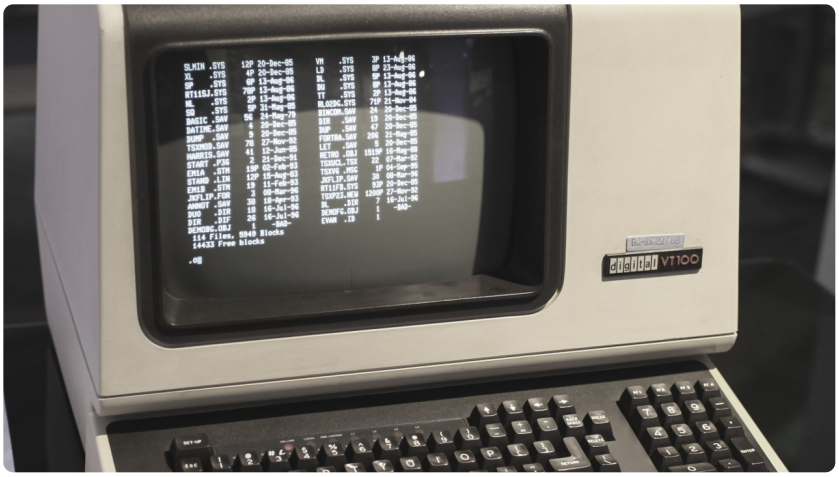

How to Execute Shell Commands with Python

22 Apr 2019

Python is a wonderful language for scripting and automating workflows and it is packed with useful tools out of the box with the Python Standard Library. A common thing to do, especially for a sysadmin, is to execute shell commands. But what usually will end up in a bash or batch file, can be also done in Python. You'll learn here how to do just that with the os and subprocess modules.

Using the `os` Module

The first and the most straight forward approach to run a shell command is by using os.system():

```
import os
os.system('ls -l')
```

If you save this as a script and run it, you will see the output in the command line. The problem with this approach is in its inflexibility since you can't even get the

resulting output as a variable. You can read more about this function in the documentation.

Note, that if you run this function in Jupyter notebook, you won't have an output inline. Instead you the inline output will be the return code of the executed programm (`0` for successful and `-1` for unsuccessful). You will find the output in the command line where you have started Jupyter notebook.

Next, the os.popen() command opens a pipe from or to the command line. This means that we can access the stream within Python. This is useful since you can now get the output as a variable:

```
import os
stream = os.popen('echo Returned output')
output = stream.read()
output
```

```
'Returned output\n'
```

When you use the `.read()` function, you will get the whole output as one string. You can also use the `.readlines()` function, which splits each line (including a trailing `\n`). Note, that you can run them only once. It is also possible to write to the stream by using the `mode='w'` argument. To delve deeper into this function, have a look at the documentation.

In this example and in the following examples, you will see that you always have trailing line breaks in the output. To remove them (including blank spaces and tabs in the beginning and end) you can use the `.strip()` function like with `output.strip()` . To remove those characters only in the beginning use `.lstrip()` and for the end `.rstrip()` .

Using the `subprocess` Module

The final approach is also the most versatile approach and the recommended module to run external commands in Python:

> The subprocess module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using this function. See the Replacing Older Functions with the subprocess Module section in the subprocess documentation for some helpful recipes. (Source)

The main function you want to keep in mind if you use Python >= 3.5 is subprocess.run(), but before we get there let's go through the functionality of the `subprocess` module. The subprocess.Popen() class is responsible for the creation and management of the executed process. In contrast to the previous functions, this class executes only a single command with arguments as a list. This means that you won't be able to pipe commands:

```
import subprocess
process = subprocess.Popen(['echo', 'More output'],
                     stdout=subprocess.PIPE,
                     stderr=subprocess.PIPE)
stdout, stderr = process.communicate()
stdout, stderr
```

```
(b'More output\n', b'')
```

You'll notice that we set `stdout` and `stderr` to subprocess.PIPE. This is a special value that indicates to `subprocess.Popen` that a pipe should be opened that you can then read with the `.communicate()` function. It is also possible to use a file object as with:

```
with open('test.txt', 'w') as f:
    process = subprocess.Popen(['ls', '-l'], stdout=f)
```

Another thing that you'll notice is that the output is of type bytes. You can solve that by typing

`stdout.decode('utf-8')` or by adding `universal_newlines=True` when calling `subprocess.Popen`.

