Calling an external command in Python

Ask Question

How can I call an external command (as if I'd typed it at the Unix shell or Windows command prompt) from within a Python script?

python shell command subprocess external

edited Sep 22 '14 at 12:40
igaurav
1,470 1 13 24

asked Sep 18 '08 at 1:35
freshWoWer
13.4k 9 25 32

hey here's a good tutorial on integrating python with shell: dreamsyssoft.com/python-scripting-tutorial/shell-tutorial.php – Triton Man Mar 19 '15 at 3:46

6 @TritonMan: it is not a good tutorial. Use `for line in proc.stdout:` (or `for line in`

44 Answers

1 2 next

Look at the subprocess module in the standard library:

```
from subprocess import call
call(["ls", "-l"])
```

The advantage of **subprocess** vs **system** is that it is more flexible (you can get the stdout, stderr, the "real" status code, better error handling, etc...).

The official docs recommend the **subprocess** module over the alternative os.system():

> The **subprocess** module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using this function [`os.system()`].

The "Replacing Older Functions with the subprocess Module" section in the **subprocess** documentation may have some helpful recipes.

Official documentation on the **subprocess** module:

- Python 2 - subprocess
- Python 3 - subprocess

edited Feb 14 at 13:05
chobok
271 1 18

answered Sep 18 '08 at 1:39
David Cournapeau
39.8k 5 49 63

145 Can't see why you'd use os.system even for quick/dirty/one-time. subprocess seems so much better. – nosklo May 26 '09 at 16:40

30 I agree completely that subprocess is better. I just had to write a quick script like this to run on an old server with Python 2.3.4 which does not have the subprocess module. – Liam Jul 29 '10 at 14:58

46 here are the subprocess docs – daonb Mar 21 '12 at 7:10

6 call (..) Gave me an error on oython 2.7.6 : Traceback (most recent call last): File "E:\Ajit\MyPython\synonyms1.py", line 27, in <module> call("dir") <br/> File "C:\Python27\lib\subprocess.py", line 524, in call return Popen(*popenargs, **kwargs).wait() <br/> File "C:\Python27\lib\subprocess.py", line 711, in **init** errread, errwrite)<br/> File "C:\Python27\lib\subprocess.py", line 948, in _execute_child startupinfo)<br/> WindowsError: [Error 2] The system cannot find the file specified – goldenmean Nov 10 '13 at 18:22

54 @goldenmean: my guess, there is no `ls.exe` on Windows. Try `call("dir", shell=True)` – J.F. Sebastian Dec 21 '13 at 4:50

Here's a summary of the ways to call external programs and the advantages and disadvantages of each:

1. `os.system("some_command with args")` passes the command and arguments to your system's shell. This is nice because you can actually run multiple commands at once in this manner and set up pipes and input/output redirection. For example:

   ```
   os.system("some_command < input_file | another_command > output_file")
   ```

   However, while this is convenient, you have to manually handle the escaping of shell characters such as spaces, etc. On the other hand, this also lets you run commands which are simply shell commands and not actually external programs. See the documentation.

2. `stream = os.popen("some_command with args")` will do the same thing as `os.system` except that it gives you a file-like object that you can use to access standard input/output for that process. There are 3 other variants of popen that all handle the i/o slightly differently. If you pass everything as a string, then your command is passed to the shell; if you pass them as a list then you don't need to worry about escaping anything. See the

   comprehensive. For example, you'd say:

   ```
   print subprocess.Popen("echo Hello World", shell=True,
   stdout=subprocess.PIPE).stdout.read()
   ```

   instead of:

   ```
   print os.popen("echo Hello World").read()
   ```

   but it is nice to have all of the options there in one unified class instead of 4 different popen functions. See the documentation.

4. The `call` function from the `subprocess` module. This is basically just like the `Popen` class and takes all of the same arguments, but it simply waits until the command completes and gives you the return code. For example:

   ```
   return_code = subprocess.call("echo Hello World", shell=True)
   ```

   See the documentation.

