

Using Linux Commands in a Python Script

Tolu Oyeniyi

Northeast Cyberteam

March 2, 2022

Table of Contents

Subprocess vs. os.....	3
Subprocess Module.....	3
os Module.....	3
Subprocess Example Script.....	4
Step 1 - Imports.....	4
Step 2 – Import Python Module Needed to Run a Module.....	4
Step 3 – Loading a Module in Python.....	6
Step 4 – Linux Commands	6
Step 5 – Subprocess.run.....	6
Additional Input Commands.....	7
Multiple User Input.....	8
Pexpect.....	8

Subprocess vs. os

This section will define what the subprocess module is and what the os module is in python, and highlight the key difference between both modules.

Subprocess Module

Subprocess is a module that can be used for running shell commands (which allow you to run Linux commands) within your python script. Subprocess is a newer replacement for the os module in python and is recommended because it has a more secure approach to running shell commands than the os module which is susceptible to many vulnerabilities. For example, subprocess does not inherit the user's environment variables by default.

os Module

os is a module that can be used to run shell commands in python. It is very simple to use but has been deprecated because of its security risks that subjects it to many vulnerabilities. For example, os by default, inherits the user's environment variables. An example of an os script within a python script is shown below:

practiceFile.py: python file using the os function to run shell commands

```
import os
cmd = 'module avail rclone' #an example of a Linux command that is saved in a variable
os.system(cmd) #os function that will run Linux commands
```

in Linux terminal: run the python file

```
$ python3 practiceFile.py
```

in Linux terminal: result of running python file

```
----- /share/module.7/utilities -----
rclone/1.50.2   rclone/1.55.0   rclone/1.57.0 (D)

Where:
D:  Default Module

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 ..." to search for all possible modules matching any of the "keys".
```

Subprocess Example Script

In this example, we will be using a python module to run a module called rclone. Then we will be using shell commands to directly run the rclone codes. Note, step 1, 4, and 5 are the codes mainly needed to run Linux commands in a python script; if you are not loading another module then don't worry about steps 2 and 3.

```
19
20 #step 1
21 import subprocess, os
22
23 #step 2
24 exec(open(f'{os.environ["MODULESHOME"]}/init/env_modules_python.py').read())
25
26 #step3
27 module("load", "rclone")
28
29
30 #step 4
31 shellCmd = 'rclone config'
32 cmd = shellCmd.split() #is really cmd=['rclone','config']
33
34 #step 5
35 subprocess.run(cmd, env=os.environ, input = b'q\n') #note: can use a variable --> inputValue = b'q\n' and input = inputValue
36
37
```

Step 1 - Imports

When using the subprocess module, the module alone can be imported, but if you want subprocess to inherit the user environment, then the os module needs to be imported too.

Step 2 – Import Python Module Needed to Run a Module

What is lmod? lmod is a program that implements environment modules, it manages the user environment. To import the init python script provided by lmod, you first need to find the

init python script within your system, then you can open and execute the script through the `exec` command.

The file path in the `exec` code contains the module environment variable, `$MODULESHOME`. Note: a possible error that could come up is that the module environment variable name is called something else.

```
#step 2
exec(open(f'{os.environ["MODULESHOME"]}/init/env_modules_python.py').read())
```

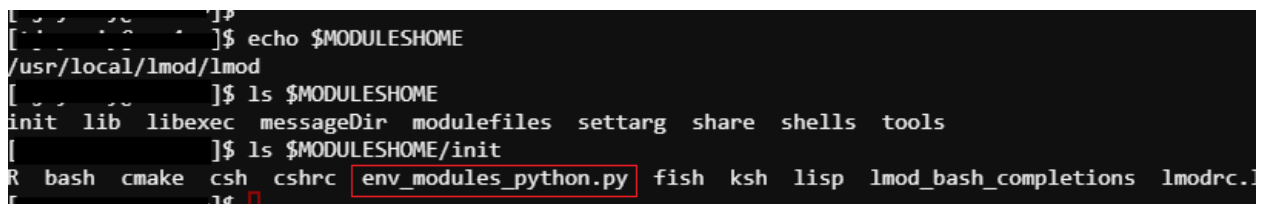
Older versions of `lmod` use the `init python.py` script:

```
exec(open(f'{os.environ["MODULESHOME"]}/init/python.py').read())
```

