Running shell command from Python and capturing the output

I want to write a function that will execute a shell command and return its output **as a string**, no matter, is it an error or success message. I just want to get the same result that I would have gotten with the command line.

What would be a code example that would do such a thing?

For example:

```
def run_command(cmd):
    # ??????

print run_command('mysqladmin create test -uroot -pmysqladmin12')
# Should output something like:
# mysqladmin: CREATE DATABASE failed; error: 'Can't create database 'test'; database exists'
```

python shell subprocess

edited May 24 '15 at 1:11
Peter Mortensen
11.8k ● 17 ● 80 ● 107

asked Jan 21 '11 at 14:55
Silver Light
17.5k ● 26 ● 97 ● 144

2 related: stackoverflow.com/questions/2924310/… – jfs Jan 24 '11 at 9:22

13 Answers

The answer to this question depends on the version of Python you're using. The simplest approach is to use the `subprocess.check_output` function:

```
>>> subprocess.check_output(['ls', '-l'])
b'total 0\n-rw-r--r--  1 memyself  staff  0 Mar 14 11:04 files\n'
```

`check_output` runs a single program that takes only arguments as input.[1] It returns the result exactly as printed to `stdout`. If you need to write input to `stdin`, skip ahead to the `run` or `Popen` sections. If you want to execute complex shell commands, see the note on `shell=True` at the end of this answer.

The `check_output` function works on almost all versions of Python still in wide use (2.7+).[2] But for more recent versions, it is no longer the recommended approach.

Modern versions of Python (3.5 or higher): `run`

If you're using **Python 3.5** or higher, and **do not need backwards compatibility**, the new `run` function is recommended. It provides a very general, high-level API for the `subprocess` module. To capture the output of a program, pass the `subprocess.PIPE` flag to the `stdout` keyword argument. Then access the `stdout` attribute of the returned `CompletedProcess` object:

```
>>> import subprocess
>>> result = subprocess.run(['ls', '-l'], stdout=subprocess.PIPE)
>>> result.stdout
b'total 0\n-rw-r--r--  1 memyself  staff  0 Mar 14 11:04 files\n'
```

The return value is a `bytes` object, so if you want a proper string, you'll need to `decode` it. Assuming the called process returns a UTF-8-encoded string:

```
>>> result.stdout.decode('utf-8')
'total 0\n-rw-r--r--  1 memyself  staff  0 Mar 14 11:04 files\n'
```

This can all be compressed to a one-liner:

```
>>> subprocess.run(['ls', '-l'], stdout=subprocess.PIPE).stdout.decode('utf-8')
'total 0\n-rw-r--r--  1 memyself  staff  0 Mar 14 11:04 files\n'
```

If you want to pass input to the process's `stdin`, pass a `bytes` object to the `input` keyword argument:

```
>>> cmd = ['awk', 'length($0) > 5']
>>> input = 'foo\nfoofoo\n'.encode('utf-8')
>>> result = subprocess.run(cmd, stdout=subprocess.PIPE, input=input)
>>> result.stdout.decode('utf-8')
'foofoo\n'
```

You can capture errors by passing `stderr=subprocess.PIPE` (capture to `result.stderr`) or `stderr=subprocess.STDOUT` (capture to `result.stdout` along with regular output). When security is not a concern, you can also run more complex shell commands by passing `shell=True` as described in the notes below.

This adds just a bit of complexity, compared to the old way of doing things. But I think it's worth the payoff: now you can do almost anything you need to do with the `run` function alone.

Older versions of Python (2.7-3.4): `check_output`

If you are using an older version of Python, or need modest backwards compatibility, you can probably use the `check_output` function as briefly described above. It has been available since Python 2.7.

```
subprocess.check_output(*popenargs, **kwargs)
```

It takes takes the same arguments as `Popen` (see below), and returns a string containing the program's output. The beginning of this answer has a more detailed usage example.

You can pass `stderr=subprocess.STDOUT` to ensure that error messages are included in the returned output -- but don't pass `stderr=subprocess.PIPE` to `check_output`. It can cause deadlocks. When security is not a concern, you can also run more complex shell commands by passing `shell=True` as described in the notes below.

If you need to pipe from `stderr` or pass input to the process, `check_output` won't be up to the task. See the `Popen` examples below in that case.

Complex applications & legacy versions of Python (2.6 and below): `Popen`

If you need deep backwards compatibility, or if you need more sophisticated functionality than `check_output` provides, you'll have to work directly with `Popen` objects, which encapsulate the low-level API for subprocesses.

