Contents

os module and system access

November 9, 2015

Reference: http://docs.python.org/library/os.html - http://docs.python.org/library/os.path.html

1 Running programs with startfile() and system()

1.1 Using base functions

There are two easy ways to run commands, but they have somewhat different behavior. All of these methods are found in the `os` module:
`os.startfile()`. Mimics double-clicking on a file. Syntax:
`os.startfile(CommandOrFile)`
where `CommandOrFile` is a string that can be either:

1. a Windows command or batch file, e.g.,

```
In [ ]: import os

        os.startfile("Notepad.exe")
        os.startfile("Firefox")
        os.startfile(r"\Workers\BatchFiles\MyBatchFile.bat") # Note the use of 'r' here.
```

If the command is not in the current directory or in the system path, you must provide fully qualified name.

2. a file whose type is registered with Windows

gg

```
In [ ]: os.startfile("newdata.xlsx")
        os.startfile("c:\\Geog 349 Research\\Norway.docx") # and without the 'r' we need to escape the
```

This starts the associated application and passes the named file to the application. But there are issues with `os.startfile()`:

1. `os.startfile` returns nothing to Python, we get no information about whether the program was executed, failed, or anything.
2. there is no way to pass arguments in to the program:

```
In [ ]: os.startfile("gdalinfo madison.tiff") #bombs
```

1.2 Using the `os` module:

`os.system(CommandString)`
where `CommandString` is a Python string containing the command to be run, including any arguments. `os.system()` returns a Windows status code. The convention is for **zero** to mean success, and anything else indicates the command failed.
For example, we could run notepad:

1.3 Example 11.1

```
In [ ]: import os

        command = 'notepad.exe NewTextFile.txt'
        print 'ready to start command:',command

        exitCode = os.system(command)

        print 'started command:',command
        print 'exitCode is',exitCode
```

As with `os.startfile()`, for this to work the command `"notepad.exe"` must either be in the Windows Path or in the working directory. Obviously, every Windows installation comes with notepad, so the above will work unless the user has changed the path.

Suppose one of the arguments contains one or more spaces. How can we be sure the operating system does not split the argument into pieces? E.g., suppose our command is:

`C:\Program Files\Mozilla Firefox\firefox`

This won't work:

```
In [ ]: import os
        cmd = '\\program files\\Mozilla Firefox\\firefox'
        os.system(cmd) #bombs
```

Nor is a raw string the answer:

```
In [ ]: import os
        cmd = r'\program files\Mozilla Firefox\firefox'
        os.system(cmd) #bombs
```

The problem is that the space in the path name means that Windows/Python sees the `cmd` as a command (`"c:\program"`) and two arguments (`"files\Mozilla"` and `"Firefox\firefox"`). We want the command to consist of the entire string, including the blanks.

Solution — embed a quoted string within the command:

1.4 Example 11.2

```
In [ ]: cmd = 'c:"\\program files\\Mozilla Firefox\\firefox.exe"'
        print 'ready to start command:',cmd
        os.system(cmd)
        print 'started command:',cmd
```

Now you see why blanks in file names are such a bad idea, or why you should assume every path name includes spaces (since it is a fairly common problem).

`os.startfile()` doesn't have this problem because it doesn't take arguments. All of the string must therefore be a command. So, we can do:

1.5 Example 11.3

```
In [ ]: fileName = 'c:\\program files\\Mozilla Firefox\\firefox.exe'
        print 'ready to start file:',fileName
        os.startfile(fileName)
        print 'started file:',fileName
```

If you don't need to pass arguments and you don't need a return value, `startfile` is just fine. If you want to pass arguments (like with `gdalwarp` or `ogr2ogr` commands) then you need to do something more complicated.

What if we want Python to run multiple commands or a sequence of commands? E.g. `gdal_translate` followed by `gdalinfo`? In this case we can either:

1. stack system calls within the program:

```
In [ ]: os.system(cmd1)
        os.system(cmd2)
        os.system(cmd3)
```

This seems intuitive since it matches what we do in a shell script, however Python does not send the commands sequentially in the same way as the CLI. Because Windows commands can run in parallel, Python doesn't actually wait for `cmd1` to finish before starting `cmd2`. So if `cmd2` requires output from `cmd1` to execute properly, then this process will bomb.

2. Create batch file containing all command and run that batch:
3. Use Python to write `cmd1`,`cmd2`, `cmd3` to a batch file e.g., `Mybatch.bat`
4. Invoke batch file from Python: `os.system(Mybatch.bat)`

1.6 Exercise 11.1:

Find the full path of one of your commonly-used programs. Write a Python program run that program. Verify that your program works: Run the Python program and see if the other program starts.