5. If you're on Python 3.5 or later, you can use the new `subprocess.run` function, which is a lot like the above but even more flexible and returns a `CompletedProcess` object when the command finishes executing.

6. The os module also has all of the fork/exec/spawn functions that you'd have in a C program, but I don't recommend using them directly.

The `subprocess` module should probably be what you use.

Finally please be aware that for all methods where you pass the final command to be executed by the shell as a string and you are responsible for escaping it. **There are serious security implications** if any part of the string that you pass can not be fully trusted. For example, if a user is entering some/any part of the string. If you are unsure, only use these methods with constants. To give you a hint of the implications consider this code:

```
print subprocess.Popen("echo %s " % user_input, stdout=PIPE).stdout.read()
```

and imagine that the user enters "my mama didnt love me && rm -rf /".

edited Apr 11 '16 at 19:18
the Tin Man
115k 22 142 211

answered Sep 18 '08 at 13:11
Eli Courtwright
89.1k 46 177 233

8 You didn't mention the commands module – Casebash May 16 '10 at 0:39

108 @Casebash: I didn't bother mentioning it because the documentation states that `The subprocess module provides more powerful facilities for spawning new processes and retrieving their results. Using the subprocess module is preferable to using the commands module.` I similarly didn't mention the popen2 module because it's also obsolete, and both it and the `commands` module are actually gone in Python 3.0 and later. In lieu of editing my answer, I'll let these comment be the way in which these modules are mentioned. – Eli Courtwright May 16 '10 at 13:16

15 Great article on the use of subprocess here : doughellmann.com/PyMOTW/subprocess – PhoebeB Nov 15 '10 at 10:31

11 commands module is deprecated now. – jldupont Jan 21 '12 at 0:35

13 For many cases you don't need to instantiate a Popen object directly, you can use `subprocess.check_call` and `subprocess.check_output` – simao Jun 12 '12 at 21:36

I typically use:

```
import subprocess

p = subprocess.Popen('ls', shell=True, stdout=subprocess.PIPE,
stderr=subprocess.STDOUT)
for line in p.stdout.readlines():
    print line,
retval = p.wait()
```

You are free to do what you want with the `stdout` data in the pipe. In fact, you can simply omit those parameters (`stdout=` and `stderr=`) and it'll behave like `os.system()` .

edited Dec 10 '16 at 13:22
Jim Fasarakis-Hilliard
35.7k 10 69 120

answered Sep 18 '08 at 18:20
EmmEff
3,487 2 10 17

Questions Jobs Documentation BETA Tags Users Log In Sign Up

`''): print line,` – J.F. Sebastian Nov 16 '12 at 14:12

Could you elaborate on what you mean by "if there is no buffering issues"? If the process blocks definitely, the subprocess call also blocks. The same could happen with my original example as well. What else could happen with respect to buffering? – EmmEff Nov 17 '12 at 13:25

9 the child process may use block-buffering in non-interactive mode instead of line-buffering so `p.stdout.readline()` (note: no `s` at the end) won't see any data until the child fills its buffer. If the child doesn't produce much data then the output won't be in real time. See the second reason in Q: Why not just use a pipe (popen())?. Some workarounds are provided in this answer (pexpect, pty, stdbuf) – J.F. Sebastian Nov 17 '12 at 13:51

2 @Paul: If your code produces unexpected results then you could create a complete minimal code example that reproduces the problem and post it as a new question. Mention what do you expect to happen and what happens instead. – J.F. Sebastian Apr 10 '13 at 18:41

2 All right took your advice stackoverflow.com/questions/15945585/… thanks! – Paul Apr 11 '13 at 9:49

Some hints on detaching the child process from the calling one (starting the child process in background).

Suppose you want to start a long task from a CGI-script, that is the child process should live longer than the CGI-script execution process.