The `Popen` constructor accepts either **a single command** without arguments, or **a list** containing a command as its first item, followed by any number of arguments, each as a separate item in the list. `shlex.split` can help parse strings into appropriately formatted lists. `Popen` objects also accept a host of different arguments for process IO management and low-level configuration.

To send input and capture output, `communicate` is almost always the preferred method. As in:

```
output = subprocess.Popen(["mycmd", "myarg"],
                          stdout=subprocess.PIPE).communicate()[0]
```

Or

```
>>> import subprocess
>>> p = subprocess.Popen(['ls', '-a'], stdout=subprocess.PIPE,
...                                    stderr=subprocess.PIPE)
>>> out, err = p.communicate()
>>> print out
.
..
foo
```

If you set `stdin=PIPE`, `communicate` also allows you to pass data to the process via `stdin`:

```
>>> cmd = ['awk', 'length($0) > 5']
>>> p = subprocess.Popen(cmd, stdout=subprocess.PIPE,
...                           stderr=subprocess.PIPE,
...                           stdin=subprocess.PIPE)
>>> out, err = p.communicate('foo\nfoofoo\n')
>>> print out
foofoo
```

Note Aaron Hall's answer, which indicates that on some systems, you may need to set `stdout`, `stderr`, and `stdin` all to `PIPE` (or `DEVNULL`) to get `communicate` to work at all.

In some rare cases, you may need complex, real-time output capturing. Vartec's answer suggests a way forward, but methods other than `communicate` are prone to deadlocks if not used carefully.

As with all the above functions, when security is not a concern, you can run more complex shell commands by passing `shell=True`.

Notes

1. Running shell commands: the `shell=True` argument

Normally, each call to `run`, `check_output`, or the `Popen` constructor executes a *single program*. That means no fancy bash-style pipes. If you want to run complex shell commands, you can pass `shell=True`, which all three functions support.

However, doing so raises security concerns. If you're doing anything more than light scripting, you might be better off calling each process separately, and passing the output from each as an input to the next, via

```
run(cmd, [stdout=etc...], input=other_output)
```

Or

```
Popen(cmd, [stdout=etc...]).communicate(other_output)
```

The temptation to directly connect pipes is strong; resist it. Otherwise, you'll likely see deadlocks or have to do hacky things like this.

2. Unicode considerations

`check_output` returns a string in Python 2, but a `bytes` object in Python 3. It's worth taking a moment to learn about unicode if you haven't already.

edited May 23 '17 at 12:03
Community ♦
1 ● 1

answered Jan 21 '11 at 15:27
senderle
78.3k ● 18 ● 143 ● 178

2 Yes, I saw this one, but I use 2.6 (my mistake not to mention python version) – Silver Light Jan 21 '11 at 15:49

3 Both with `check_output()` and `communicate()` you have to wait until the process is done, with `poll()` you're getting output as it comes. Really depends what you need. – vartec Apr 5 '12 at 9:44

1 This answer worked for me where others didn't. – Cody Brown Mar 18 '13 at 19:54

1 Not sure if this only applies to later versions of Python, but the variable `out` was of type `<class 'bytes'>` for me. In order to get the output as a string I had to decode it before printing like so:

`out.decode("utf-8")` – PolyMesh Oct 31 '13 at 19:42

1 @par That doesn't work for you when you pass `shell=True`? It works for me. You don't need `shlex.split` when you pass `shell=True`. `shlex.split` is for non-shell commands. I think I'm going to take that bit out because this is muddying the waters. – senderle Apr 10 '17 at 12:00

This is way easier, but only works on Unix (including Cygwin).

```
import commands
print commands.getstatusoutput('wc -l file')
```

it returns a tuple with the (return_value, output)

edited Jul 11 '16 at 23:16
rogerdpack
28.5k ● 13 ● 114 ● 215

answered Feb 13 '12 at 19:41
byte_array
2,120 ● 1 ● 10 ● 8

2 simple but effective – rikAtee May 20 '12 at 21:54

25 Deprecated now, but very useful for old python versions without subprocess.check_output – static_rtti Jun 13 '12 at 8:20

17 Note that this is Unix-specific. It will for example fail on Windows. – Zitrax Jan 21 '13 at 9:50

2 +1 I have to work on ancient version of python 2.4 and this was VERY helpful – javadba Mar 14 '14 at 22:14

1 nicer than others. deprecated in 3.x which no one here uses – Erik Aronesty May 12 '15 at 14:06

Something like that:

```
def runProcess(exe):
    p = subprocess.Popen(exe, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    while(True):
      retcode = p.poll() #returns None while subprocess is running
      line = p.stdout.readline()
      yield line
      if(retcode is not None):
        break
```

Note, that I'm redirecting stderr to stdout, it might not be exactly what you want, but I want error messages also.