When you run `.communicate()`, it will wait until the process is complete. However if you have a long program that you want to run and you want to continuously check the status in realtime while doing something else, you can do this like here:

```
process = subprocess.Popen(['ping', '-c 4', 'python.org'],
                           stdout=subprocess.PIPE,
                           universal_newlines=True)

while True:
    output = process.stdout.readline()
    print(output.strip())
    # Do something else
    return_code = process.poll()
    if return_code is not None:
        print('RETURN CODE', return_code)
        # Process has finished, read rest of the output
        for output in process.stdout.readlines():
            print(output.strip())
        break
```

```
PING python.org (45.55.99.72) 56(84) bytes of data.
64 bytes from 45.55.99.72 (45.55.99.72): icmp_seq=1 ttl=51 time=117 ms
64 bytes from 45.55.99.72 (45.55.99.72): icmp_seq=2 ttl=51 time=118 ms
64 bytes from 45.55.99.72 (45.55.99.72): icmp_seq=3 ttl=51 time=117 ms
64 bytes from 45.55.99.72 (45.55.99.72): icmp_seq=4 ttl=51 time=118 ms

--- python.org ping statistics ---
RETURN CODE 0
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 117.215/117.874/118.358/0.461 ms
```

You can use the `.poll()` function to check the return code of the process. It will return `None` while the process is still running. To get the output, you can use `process.stdout.readline()` to read a single line. Conversely, when you use `process.stdout.readlines()`, it reads all lines and it also waits for the process to finish if it has not finished yet. For more information on the functionionality of `subprocess.Popen`, have a look at the documentation.

Also note, that you won't need quotations for arguments with spaces in between like `'\"More output\"'`. If you

are unsure how to tokenize the arguments from the command, you can use the shlex.split() function:

```
import shlex
shlex.split("/bin/prog -i data.txt -o \"more data.txt\"")
```

```
['/bin/prog', '-i', 'data.txt', '-o', 'more data.txt']
```

You have also the subprocess.call() function to your disposal which works like the `Popen` class, but it waits until the command completes and gives you the return code as in `return_code = subprocess.call(['echo', 'Even more output'])`. The recommended way however is to use subprocess.run() which works since Python 3.5. It has been added as a simplification of `subprocess.Popen`. The function will return a subprocess.CompletedProcess object:

```
process = subprocess.run(['echo', 'Even more output'],
                         stdout=subprocess.PIPE,
                         universal_newlines=True)
process
```

```
CompletedProcess(args=['echo', 'Even more output'], returncode=0, stdo
```

You can now find the resulting output in this variable:

```
process.stdout
```

```
'Even more output\n'
```

Similar to `subprocess.call()` and the previous `.communicate()` function, it will wait untill the process is completed. Finally, here is a more advanced example on how to access a server with ssh and the `subprocess` module:

```
import subprocess

ssh = subprocess.Popen(["ssh", "-i .ssh/id_rsa", "user@host"],
                       stdin =subprocess.PIPE,
```

```
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE,
                        universal_newlines=True,
                        bufsize=0)

# Send ssh commands to stdin
ssh.stdin.write("uname -a\n")
ssh.stdin.write("uptime\n")
ssh.stdin.close()

# Fetch output
for line in ssh.stdout:
    print(line.strip())
```

Here you can see how to write input to the process. In this case you need to set the `bufsize=0` in order to have unbuffered output. After you are finished writing to the `stdin`, you need to close the connection.

Conclusion

You have seen now how to run external commands in Python. The most effective way is to use the `subprocess` module with all the functionality it offers. Most notably, you should consider using `subprocess.run`. For a short and quick script you might just want to use the `os.system()` or `os.popen()` functions. If you have any questions, feel free to leave them in the comments below. There are also other useful libraries that support shell commands in Python, like plumbum, sh, psutils and pexpect.

Further Reading

- 4 Techniques for Testing Python Command-Line (CLI) Apps
- Frequently Used Arguments - subprocess

Image from Wikimedia Commons

ALSO ON JANAKIEV

10 months ago • 5 comments
How to Create Your Data Science Blog ...

2 years ago • 2 comments
Compare Countries and Cities with ...

What do you think?
83 Responses

Upvote Funny Love

Surprised Angry Sad

Comments Community Privacy Policy Login

1

Sort by Best

Related Posts

How to Manage Apache Airflow with Systemd on Debian or Ubuntu
20 Dec 2019

Local Testing Server with Python
30 Oct 2018

Running a Python Script in the Background
19 Oct 2018

Using the Blender Interactive Console from the Command-Line
24 Mar 2018