The classical example from the subprocess module docs is:

```
import subprocess
import sys

# some code here

pid = subprocess.Popen([sys.executable, "longtask.py"]) # call subprocess

# some more code here
```

The idea here is that you do not want to wait in the line 'call subprocess' until the longtask.py is finished. But it is not clear what happens after the line 'some more code here' from the example.

My target platform was freebsd, but the development was on windows, so I faced the problem on windows first.

On windows (win xp), the parent process will not finish until the longtask.py has finished its work. It is not what you want in CGI-script. The problem is not specific to Python, in PHP community the problems are the same.

The solution is to pass DETACHED_PROCESS flag to the underlying CreateProcess function in win API. If you happen to have installed pywin32 you can import the flag from the win32process module, otherwise you should define it yourself:

```
DETACHED_PROCESS = 0x00000008

pid = subprocess.Popen([sys.executable, "longtask.py"],
                       creationflags=DETACHED_PROCESS).pid
```

/* *UPD 2015.10.27* @eryksun in a comment below notes, that the semantically correct flag is CREATE_NEW_CONSOLE (0x00000010) */

On freebsd we have another problem: when the parent process is finished, it finishes the child processes as well. And that is not what you want in CGI-script either. Some experiments showed that the problem seemed to be in sharing sys.stdout. And the working solution was the following:

```
pid = subprocess.Popen([sys.executable, "longtask.py"], stdout=subprocess.PIPE,
stderr=subprocess.PIPE, stdin=subprocess.PIPE)
```

I have not checked the code on other platforms and do not know the reasons of the behaviour on freebsd. If anyone knows, please share your ideas. Googling on starting background processes in Python does not shed any light yet.

edited Oct 27 '15 at 10:32 answered Feb 12 '10 at 10:15
newtover
17.5k 5 49 66

thanks for the answer! i noticed a possible "quirk" with developing py2exe apps in pydev+eclipse. i was able to tell that the main script was not detached because eclipse's output window was not terminating; even if

Windows gotcha: even though I spawned process with DETACHED_PROCESS, when I killed my Python daemon all ports opened by it wouldn't free until all spawned processes terminate. WScript.Shell solved all my problems. Example here: pastebin.com/xGmuvwSx – Alexey Lebedev Apr 16 '12 at 10:04

1 you might also need CREATE_NEW_PROCESS_GROUP flag. See Popen waiting for child process even when the immediate child has terminated – J.F. Sebastian Nov 16 '12 at 14:16

1 The following is incorrect: "[o]n windows (win xp), the parent process will not finish until the longtask.py has finished its work". The parent will exit normally, but the console window (conhost.exe instance) only closes when the last attached process exits, and the child may have inherited the parent's console. Setting `DETACHED_PROCESS` in `creationflags` avoids this by preventing the child from inheriting or creating a console. If you instead want a new console, use `CREATE_NEW_CONSOLE` (0x00000010). – eryksun Oct 27 '15 at 0:27

1 I didn't mean that executing as a detached process is incorrect. That said, you may need to set the standard handles to files, pipes, or `os.devnull` because some console programs exit with an error otherwise. Create a new console when you want the child process to interact with the user concurrently with the parent process. It would be confusing to try to do both in a single window. – eryksun Oct 27 '15 at 17:37

I'd recommend using the subprocess module instead of os.system because it does shell escaping for you and is therefore much safer: http://docs.python.org/library/subprocess.html

```
subprocess.call(['ping', 'localhost'])
```

edited Apr 4 '14 at 19:46
cincodenada
1,952 11 28

answered Sep 18 '08 at 1:42
sirwart
1,133 7 7

And subprocess will allow you to easily attach to the input/output streams of the process, etc. – Joe Skora Sep 18 '08 at 2:14

5 subprocess doesn't do shell escaping for you because it avoids using the shell entirely. It actually means that startup is a little faster and there's less overhead. – habnabit Sep 18 '08 at 22:53

```
import os
cmd = 'ls -al'
os.system(cmd)
```