This function **yields line by line as they come** (normally you'd have to wait for subprocess to finish to get the output as a whole).

For your case the usage would be:

```
for line in runProcess('mysqladmin create test -uroot -pmysqladmin12'.split()):
    print line,
```

edited Sep 17 '15 at 17:02
Anders
4,369 ● 5 ● 26 ● 45

answered Jan 21 '11 at 15:02
vartec
87.2k ● 25 ● 166 ● 214

1 Thank you for your help! But function goes into an infinite loop for me... – Silver Light Jan 21 '11 at 15:17

4 -1: it is an infinite loop the if `retcode` is `0`. The check should be `if retcode is not None`. You should not yield empty strings (even an empty line is at least one symbol '\n'): `if line: yield line`. Call `p.stdout.close()` at the end. – jfs Jan 24 '11 at 9:37

2 I tried the code with ls -l /dirname and it breaks after listing two files while there are much more files in the directory – Vasilis Sep 30 '13 at 20:01

1 @Vasilis: check similar answer – jfs Nov 13 '13 at 1:28

3 @fuenfundachtzig: `.readlines()` won't return until *all* output is read and therefore it breaks for large output that does not fit in memory. Also to avoid missing buffered data after the subprocess exited there should be an analog of `if retcode is not None: yield from p.stdout.readlines(); break` – jfs Dec 21 '13 at 5:15

Vartec's answer doesn't read all lines, so I made a version that did:

```
def run_command(command):
    p = subprocess.Popen(command,
                         stdout=subprocess.PIPE,
                         stderr=subprocess.STDOUT)
    return iter(p.stdout.readline, b'')
```

Usage is the same as the accepted answer:

```
command = 'mysqladmin create test -uroot -pmysqladmin12'.split()
for line in run_command(command):
    print(line)
```

edited May 23 '17 at 11:33
Community ♦
1 ● 1

answered Oct 30 '12 at 9:24
Max Ekman
659 ● 5 ● 5

6 you could use `return iter(p.stdout.readline, b'')` instead of the while loop – jfs Nov 22 '12 at 15:44

1 That is a pretty cool use of iter, didn't know that! I updated the code. – Max Ekman Nov 28 '12 at 21:53

I'm pretty sure stdout keeps all output, it's a stream object with a buffer. I use a very similar technique to deplete all remaining output after a Popen have completed, and in my case, using poll() and readline during the execution to capture output live also. – Max Ekman Nov 28 '12 at 21:55

I've removed my misleading comment. I can confirm, `p.stdout.readline()` may return the non-empty previously-buffered output even if the child process have exited already (`p.poll()` is not `None`). – jfs Sep 18 '14 at 3:12

This code doesn't work. See here stackoverflow.com/questions/24340877/… – thang May 3 '15 at 6:00

This is a **tricky** but **super simple** solution which works in many situations:

```
import os
os.system('sample_cmd > tmp')
print open('tmp', 'r').read()
```

A temporary file(here is tmp) is created with the output of the command and you can read from it your desired output.

Extra note from the comments: You can remove the tmp file in the case of one-time job. If you need to do this several times, there is no need to delete the tmp.

```
os.remove('tmp')
```

edited Oct 27 '17 at 7:19

answered Aug 21 '16 at 21:28
mehD
707 ● 3 ● 13 ● 26

2 Hacky but super simple + works anywhere .. can combine it with `mktemp` to make it work in threaded situations I guess – Prakash Rajagaopal Oct 18 '16 at 1:32

1 Maybe the fastest method, but better add `os.remove('tmp')` to make it "fileless". – XuMuK Jul 3 '17 at 16:11

@XuMuK You're right in the case of a one-time job. If it is a repetitive work maybe deleting is not necessary – mehD Jul 5 '17 at 15:18

In Python 3.5:

```
import subprocess

output = subprocess.run("ls -l", shell=True, stdout=subprocess.PIPE,
                        universal_newlines=True)
print(output.stdout)
```

edited Jul 29 '16 at 23:00

answered Jul 29 '16 at 20:03
cmlaverdiere
626 ● 7 ● 4

Modern Python solution (>= 3.1):

```
res = subprocess.check_output(lcmd, stderr=subprocess.STDOUT)
```

answered Apr 20 '14 at 20:30
zlr
516 ● 7 ● 16

5 As the accepted answer says, `check_output()` is available since Python 2.7. – jfs Apr 21 '14 at 17:13

Your Mileage May Vary, I attempted @senderle's spin on Vartec's solution in Windows on Python 2.6.5, but I was getting errors, and no other solutions worked. My error was: `WindowsError: [Error 6] The handle is invalid` .

