Doing operating system tasks in Python

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Mar 23, 2015

Python has extensive support for operating system tasks, such as file and folder management. The great advantage of doing operating system tasks in Python and not directly in the operating system is that the Python code works uniformly on Unix/Linux, Windows, and Mac (there are exceptions, but they are few). Below we list some useful operations that can be done inside a Python program or in an interactive session.

Make a folder. Python applies the term directory instead of folder. The equivalent of the Unix `mkdir mydir` is

```
import os
os.mkdir('mydir')
```

Ordinary files are created by the `open` and `close` functions in Python.

Make intermediate folders. Suppose you want to make a subfolder under your home folder:

```
    $HOME/python/project1/temp
```

but the intermediate folders `python` and `project1` do not exist. This requires each new folder to be made separately by `os.mkdir`, or you can make all folders at once with `os.makedirs`:

```
foldername = os.path.join(os.environ['HOME'], 'python',
                          'project1', 'temp')
os.makedirs(foldername)
```

With `os.environ[var]` we can get the value of any environment variable `var` as a string. The `os.path.join` function joins folder names and a filename in a platform-independent way.

Move to a folder. The `cd` command reads `os.chdir` and `cwd` is `os.getcwd`:

```
origfolder = os.getcwd()  # get name of current folder
os.chdir(foldername)      # move ("change directory")
...
os.chdir(origfolder)      # move back
```

Rename a file or folder. The cross-platform `mv` command is

```
os.rename(oldname, newname)
```

List files. Unix wildcard notation can be used to list files. The equivalent of `ls *.py` and `ls plot*[1-4]*.dat` reads

```
import glob
filelist1 = glob.glob('*.py')
filelist2 = glob.glob('plot*[1-4]*.dat')
```

List all files and folders in a folder. The counterparts to `ls -a mydir` and just `ls -a` are

```
filelist1 = os.listdir('mydir')
filelist1 = os.listdir(os.curdir)  # current folder (directory)
filelist1.sort()                   # sort alphabetically
```

Check if a file or folder exists. The widely used constructions in Unix scripts for testing if a file or folder exist are `if [ -f $filename ]; then` and `if [ -d $dirname ]; then`. These have very readable counterparts in Python:

```
if os.path.isfile(filename):
    inputfile = open(filename, 'r')
    ...

if os.path.isdir(dirnamename):
    filelist = os.listdir(dirname)
    ...
```

Remove files. Removing a single file is done with `os.rename`, and a loop is required for doing `rm tmp_*.df`:

```
import glob
filelist = glob.glob('tmp_*.pdf')
for filename in filelist:
    os.remove(filename)
```

Remove a folder and all its subfolders. The `rm -rf mytree` command removes an entire folder tree. In Python, the cross-platform valid command becomes

```
import shutil
shutil.rmtree(foldername)
```

It goes without saying that this command must be used with great care!

Copy a file to another file or folder. The `cp fromfile tofile` construction applies `shutil.copy` in Python:

```
shutil.copy('fromfile', 'tofile')
```

Copy a folder and all its subfolders. The recursive copy command `cp -r` for folder trees is in Python expressed by `shell.copytree`:

```
shutil.copytree(sourcefolder, destination)
```

Run any operating system command. The simplest way of running another program from Python is to use `os.system`:

```
cmd = 'python myprog.py 21 --mass 4'   # command to be run
failure = os.system(cmd)
if failure:
    print 'Execution of "%s" failed!\n' % cmd
    sys.exit(1)
```

The recommended way to run operating system commands is to use the `subprocess` module. The above command is equivalent to

```
import subprocess
cmd = 'python myprog.py 21 --mass 4'
failure = subprocess.call(cmd, shell=True)

# or
failure = subprocess.call(
            ['python', 'myprog.py', '21', '--mass', '4'])
```

The output of an operating system command can be stored in a string object:

```
try:
    output = subprocess.check_output(cmd, shell=True,
                                     stderr=subprocess.STDOUT)
except subprocess.CalledProcessError:
    print 'Execution of "%s" failed!\n' % cmd
    sys.exit(1)

# Process output
for line in output.splitlines():
    ...
```

The `stderr` argument ensures that the `output` string contains everything that the command `cmd` wrote to both standard output and standard error.

The constructions above are mainly used for running stand-alone programs. Any file or folder listing or manipulation should be done by the functionality in the `os` and `shutil` modules.

Split file or folder name. Given `data/file1.dat` as a file path relative to the home folder `/users/me` (`$HOME/data/file1.dat` in Unix). Python has tools for extracting the complete folder name `/users/me/data`, the basename `file1.dat`, and the extension `.dat`:

```
>>> path = os.path.join(os.environ['HOME'], 'data', 'file1.dat')
>>> path
'/users/me/data/file1.dat'
>>> foldername, basename = os.path.split(path)
>>> foldername
'/users/me/data'
>>> basename
'file1.dat'
>>> stem, ext = os.path.splitext(basename)
>>> stem
'file1'
>>> ext
'.dat'
>>> outfile = stem + '.out'
>>> outfile
'file1.out'
```

Index