If you want to return the results of the command, you can use `os.popen`. However, this is deprecated since version 2.6 in favor of the subprocess module, which other answers have covered well.

edited Jan 26 '16 at 16:53
Patrick M
5,054 5 32 73

answered Sep 18 '08 at 1:37
Alexandra Franks
1,495 12 19

1 popen is deprecated in favor of subprocess. – Fox Wilson Aug 8 '14 at 0:22

```
import os
os.system("your command")
```

Note that this is dangerous, since the command isn't cleaned. I leave it up to you to google for the relevant docs on the 'os' and 'sys' modules. There are a bunch of functions (exec* , spawn*) that will do similar things.

answered Sep 18 '08 at 1:37
nimish
1,772 2 13 22

Check "pexpect" python library, too. It allows for interactive controlling of external programs/commands, even ssh, ftp, telnet etc. You can just type something like:

```
child = pexpect.spawn('ftp 192.168.0.24')

child.expect('(?i)name .*: ')

child.sendline('anonymous')
```

edited Dec 15 '11 at 22:53
zellus
8,902 5 28 51

answered Oct 7 '10 at 7:09
athanassis
471 4 3

I always use `fabric` for this things like:

```
from fabric.operations import local
result = local('ls', capture=True)
print "Content:/n%s" % (result, )
```

But this seem to be a good tool: `sh` (Python subprocess interface).

Look an example:

```
from sh import vgdisplay
print vgdisplay()
print vgdisplay('-v')
print vgdisplay(v=True)
```

edited Jul 23 '13 at 18:28
Eric
54k 25 121 239

answered Mar 13 '12 at 0:12
Jorge E. Cardona
47.9k 2 20 32

2 sh is superior to subprocess module. It allows a better shell integration – Yauhen Yakimovich May 23 '13 at 17:39

If what you need is the output from the command you are calling, then you can use subprocess.check_output (Python 2.7+).

```
>>> subprocess.check_output(["ls", "-l", "/dev/null"])
'crw-rw-rw- 1 root root 1, 3 Oct 18  2007 /dev/null\n'
```

Also note the shell parameter.

> If shell is `True`, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of ~ to a user's home directory. However, note that Python itself offers implementations of many shell-like features (in particular, `glob`, `fnmatch`, `os.walk()`, `os.path.expandvars()`, `os.path.expanduser()`, and `shutil`).

edited Dec 10 '16 at 13:24
Jim Fasarakis-Hilliard
35.7k 10 69 120

answered Apr 28 '11 at 20:29
Facundo Casco
4,914 3 30 52

This is how I run my commands. This code has everything you need pretty much

```
from subprocess import Popen, PIPE
cmd = "ls -l ~/"
p = Popen(cmd , shell=True, stdout=PIPE, stderr=PIPE)
out, err = p.communicate()
print "Return code: ", p.returncode
print out.rstrip(), err.rstrip()
```

edited Oct 28 '12 at 5:44

answered Oct 28 '12 at 5:14
Usman Khan
307 3 3

Passing commands as strings is normally a bad idea – Eric Jul 23 '13 at 18:29

1 I think it's acceptable for hard-coded commands, if it increases readability. – Adam Matan Apr 2 '14 at 13:07

Thanks. Coming from Perl and Ruby, Python is a PITA when it comes to running commands. Read a lot of solutions. Like yours with popen. – sam Sep 12 '16 at 22:50

```
from subprocess import call
call(['ls', '-l'])
```

It is the recommended standard way. However, more complicated tasks (pipes, output, input, etc.) can be tedious to construct and write.

Note: shlex.split can help you to parse the command for `call` and other `subprocess` functions in case you don't want (or you can't!) provide them in form of lists:

```
import shlex
from subprocess import call
call(shlex.split('ls -l'))
```

With External Dependencies

If you do not mind external dependencies, use plumbum:

```
from plumbum.cmd import ifconfig
print(ifconfig['wlan0']())
```

It is the best `subprocess` wrapper. It's cross-platform, i.e. it works on both Windows and Unix-like systems. Install by `pip install plumbum`.