I found that I had to assign PIPE to every handle to get it to return the output I expected - the following worked for me.

```
import subprocess

def run_command(cmd):
    """given shell command, returns communication tuple of stdout and stderr"""
    return subprocess.Popen(cmd,
                            stdout=subprocess.PIPE,
                            stderr=subprocess.PIPE,
                            stdin=subprocess.PIPE).communicate()
```

and call like this, (`[0]` gets the first element of the tuple, `stdout`):

```
run_command('tracert 11.1.0.1')[0]
```

After learning more, I believe I need these pipe arguments because I'm working on a custom system that uses different handles, so I had to directly control all the std's.

To stop console popups (with Windows), do this:

```
def run_command(cmd):
    """given shell command, returns communication tuple of stdout and stderr"""
    # instantiate a startupinfo obj:
    startupinfo = subprocess.STARTUPINFO()
    # set the use show window flag, might make conditional on being in Windows:
    startupinfo.dwFlags |= subprocess.STARTF_USESHOWWINDOW
    # pass as the startupinfo keyword argument:
    return subprocess.Popen(cmd,
                            stdout=subprocess.PIPE,
                            stderr=subprocess.PIPE,
                            stdin=subprocess.PIPE,
                            startupinfo=startupinfo).communicate()

run_command('tracert 11.1.0.1')
```

edited Aug 7 '15 at 15:21
user2141741

answered Feb 18 '14 at 23:32
Aaron Hall ♦
118k ● 33 ● 240 ● 207

1 Interesting -- this must be a Windows thing. I'll add a note pointing to this in case people are getting similar errors. – senderle May 1 '14 at 14:04

use `DEVNULL` instead of `subprocess.PIPE` if you don't write/read from a pipe otherwise you may hang the child process. – jfs Sep 9 '14 at 10:57

Sounds like a good tip, @J.F.Sebastian – Aaron Hall ♦ Sep 18 '14 at 1:45

I had a slightly different flavor of the same problem with the following requirements:

1. Capture and return STDOUT messages as they accumulate in the STDOUT buffer (i.e. in realtime).
   - *@vartec solved this Pythonically with his use of generators and the 'yield' keyword above*
2. Print all STDOUT lines (*even if process exits before STDOUT buffer can be fully read*)
3. Don't waste CPU cycles polling the process at high-frequency
4. Check the return code of the subprocess
5. Print STDERR (separate from STDOUT) if we get a non-zero error return code.

I've combined and tweaked previous answers to come up with the following:

```
import subprocess
from time import sleep

def run_command(command):
    p = subprocess.Popen(command,
                         stdout=subprocess.PIPE,
                         stderr=subprocess.PIPE,
                         shell=True)
    # Read stdout from subprocess until the buffer is empty !
    for line in iter(p.stdout.readline, b''):
        if line: # Don't print blank lines
            yield line
    # This ensures the process has completed, AND sets the 'returncode' attr
    while p.poll() is None:
        sleep(.1) #Don't waste CPU-cycles
    # Empty STDERR buffer
    err = p.stderr.read()
    if p.returncode != 0:
       # The run_command() function is responsible for logging STDERR
       print("Error: " + str(err))
```

This code would be executed the same as previous answers:

```
for line in run_command(cmd):
    print(line)
```

edited Sep 18 '17 at 3:31
Community ♦
1 ● 1

answered Oct 19 '16 at 18:35
The Aelfinn
2,017 ● 16 ● 22

Do you mind explaining how the addition of sleep(.1) won't waste CPU cycles? – Moataz Elmasry Aug 2 '17 at 9:41

1 If we continued to call `p.poll()` without any sleep in between calls, we would waste CPU cycles by calling this function millions of times. Instead, we "throttle" our loop by telling the OS that we don't need to be bothered for the next 1/10th second, so it can carry out other tasks. (It's possible that p.poll() sleeps too, making our sleep statement redundant). – The Aelfinn Aug 2 '17 at 11:04