Newer versions of `lmod` use the `init env_modules_python.py` script:

```
exec(open(f'{os.environ["MODULESHOME"]}/init/env_modules_python.py').read())
```

To find the python script name your system has, you can use the module environment variable, `$MODULESHOME`, to list all the initialization scripts. The `'ls $MODULESHOME/init'` command does the job.



```
[...]  
[...]$ echo $MODULESHOME  
/usr/local/lmod/lmod  
[...]$ ls $MODULESHOME  
init lib libexec messageDir modulefiles settarg share shells tools  
[...]$ ls $MODULESHOME/init  
R bash cmake csh cshrc env_modules_python.py fish ksh lisp lmod_bash_completions lmodrc.  
[...]
```

To avoid constantly checking what the init python script is called, a better solution would be to use a `try and except` statement in your python script to account for both possibilities (of the names) of the init python script.

```
#step 2
try:
    #older version of lmod
    exec(open(f'{os.environ["MODULESHOME"]}/init/python.py').read())
except:
    #newer version of lmod
    exec(open(f'{os.environ["MODULESHOME"]}/init/env_modules_python.py').read())
```

Step 3 – Loading a Module in Python

To run modules to the shell, the module function, ‘module(command_name, command_arg)’ can be used.

Step 4 – Linux Commands

The Linux commands need to be represented as a list of arguments, not just one string; Subprocess expects a list. The commands will be strings. For example, “rclone config” needs to be represented as [“rclone”,”config”]. You can save the list into a variable (cmd = [“rclone”,”config”]) or save the string into a variable then split the string into list and save again.

```
#step 4
shellCmd = 'rclone config'
cmd = shellCmd.split() #is really cmd=['rclone','config']
```

Step 5 – Subprocess.run

The first argument in the subprocess.run command is the list of Linux commands that we are sending to the Shell to run, in the example the cmd variable contains the Linux codes we are trying to run.

The second argument in the subprocess.run command, in the example, tells subprocess to inherit the user environment. Specifically, the ‘env’ keyword argument expects a dictionary of

environment variables and their values, which can be provided by using the `os` module and import the user environment.

```
7
8 #step 1
9 import subprocess, os
10
11 #step 2
12 shellCmd = 'ls -al'
13 cmd = shellCmd.split() #is really cmd=['ls', '-al']
14
15 #step 3
16 subprocess.run(cmd, env=os.environ)
17
```

Additional Input Commands

In our example, the shell command ‘`rclone config`’ requires user input after being run.

```
e) Edit existing remote
n) New remote
d) Delete remote
r) Rename remote
c) Copy remote
s) Set configuration password
q) Quit config
e/n/d/r/c/s/q> 
```

User input can be added by using the ‘`input`’ argument, the third argument, specifically by entering the command (as a string/between quotations) or using a variable containing the string commands, into the input argument of `subprocess.run`

Note: even within Linux, after entering an input, the user will mostly likely hit enter next, thus, the enter key needs to be accounted for within the code. ‘`\n`’ after the string command will signal the enter key. Also, ‘`b`’ can be added before the quotations to tell the system that the string is a bytes string. Another note: only one user input can be entered, not a list of inputs.

```
subprocess.run(cmd, env=os.environ, input = b'q\n')
```

```
e) Edit existing remote
n) New remote
d) Delete remote
r) Rename remote
c) Copy remote
s) Set configuration password
q) Quit config
e/n/d/r/c/s/q> [REDACTED ~]$
```

Multiple User Input

Subprocess only accepts a single input, which means that if you want to accept a lot of user input, you will need to use another package.

Pexpect

Pexpect has good documentation and allows you to send multiple inputs. Using Pexpect might look something like this:

```
import pexpect, os

exec(open(f'{os.environ["MODULESHOME"]}/init/env_modules_python.py').read())
module("load", "rclone")

cmd = "rclone config"
child = pexpect.spawn(cmd)
child.expect('e/n/d/r/c/s/q> ')
child.sendline('q')
```

Pexpect: <https://pexpect.readthedocs.io/en/stable/overview.html>