Another popular library is sh:

```
from sh import ifconfig
print(ifconfig('wlan0'))
```

However, `sh` dropped Windows support, so it's not as awesome as it used to be. Install by `pip install sh`.

edited Aug 12 '16 at 13:09

answered Apr 11 '13 at 17:17
Honza Javorek
2,273 1 20 43

without the output of result

```
import os
os.system("your command here")
```

with output of result

```
import commands
commands.getoutput("your command here")
or
commands.getstatusoutput("your command here")
```

answered Apr 18 '13 at 1:09
Zuckonit
369 2 14

2 I like the part `with output of result` . I needed this for using in sublime console. – Ramsharan Oct 19 '13 at 9:03

Update:

`subprocess.run` is the recommended approach as of Python 3.5 if your code does not need to maintain compatibility with earlier Python versions. It's more consistent and offers similar ease-of-use as Envoy. (Piping isn't as straightforward though. See this question for how.)

Here's some examples from the docs.

Run a process:

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)
```

Raise on failed run:

Capture output:

```
>>> subprocess.run(["ls", "-l", "/dev/null"], stdout=subprocess.PIPE)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n')
```

Original answer:

I recommend trying Envoy. It's a wrapper for subprocess, which in turn aims to replace the older modules and functions. Envoy is subprocess for humans.

Example usage from the readme:

```
>>> r = envoy.run('git config', data='data to pipe in', timeout=2)

>>> r.status_code
129
>>> r.std_out
'usage: git config [options]'
>>> r.std_err
''
```

Pipe stuff around too:

```
>>> r = envoy.run('uptime | pbcopy')

>>> r.command
'pbcopy'
>>> r.status_code
0

>>> r.history
[<Response 'uptime'>]
```

edited Oct 3 '15 at 4:30

answered Nov 15 '12 at 17:13
Joe
7,750 6 36 57

4

note: `subprocess.run()` that ignores non-zero exit status by default is a regression compared to `subprocess.check_call()` or `subprocess.check_output()` . `python -mthis` : *"Errors should never pass silently. Unless explicitly silenced."* – J.F. Sebastian Oct 3 '15 at 7:19

Thanks man envoy is working better than subprocess, You saved my day. The command which is command = "ansible-playbook playbook.yaml --extra-vars=\"esxi_host={0} extravar1={1} extravar2={2} extravar3={3}\"".format(extravar1,extravar2,extravar3) and r.envoy.run(command) – itirazimvar Sep 1 '16 at 12:54

There is also Plumbum

```
>>> from plumbum import local
>>> ls = local["ls"]
>>> ls
LocalCommand(<LocalPath /bin/ls>)
>>> ls()
u'build.py\ndist\ndocs\nLICENSE\nplumbum\nREADME.rst\nsetup.py\ntests\ntodo.txt\n'
>>> notepad = local["c:\\windows\\notepad.exe"]
>>> notepad()                                   # Notepad window pops up
u''                                             # Notepad window is closed by user,
command returns
```

answered Oct 10 '14 at 17:41
stuckintheshuck
1,206 13 25

or to add a bit of magic: `from plumbum.cmd import ls, grep; output = (ls | grep['pattern'])()` – J.F. Sebastian Jul 13 '15 at 18:46

`os.system` is OK, but kind of dated. It's also not very secure. Instead, try `subprocess` . `subprocess` does not call sh directly and is therefore more secure than `os.system` .

Get more information here.

edited Dec 10 '16 at 13:25 answered Sep 18 '08 at 1:53

Questions Jobs Documentation BETA Tags Users Log In Sign Up

https://docs.python.org/2/library/subprocess.html

...or for a very simple command:

