act as end-of-line characters.

The object returned by open is a file object; you can check in an interactive session. Calling the `type()` function on a file pipe results in the reply

```
type<'file'>
```

It is best to think of a file object as a pipe to a file; this pipe is outgoing if we are writing or appending and incoming if we are reading. You can have as many file objects present in your program as you wish, each slurping from and spewing data into different places. These file objects can open and close as needed throughout the lifetime of your program.

4.4 A Bigger Example: `copy.py`

We shall now produce an example of a program that emulates the action of the UNIX command `cp` for files. Recall that the `cp` command needs a *donor file*, which is the file being copied, and a *recipient file*, the destination of the data being copied from the donor. We begin by specifying the *behavior* of our program. We want something like this

```
$ ./copy.py source sink
```

The file `source` should exist. We do not yet have the ability to check this, but that will change before the end of the chapter. The file *sink* will be overwritten by the contents of `source` if it exists. Otherwise it will be created and the contents of `source` will be placed in it.

Create a file named `copy.py` and put this outline of comments in it. In this file, we see the broad details of what we need to do to accomplish our stated task. If you run this program (clearly) it will do nothing.

```
#get the name of the donor file
#get the name of the recipient file
#read in the donor file
#print its contents to the recipient
#make sure all is closed when we are done or else.
```

Let's start at the beginning. We shall begin by just getting the file names from `stdin`. Let us add the shebang line and make the file executable too.

```
#!/usr/bin/python
#get the name of the donor file
#get the name of the recipient file
donor = raw_input("Enter a donor file: " )
recipient = raw_input("Enter a destination file:  ")
#read in the donor file
#print its contents to the recipient
#make sure all is closed when we are done or else.
```

Run the program we have so far and see the donor and recipient file getting requested. So far, nothing has really happened, other than the garnering of the file names.

```
$ chmod u+x copy.py
$ ./copy.py
Enter a donor file: foo.txt
Enter a destination file:  goo.txt
$
```

Next, we open the file for reading, get everything and get out.

```
#!/usr/bin/python
#get the name of the donor file
#get the name of the recipient file
donor = raw_input("Enter a donor file: " )
recipient = raw_input("Enter a destination file:  ")
#read in the donor file
inFilePipe = open(donor, "r")
buf = inFilePipe.read()
inFilePipe.close()
#print its contents to the recipient
#make sure all is closed when we are done or else.
```

Run the program again to check our progress.

```
$ python copy.py
Enter a donor file: in.txt
Enter a destination file:  noplace.txt
```

Since we are doubting Thomases here, we will put in some temporary code to print `buf` to `stdout` and to verify that all is in good order. Now let's write it all into the recipient file.

```
#!/usr/bin/python
#get the name of the donor file
#get the name of the recipient file
donor = raw_input("Enter a donor file: " )
recipient = raw_input("Enter a destination file:  ")
#read in the donor file
inFilePipe = open(donor, "r")
buf = inFilePipe.read()
inFilePipe.close()
#print its contents to the recipient
print (buf)     ##temporary code to test file open.
#make sure all is closed when we are done or else.
```

All is working so far.

```
$ python copy.py
Enter a donor file: in.txt
Enter a destination file:  nowhere.txt
I know am an innie
I therefore collect lint daily
It is my lot in life.

$
```

Notice the extra line at the end; it is the `\n` at the end of the last line of the file that is doing this. Now we will get rid of the temporary code and go for the real thing. Notice how we remembered to open the recipient file for writing.

```
#!/usr/bin/python
#get the name of the donor file
donor = raw_input("Enter a donor file: " )
#get the name of the recipient file
recipient = raw_input("Enter a destination file:  ")
#read in the donor file
inFilePipe = open(donor, "r")
buf = inFilePipe.read()
#open to write to the recipient file  (can't forget this)
outFilePipe = open(recipient, "w")
#print its contents to the recipient
outFilePipe.write(buf)
#make sure all is closed when we are done or else.
inFilePipe.close()
outFilePipe.close()
```

Here is a shell session that shows all.

```
$ ./copy.py
Enter a donor file: in.txt
Enter a destination file:  nowhere.txt
$ more in.txt
I know am an innie
I therefore collect lint daily
It is my lot in life.
$ more nowhere.txt
I know am an innie
I therefore collect lint daily
It is my lot in life.
$
```

We will do one more thing to add polish: we will use the command–line argument feature to give this program a professional look.

```
#!/usr/bin/python
import sys
#get the name of the donor file
donor = sys.argv[1]
#get the name of the recipient file
recipient = sys.argv[2]
#read in the donor file
inFilePipe = open(donor, "r")
buf = inFilePipe.read()
#open to write to the recipient file  (can't forget this)
outFilePipe = open(recipient, "w")
#print its contents to the recipient
outFilePipe.write(buf)
```

```
#make sure all is closed when we are done or else.
inFilePipe.close()
outFilePipe.close()
```

Our program has all of the slickness of a nice UNIX command.

```
$ ./copy.py in.txt nowhere.txt
$ cat in.txt
I know am an innie
I therefore collect lint daily
It is my lot in life.
$ cat nowhere.txt
I know am an innie
I therefore collect lint daily
It is my lot in life.
$
```

5 Some Useful Techniques for File Input

We begin by showing some useful methods for `file` objects. Each method is shown with the modes for which it applies.

The file reading mechanism has a *file pointer*, which keeps track of how much of the file has been read. As the file read, the file pointer advances.

You can traverse a file all at once, a line at a time, a character at a time, or in pieces of any size you specify.

5.1 Methods of Traversing Files with `for` loops

A file open for reading has an iterator that shows the file one line at a time. Hence, we can use the `for` loop to traverse a file.

```
f = open("foo.txt", "r")
for k in f:
    ##do something with each line in the file
```

This `for` loop concludes its business when it reaches the end of the file.

Python 2/3 note This next loop is ideal for large files in Python 2. The xreadlines() method reads the lines in seriatum and it terminates when the file ends. In Python 3, this is actually what happens.

```
for k in open("foo.txt").xreadlines():
    ##do stuff to each line
```