# Doing operating system tasks in Python[1]

**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.

## Contents



[1]The material in this document is taken from a chapter in the book *A Primer on Scientific Programming with Python*, 4th edition, by the same author, published by Springer, 2014.

## 0.1 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.

## 0.2 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.

## 0.3 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
```

## 0.4 Rename a file or folder

The cross-platform `mv` command is

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

## 0.5 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')
```

## 0.6 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
```

## 0.7 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)
    ...
```

## 0.8 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)
```

## 0.9 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!

## 0.10 Copy a file to another file or folder

The `cp fromfile tofile` construction applies `shutil.copy` in Python:

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

## 0.11 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)
```

## 0.12 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.

## 0.13 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

# PYTHON OS FILE/DIRECTORY METHODS

 

The **os** module provides a big range of useful methods to manipulate files and directories. Most of the useful methods are listed here:

| SN | Methods with Description |
|---|---|
| 1 | os.access*path, mode*<br><br>Use the real uid/gid to test for access to path. |
| 2 | os.chdir*path*<br><br>Change the current working directory to path |
| 3 | os.chflags*path, flags*<br><br>Set the flags of path to the numeric flags. |
| 4 | os.chmod*path, mode*<br><br>Change the mode of path to the numeric mode. |
| 5 | os.chown*path, uid, gid*<br><br>Change the owner and group id of path to the numeric uid and gid. |
| 6 | os.chroot*path*<br><br>Change the root directory of the current process to path. |
| 7 | os.close*fd*<br><br>Close file descriptor fd. |
| 8 | os.closerange$fd_{low}$, $fd_{high}$<br><br>Close all file descriptors from fd_low *inclusive* to fd_high *exclusive*, ignoring errors. |
| 9 | os.dup*fd*<br><br>Return a duplicate of file descriptor fd. |
| 10 | os.dup2*fd, fd2* |

Duplicate file descriptor fd to fd2, closing the latter first if necessary.

11

os.fchdir*fd*

Change the current working directory to the directory represented by the file descriptor fd.

12

os.fchmod*fd, mode*

Change the mode of the file given by fd to the numeric mode.

13

os.fchown*fd, uid, gid*

Change the owner and group id of the file given by fd to the numeric uid and gid.

14

os.fdatasync*fd*

Force write of file with filedescriptor fd to disk.

15

os.fdopen*fd*[, *mode*[, *bufsize*]]

Return an open file object connected to the file descriptor fd.

16

os.fpathconf*fd, name*

Return system configuration information relevant to an open file. name specifies the configuration value to retrieve.

17

os.fstat*fd*

Return status for file descriptor fd, like stat.

18

os.fstatvfs*fd*

Return information about the filesystem containing the file associated with file descriptor fd, like statvfs.

19

os.fsync*fd*

Force write of file with filedescriptor fd to disk.

20

os.ftruncate*fd, length*

Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size.

21

os.getcwd

Return a string representing the current working directory.

22

os.getcwdu

Return a Unicode object representing the current working directory.

23

os.isatty*fd*

Return True if the file descriptor fd is open and connected to a tty−*like* device, else False.

24

os.lchflags*path, flags*

Set the flags of path to the numeric flags, like chflags, but do not follow symbolic links.

25

os.lchmod*path, mode*

Change the mode of path to the numeric mode.

26

os.lchown*path, uid, gid*

Change the owner and group id of path to the numeric uid and gid. This function will not follow symbolic links.

27

os.link*src, dst*

Create a hard link pointing to src named dst.

28

os.listdir*path*

Return a list containing the names of the entries in the directory given by path.

29

os.lseek*fd, pos, how*

Set the current position of file descriptor fd to position pos, modified by how.

30

os.lstat*path*

Like stat, but do not follow symbolic links.

31

os.major*device*

Extract the device major number from a raw device number.

32

os.makedev*major, minor*

Compose a raw device number from the major and minor device numbers.

33

os.makedirs*path*[, *mode*]

Recursive directory creation function.

34

os.minor*device*

Extract the device minor number from a raw device number .

35

os.mkdir*path*[, *mode*]

Create a directory named path with numeric mode mode.

36

os.mkfifo*path*[, *mode*]

Create a FIFO *anamedpipe* named path with numeric mode mode. The default mode is 0666 *octal*.

37

os.mknod*filename*[, *mode* = 0600, *device*]
Create a filesystem node *file, devicespecialfileornamedpipe* named filename.

38

os.open*file, flags*[, *mode*]

Open the file file and set various flags according to flags and possibly its mode according to mode.

39

os.openpty

Open a new pseudo-terminal pair. Return a pair of file descriptors *master, slave* for the pty and the tty, respectively.

40

os.pathconf*path, name*

Return system configuration information relevant to a named file.

41

os.pipe

Create a pipe. Return a pair of file descriptors *r, w* usable for reading and writing, respectively.

42

os.popen*command*[, *mode*[, *bufsize*]]

Open a pipe to or from command.

43

os.read*fd, n*

Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned.

44

os.readlink*path*

Return a string representing the path to which the symbolic link points.

45

os.remove*path*

Remove the file path.

46

os.removedirs*path*

Remove directories recursively.

47

os.rename*src, dst*

Rename the file or directory src to dst.

48

os.renames*old, new*

Recursive directory or file renaming function.

49

os.rmdir*path*

Remove the directory path

50

os.stat*path*

Perform a stat system call on the given path.

51

os.stat_float_times[*newvalue*]
Determine whether stat_result represents time stamps as float objects.

52

os.statvfs*path*

Perform a statvfs system call on the given path.

53

os.symlink*src, dst*

Create a symbolic link pointing to src named dst.

54

os.tcgetpgrp*fd*

Return the process group associated with the terminal given by fd
*anopenfiledescriptorasreturnedbyopen*().

55

os.tcsetpgrp*fd, pg*

Set the process group associated with the terminal given by fd $anopenfiledescriptorasreturnedbyopen()$ to pg.

56

os.tempnam[*dir*[, *prefix*]]

Return a unique path name that is reasonable for creating a temporary file.

57

os.tmpfile

Return a new file object opened in update mode $w+b$.

58

os.tmpnam

Return a unique path name that is reasonable for creating a temporary file.

59

os.ttyname*fd*

Return a string which specifies the terminal device associated with file descriptor fd. If fd is not associated with a terminal device, an exception is raised.

60

os.unlink*path*

Remove the file path.

61

os.utime*path*, *times*

Set the access and modified times of the file specified by path.

62

os.walk*top*[, *topdown = True*[, *onerror = None*[, *followlinks = False*]]]

Generate the file names in a directory tree by walking the tree either top-down or bottom-up.

63

os.write*fd*, *str*

Write the string str to file descriptor fd. Return the number of bytes actually written.