```
import os
os.system('cat testfile')
```

edited Nov 2 '14 at 4:00
Jonathan Callen
6,846 2 12 35

answered Sep 18 '08 at 1:43
Ben Hoffstein
70.1k 7 81 108

There are lots of different libraries which allow you to call external commands with python. For each library I've given a description and shown an example of calling an external command, the command I used as the example is `ls -l` (list all files). If you want to find out more about any of the libraries I've listed and linked the documentation for each of them.

> **Sources:**

- subprocess: https://docs.python.org/3.5/library/subprocess.html
- shlex: https://docs.python.org/3/library/shlex.html
- os: https://docs.python.org/3.5/library/os.html
- sh: https://amoffat.github.io/sh/
- plumbum: https://plumbum.readthedocs.io/en/latest/
- pexpect: https://pexpect.readthedocs.io/en/stable/
- fabric: http://www.fabfile.org/
- envoy: https://github.com/kennethreitz/envoy
- commands: https://docs.python.org/2/library/commands.html

These are all the libraries:

Hopefully this will help you make a decision on which library to use :)

subprocess

Subprocess allows you to call external commands and connect them to their input/output/error pipes (stdin, stdout and stderr). Subprocess is the default choice for running commands, but sometimes other modules are better.

```
subprocess.run(["ls", "-l"]) # run command
subprocess.run(["ls", "-l"], stdout=subprocess.PIPE) # this will run the command
and return any output
subprocess.run(shlex.split("ls -l")) # you can also use the shlex library to split
the command
```

os

os is used for "operating system dependent functionality". It can also be used to call external commands with `os.system` and `os.popen` (Note: There is also a subprocess.popen). os will always run the shell and is a simple alternative for people who don't need to, or don't know how to use `subprocess.run`.

```
os.system("ls -l") # run command
os.popen("ls -l").read() # this will run the command and return any output
```

sh

sh is a subprocess interface which lets you call programs as if they were functions, this is useful if you want to run a command multiple times.

```
sh.ls("-l") # run command normally
ls_cmd = sh.Command("ls") # save command as a variable
```

plumbum is a library for "script-like" python programs. You can call programs like functions as in sh. Plumbum is useful if you want to run a pipeline without the shell.

```
ls_cmd = plumbum.local("ls -l") # get command
ls_cmd() # run command
```

pexpect

pexpect lets you spawn child applications, control them and find patterns in their output. This is a better alternative to subprocess for commands that expect a tty on unix.

```
pexpect.run("ls -l") # run command as normal
child = pexpect.spawn('scp foo user@example.com:.') # spawns child application
child.expect('Password:') # when this is the output
child.sendline('mypassword')
```

fabric

fabric is a Python 2.5 and 2.7 library, it allows you to execute local and remote shell commands. Fabric is simple alternative for running commands in a secure shell (SSH)

```
fabric.operations.local('ls -l') # run command as normal
fabric.operations.local('ls -l', capture = True) # run command and receive output
```

envoy

envoy is known as "subprocess for humans", it is used as a convenience wrapper around the `subprocess` module.

