Executing Shell Commands with Python

Sajjad Heydari

Introduction

Repetitive tasks are ripe for automation. It is common for developers and system administrators to automate routine tasks like health checks and file backups with shell scripts. However, as those tasks become more complex, shell scripts may become harder to maintain.

Fortunately, we can use Python instead of shell scripts for automation. Python provides methods to run shell commands, giving us the same functionality of those shells scripts. Learning how to run shell commands in Python opens the door for us to automate computer tasks in a structured and scalable way.

In this article, we will look at the various ways to execute shell commands in Python, and the ideal situation to use each method.

Using os.system to Run a Command

Python allows us to immediately execute a shell command that's stored in a string using the `os.system()` function.

Let's start by creating a new Python file called `echo_adelle.py` and enter the following:

```
import os

os.system("echo Hello from the other side!")
```

The first thing we do in our Python file is import the `os` module, which contains the `system` function that can execute shell commands. The next line does exactly that, runs the `echo` command in our shell through Python.

In your Terminal, run this file with using the following command, and you should see the corresponding output:

```
$ python3 echo_adelle.py
Hello from the other side!
```

As the `echo` commands prints to our `stdout`, `os.system()` also displays the output on our `stdout` stream. While not visible in the console, the `os.system()` command returns the exit code of the shell command. An exit code of 0 means it ran without any problems and any other number means an error.

Let's create a new file called `cd_return_codes.py` and type the following:

```
import os

home_dir = os.system("cd ~")
print("`cd ~` ran with exit code %d" % home_dir)
unknown_dir = os.system("cd doesnotexist")
print("`cd doesnotexis` ran with exit code %d" % unknown_dir)
```

In this script, we create two variables that store the result of executing commands that change the directory to the home folder, and to a folder that does not exist. Running this file, we will see:

```
$ python3 cd_return_codes.py
`cd ~` ran with exit code 0
sh: line 0: cd: doesnotexist: No such file or directory
`cd doesnotexist` ran with exit code 256
```

The first command, which changes the directory to the home directory, executes successfully. Therefore, `os.system()` returns its exit code, zero, which is stored in `home_dir`. On the other hand, `unknown_dir` stores the exit code of the failed bash command to change the directory to a folder that does not exist.

The `os.system()` function executes a command, prints any output of the command to the console, and returns the exit code of the command. If we would like more fine grained control of a shell command's input and output in Python, we should use the `subprocess` module.

Running a Command with subprocess

The subprocess module is Python's recommended way to executing shell commands. It gives us the flexibility to suppress the output of shell commands or chain inputs and outputs of various commands together, while still providing a similar experience to `os.system()` for basic use cases.

In a new filed called `list_subprocess.py`, write the following code:


```
import subprocess

list_files = subprocess.run(["ls", "-l"])
print("The exit code was: %d" % list_files.returncode)
```

In the first line, we import the `subprocess` module, which is part of the Python standard library. We then use the `subprocess.run()` function to execute the command. Like `os.system()`, the `subprocess.run()` command returns the exit code of what was executed.

Unlike `os.system()`, note how `subprocess.run()` requires a list of strings as input instead of a single string. The first item of the list is the name of the command. The remaining items of the list are the flags and the arguments of the command.

Note: As a rule of thumb, you need to separate the arguments based on space, for example `ls -alh` would be `["ls", "-alh"]`, while `ls -a -l -h`, would be `["ls", "-a", -"l", "-h"]`. As another example, `echo hello world` would be `["echo", "hello", "world"]`, whereas `echo "hello world"` or `echo hello\ world` would be `["echo", "hello world"]`.

Run this file and your console's output would be similar to:

```
$ python3 list_subprocess.py
total 80
-rw-r--r--@ 1 stackabuse  staff    216 Dec  6 10:29 cd_return_codes.py
-rw-r--r--@ 1 stackabuse  staff     56 Dec  6 10:11 echo_adelle.py
-rw-r--r--@ 1 stackabuse  staff    116 Dec  6 11:20 list_subprocess.py
The exit code was: 0
```

Now let's try to use one of the more advanced features of `subprocess.run()`, namely ignore output to `stdout`. In the same `list_subprocess.py` file, change:

```
list_files = subprocess.run(["ls", "-l"])
```

To this:

```
list_files = subprocess.run(["ls", "-l"], stdout=subprocess.DEVNULL)
```


The standard output of the command now pipes to the special `/dev/null` device, which means the output would not appear on our consoles. Execute the file in your shell to see the following output:

```
$ python3 list_subprocess.py
The exit code was: 0
```