Go to the Exercise 11.1. PDF on Learn@UW

2 Other Useful os Methods

The following commands do the obvious. They can be placed inside loops or functions whenever necessary.

```
In [ ]: os.remove(file)      #delete a file
        os.rename(oldName, newName) #rename a file
        os.chdir(dir)        #change working directory to dir
        os.mkdir(dir)        #creates new directory
        os.rmdir(dir)        #deletes directory (must be empty)
        os.getcwd(dir)       #returns current working directory
        os.listdir(path)     #returns list of files and directories in path
```

2.1 Example 11.4.

List the files in a pre-defined directory, then run whichever file the user specifies.

```
In [ ]: files = os.listdir("\\g378")

        print "Files are:"
        for i in range(len(files)):
           print i,files[i]

        i = input("Enter number of file to open >")
        f = files[i]

        try:
            os.startfile(f)
        except:
            print "Couldn't run",f
```

The os methods above work on any system and are preferable to using a native operating system command via `os.system()`. For example, either of the following two statements could be used to delete `Madison.tiff` on a Windows computer:

```
In [ ]: os.system("del Madison.tiff")
        os.remove("Madison.tiff")
```

However, the first version won't work on a Unix machine because on Unix systems the command to delete a file is `rm`, not `del`.

2.2 Exercise 11.2:

Write code to find all programs and files ithin a directory and add the prefix "new_".

Note: this program will rename files, not make new copies. As safeguard, first create a test folder and copy some files to the folder for the experiment. DO NOT use existing folders containing files you can't afford to lose.

Go to the Exercise 11.2. PDF on Learn@UW

3 The `os.path` module

A submodule of os named 'path' is very useful. Some os.path methods:

```
In [ ]: os.path.basename(path)  #return just the file name of path
        os.path.dirname(path)   #return just the directory name of path
        os.path.split(path)     #return dirname and filename as tuple
        os.path.getatime(file)  #return last access time
        os.path.getctime(file)  #return creation time
        os.path.getsize(file)   #returns filesize
        os.path.exists(path)    #return True if path exists
                                #(works for file or directory)
```

3.1 Example 11.5:

Problem: I have C++ programming project with about 50 files of source code (.*cpp and* .h files). These program source files are among hundreds of other files in project. I want to know the total number of lines and total bytes in source files.

Pseudo-code:

1. Use os.listdir() to get list of file names
2. Initialize line count and byte count to zero
3. For each source file name in list:
4. Use os.path.getsize() to find the file's size, add that to total bytes
5. Count the lines in the file and add to total line count
6. Print total number of lines and total bytes

But how would we "Count the lines"?

More detailed pseudo-code:

1. Use os.listdir() to get list of file names

2. Initialize line count and byte count to zero
3. For each source file name in list:
4. Use os.path.getsize() to find the file's size, add to total bytes
5. Count the lines and add to total line count
6. Open the file
7. Read all lines into list
8. Add length of list to line count
9. Close the file
10. Print total number of lines and total bytes

3.2 Example 11.5

```
In [ ]: sourceDir = r'c:\ufiles\solim\3dm\_code'
        fileNames = os.listdir(sourceDir)

        print 'There are',len(fileNames),'files, counting lines...'

        lineCount = 0
        byteCount = 0
        for fileName in fileNames:

            fileName = fileName.strip()

            if fileName.endswith('.h') or fileName.endswith('.cpp'):

               byteCount += os.path.getsize(sourceDir + '\\'+ fileName)

               f = open(sourceDir + '\\'+ fileName,'rt')
               tempStrings = f.readlines()
               lineCount += len(tempStrings)
               f.close()

        print 'There are',lineCount,'total lines.'
        print 'There are',byteCount,'total bytes.'
```

3.3 Exercise 11.3:

Print out all the names of `.tif` files in a folder. Also print out the names of the two `.tif` files with largest and smallest size. [Hint: you will need to get the size of each file and keep track of the largest and smallest value encountered. Don't worry about ties for this exercise]

Go to the Exercise 11.3. PDF on Learn@UW

4 Closure

`os.system()` is fine for uncomplicated tasks. For more control, use the methods in the subprocess module. Methods in subprocess will let you wait for a command to complete before executing the next Python statement, redirect console output and error messages, spawn parallel processes, and create other more complex behaviors.