```
r = envoy.run("ls -l") # run command
r.std_out # get output
```

commands

commands contains wrapper functions for `os.popen` but has been removed from Python 3 since `subprocess` is a better alternative

[EDIT] Based on J.F. Sebastian's comment

edited Nov 26 '16 at 11:44 answered Oct 29 '16 at 14:02
Tom Fuller
1,434 2 14 33

Did I miss any? – Tom Fuller Oct 29 '16 at 14:04

1 It could be useful to specify explicitly *when* and *why* you would prefer one library over another e.g., `pexpect` is useful for commands that expect a tty on Unix, `plumbum` could be use to run a pipeline without invoking the shell, `fabric` is a simple way to run commands via ssh, `subprocess` (unlike `os`) never runs the shell unless you ask—it is the default choice for running external commands, sometimes you might need alternatives. – J.F. Sebastian Nov 2 '16 at 2:21

I've edited my answer based on your feedback :) – Tom Fuller Nov 2 '16 at 16:34

1 `os` "external commands" functions are implemented in terms of `subprocess` internally. It might be useful for people from other languages (`system()` , `popen()` is a common API) who do not need the full power of `subprocess` module and who do not have the time to learn how to use `subprocess.run()` and other subprocess' functionality. – J.F. Sebastian Nov 2 '16 at 17:27

`os.system` has been superseded by the `subprocess` module. Use subproccess instead.

edited Dec 10 '16 at 13:26 answered Sep 18 '08 at 1:43
Jim Fasarakis-Hilliard William Keller
35.7k 10 69 120 **3,760** 19 21

11 Perhaps an example of using `subprocess` ? – Michael Mior Mar 29 '10 at 19:09

2 Given that the accepted answer suggested `subprocess` earlier and with more detail, I see no value to this answer sticking around. – Mark Amery Dec 22 '16 at 21:45

Questions Jobs Documentation BETA Tags Users Log In Sign Up

There is another difference here which is not mentioned above.

`subprocess.Popen` executes the as a subprocess. In my case, I need to execute file which needs to communicate with another program .

I tried subprocess, execution was successful. However could not comm w/ . everything normal when I run both from the terminal.

One more: (NOTE: kwrite behaves different from other apps. If you try below with firefox results will not be the same)

If you try `os.system("kwrite")` , program flow freezes until user closes kwrite. To overcome that I tried instead `os.system(konsole -e kwrite)` . This time program continued to flow but kwrite became the subprocess of the konsole.

Anyone runs the kwrite not being a subprocess (i.e. at the system monitor it must be appear at the leftmost edge of the tree)

edited Dec 10 '16 at 13:25 answered Jan 8 '10 at 21:11
Jim Fasarakis-Hilliard Atinc Delican
35.7k 10 69 120 **111** 1 2

`subprocess.check_call` is convenient if you don't want to test return values. It throws an exception on any error.

answered Jan 18 '11 at 19:21
cdunn2001
8,771 3 36 34

`os.system` does not allow you to store results, so if you want to store results in some list or something `subprocess.call` works.

edited Jun 11 '12 at 22:45 answered Jun 11 '12 at 22:28

Mariusz Jamro
15.8k 8 53 98

Saurabh Bangad
101 1 2

```
import os

cmd = 'ls -al'

os.system(cmd)
```

os - This module provides a portable way of using operating system dependent functionality

for the more os functions here is the documentation.

edited Jun 29 '15 at 13:15
joslìber ♦
32.2k 11 39 80

answered Jun 29 '15 at 11:34
Priyankara
382 7 19

Is there a way to push the result of cmd to a file? I am curling a website and I want it to go to a file. – PolarisUser Oct 9 '15 at 16:59

This is by far the simplest and powerful solution. @PolarisUser you can use the generic linux command: `<command> > outputfile.txt` – user2820579 Oct 16 '15 at 16:44

it's also deprecated. use subprocess – Corey Goldberg Dec 9 '15 at 18:13

I tend to use subprocess together with shlex (to handle escaping of quoted strings):

```
>>> import subprocess, shlex
>>> command = 'ls -l "/your/path/with spaces/"'
>>> call_params = shlex.split(command)
>>> print call_params
["ls", "-l", "/your/path/with spaces/"]
>>> subprocess.call(call_params)
```

5,002 3 28 44

Shameless plug, I wrote a library for this :P https://github.com/houqp/shell.py

It's basically a wrapper for popen and shlex for now. It also supports piping commands so you can chain commands easier in Python. So you can do things like:

```
ex('echo hello shell.py') | "awk '{print $2}'"
```

answered May 1 '14 at 20:49
houqp
310 6 8

you can use Popen, then you can check procedure's status

```
from subprocess import Popen

proc = Popen(['ls', '-l'])
if proc.poll() is None:
    proc.kill()
```

Check this out subprocess.Popen

answered Jul 16 '12 at 15:16
admire
171 1 5

Here are my 2 cents: In my view this is best practice when dealing with external commands...

This is return values from execute method...