You can use following commands to run any shell command. I have used them on ubuntu.

```
import os
os.popen('your command here').read()
```

edited Jul 3 '17 at 19:04

answered Apr 4 '17 at 19:08
Muhammad Hassan
4,371 ● 2 ● 12 ● 29

Deprecated since version 2.6 – docs.python.org/2/library/os.html#os.popen – Filippo Vitale May 26 '17 at 13:28

1 @FilippoVitale Thanks. I did not know that it is deprecated. – Muhammad Hassan May 26 '17 at 14:44

I had the same problem But figured out a very simple way of doing this follow this

```
import subprocess
Input = subprocess.getoutput("ls -l")
print(Input)
```

Hope it helps out

Note: This solution is python3 specific as `subprocess.getoutput()` don't work in python2

edited May 18 '17 at 12:29

answered Nov 12 '16 at 19:44
itz-azhar
342 ● 4 ● 11

How does this solve the OP's problem? Please elaborate. – RamenChef Nov 12 '16 at 20:21

1 It returns the output of command as string, as simple as that – itz-azhar Dec 4 '16 at 7:55

3 Doesn't work on Python 2 – Allan Deamon Jan 15 '17 at 17:45

1 Of course, print is a statement on Python 2. You should be able to figure out this is a Python 3 answer. – Dev Jan 25 '17 at 21:07

@Dev print(s) is valid python 2. subprocess.getoutput is not. – user48956 Apr 27 '17 at 17:46

If you need to run a shell command on multiple files, this did the trick for me.

```
import os
import subprocess

# Define a function for running commands and capturing stdout line by line
# (Modified from Vartec's solution because it wasn't printing all lines)
def runProcess(exe):
    p = subprocess.Popen(exe, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    return iter(p.stdout.readline, b'')

# Get all filenames in working directory
for filename in os.listdir('./'):
    # This command will be run on each file
    cmd = 'nm ' + filename

    # Run the command and capture the output line by line.
    for line in runProcess(cmd.split()):
        # Eliminate leading and trailing whitespace
        line.strip()
        # Split the output
        output = line.split()

        # Filter the output and print relevant lines
        if len(output) > 2:
            if ((output[2] == 'set_program_name')):
                print filename
                print line
```

Edit: Just saw Max Persson's solution with J.F. Sebastian's suggestion. Went ahead and incorporated that.

answered Apr 1 '15 at 15:54
Ethan Strider
2,875 ● 2 ● 11 ● 23

eg, execute('ls -ahl') differentiated three/four possible returns and OS platforms:

1. no output, but run successfully
2. output empty line, run successfully
3. run failed
4. output something, run successfully

function below

```
def execute(cmd, output=True, DEBUG_MODE=False):
"""Executes a bash command.
(cmd, output=True)
output: whether print shell output to screen, only affects screen display, does not affect returned values
return: ...regardless of output=True/False...
        returns shell output as a list with each elment is a line of string (whitespace stripped both sides) from output
        could be
        [], ie, len()=0 --> no output;
        [''] --> output empty line;
        None --> error occured, see below

        if error ocurs, returns None (ie, is None), print out the error message to screen
"""
if not DEBUG_MODE:
    print "Command: " + cmd

    # https://stackoverflow.com/a/40139101/2292993
    def _execute_cmd(cmd):
        if os.name == 'nt' or platform.system() == 'Windows':
            # set stdin, out, err all to PIPE to get results (other than None) after run the Popen() instance
            p = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True)
        else:
            # Use bash; the default is sh
            p = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True, executable="/bin/bash")

        # the Popen() instance starts running once instantiated (??)
        # additionally, communicate(), or poll() and wait process to terminate
        # communicate() accepts optional input as stdin to the pipe (requires setting stdin=subprocess.PIPE above), return out, err as tuple
        # if communicate(), the results are buffered in memory

        # Read stdout from subprocess until the buffer is empty !
        # if error occurs, the stdout is '', which means the below loop is essentially skipped
        # A prefix of 'b' or 'B' is ignored in Python 2;
        # it indicates that the literal should become a bytes literal in Python 3
        # (e.g. when code is automatically converted with 2to3).
        # return iter(p.stdout.readline, b'')
        for line in iter(p.stdout.readline, b''):
            # # Windows has \r\n, Unix has \n, Old mac has \r
            # if line not in ['','\n','\r','\r\n']: # Don't print blank lines
                yield line
        while p.poll() is None:
            sleep(.1) #Don't waste CPU-cycles
```