```
pass, stdout, stderr = execute(["ls","-la"],"/home/user/desktop")
```

This is execute method...

```
def execute(cmdArray,workingDir):

    stdout = ''
    stderr = ''

    try:
        try:
            process = subprocess.Popen(cmdArray,cwd=workingDir,
stdout=subprocess.PIPE, stderr=subprocess.PIPE, bufsize=1)
        except OSError:
            return [False, '', 'ERROR : command(' + ' '.join(cmdArray) + ') could
not get executed!']

        for line in iter(process.stdout.readline, b''):

            try:
                echoLine = line.decode("utf-8")
            except:
                echoLine = str(line)

            stdout += echoLine

        for line in iter(process.stderr.readline, b''):

            try:
                echoLine = line.decode("utf-8")
            except:
                echoLine = str(line)

            stderr += echoLine

    except (KeyboardInterrupt,SystemExit) as err:
        return [False,'',str(err)]

    process.stdout.close()

    returnCode = process.wait()
    if returnCode != 0 or stderr != '':
        return [False, stdout, stderr]
    else:
```

urosjarc
450 6 15

Deadlock potential: use the `.communicate` method instead – ppperry Jul 7 '16 at 2:15

To fetch the network id from the openstack neutron:

```
#!/usr/bin/python
import os
netid= "nova net-list | awk '/ External / { print $2 }'"
temp=os.popen(netid).read()  /* here temp also contains new line (\n) */
networkId=temp.rstrip()
print(networkId)
```

Output of **nova net-list**

```
+--------------------------------------+------------+------+
| ID                                   | Label      | CIDR |
+--------------------------------------+------------+------+
| 431c9014-5b5d-4b51-a357-66020ffbb123 | test1      | None |
| 27a74fcd-37c0-4789-9414-9531b7e3f126 | External   | None |
| 5a2712e9-70dc-4b0e-9281-17e02f4684c9 | management | None |
| 7aa697f5-0e60-4c15-b4cc-9cb659698512 | Internal   | None |
+--------------------------------------+------------+------+
```

Output of **print(networkId)**

```
27a74fcd-37c0-4789-9414-9531b7e3f126
```

answered Jul 20 '16 at 9:50
IRSHAD
463 6 18

Just to add to the discussion, if you include using a Python console, you can call external commands from ipython. While in the ipython prompt, you can call call shell commands by prefixing '!'. You can also combine python code with shell, and assign the output of shell scripts to python variables.

For instance:

```
In [9]: mylist = !ls

In [10]: mylist
Out[10]:
['file1',
 'file2',
 'file3',]
```

answered Jun 19 '13 at 23:18
imagineerThat
1,076 3 16 30

There are a lot of different ways to run external commands in python, and all of them have their own plus sides and drawbacks.

My colleagues and me have been writing python sysadmin tools, so we need to run a lot of external commands, and sometimes you want them to block or run asynchronously, time-out, update every second...

There are also different ways of handling the return code and errors, and you might want to parse the output, and provide new input (in an expect kind of style) Or you will need to redirect stdin, stdout and stderr to run in a different tty (e.g., when using screen)

So you will probably have to write a lot of wrappers around the external command. So here is a python module which we have written which can handle almost anything you would want, and if not, it's very flexible so you can easily extend it:

https://github.com/hpcugent/vsc-base/blob/master/lib/vsc/utils/run.py

edited Nov 8 '13 at 14:28 answered Apr 17 '13 at 14:10
Jens Timmerman

Questions Jobs Documentation BETA Tags Users Log In Sign Up

1 2 next

protected by Martijn Pieters ♦ Apr 16 '13 at 20:23

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?