What if we wanted to provide input to a command? The `subprocess.run()` facilitates this by its `input` argument. Create a new file called `cat_subprocess.py`, typing the following:

```
import subprocess

useless_cat_call = subprocess.run(["cat"], stdout=subprocess.PIPE, text=True, input="He
print(useless_cat_call.stdout)  # Hello from the other side
```

We use `subprocess.run()` with quite a few commands, let's go through them:

- `stdout=subprocess.PIPE` tells Python to redirect the output of the command to an object so it can be manually read later
- `text=True` returns `stdout` and `stderr` as strings. The default return type is bytes.
- `input="Hello from the other side"` tells Python to add the string as input to the `cat` command.

Running this file produces the following output:

```
Hello from the other side
```

We can also raise an `Exception` without manually checking the return value. In a new file, `false_subprocess.py`, add the code below:

```
import subprocess

failed_command = subprocess.run(["false"], check=True)
print("The exit code was: %d" % failed_command.returncode)
```

In your Terminal, run this file. You will see the following error:

```
$ python3 false_subprocess.py
Traceback (most recent call last):
  File "false_subprocess.py", line 4, in <module>
    failed_command = subprocess.run(["false"], check=True)
  File "/usr/local/python/3.7.5/Frameworks/Python.framework/Versions/3.7/lib/python3.7/
    output=stdout, stderr=stderr)
subprocess.CalledProcessError: Command '['false']' returned non-zero exit status 1.
```

By using `check=True`, we tell Python to raise any exceptions if an error is encountered. Since we did encounter an error, the `print` statement on the final line was not executed.

The `subprocess.run()` function gives us immense flexibility that `os.system()` doesn't when executing shell commands. This function is a simplified abstraction of the `subprocess.Popen` class, which provides additional functionality we can explore.

Running a Command with Popen

The `subprocess.Popen` class exposes more options to the developer when interacting with the shell. However, we need to be more explicit about retrieving results and errors.

By default, `subprocess.Popen` does not stop processing of a Python program if its command has not finished executing. In a new file called `list_popen.py`, type the following:

```
import subprocess

list_dir = subprocess.Popen(["ls", "-l"])
list_dir.wait()
```

This code is equivalent to that of `list_subprocess.py`. It runs a command using `subprocess.Popen`, and waits for it to complete before executing the rest of the Python script.

Let's say we do not want to wait for our shell command to complete execution so the program can work on other things. How would it know when the shell command has finished execution?

The `poll()` method returns the exit code if a command has finished running, or `None` if it's still executing. For example, if we wanted to check if `list_dir` was complete instead of wait for it, we would have the following line of code:

```
list_dir.poll()
```


To manage input and output with `subprocess.Popen`, we need to use the `communicate()` method.

In a new file called `cat_popen.py`, add the following code snippet:

```
import subprocess

useless_cat_call = subprocess.Popen(["cat"], stdin=subprocess.PIPE, stdout=subprocess.P
output, errors = useless_cat_call.communicate(input="Hello from the other side!")
useless_cat_call.wait()
print(output)
print(errors)
```

The `communicate()` method takes an `input` argument that's used to pass input to the shell command. The `communicate` method also returns both the `stdout` and `stderr` when they are

set.

Having seen the core ideas behind `subprocess.Popen`, we have now covered three ways to run shell commands in Python. Let's re-examine their characteristics so we'll know which method is best suited for a project's requirements.

Which one should I use?

If you need to run one or a few simple commands and do not mind if their output goes to the console, you can use the `os.system()` command. If you want to manage the input and output of a shell command, use `subprocess.run()`. If you want to run a command and continue doing other work while it's being executed, use `subprocess.Popen`.

Here is a table with some usability differences that you can also use to inform your decision:

	os.system	subprocess.run	subprocess.Popen
Requires parsed arguments	no	yes	yes
Waits for the command	yes	yes	no
Communicates with stdin and stdout	no	yes	yes
Returns	return value	object	object

Conclusion

Python allows you to execute shell commands, which you can use to start other programs or better manage shell scripts that you use for automation. Depending on our use case, we can use `os.system()`, `subprocess.run()` or `subprocess.Popen` to run bash commands.

Using these techniques, what external task would you run via Python?

#python #bash #shell

Last Updated: September 19th, 2021

Was this article helpful? ☆☆☆☆☆

You might also like...

- Array Loops in Bash
- Python's os and subprocess Popen Commands
- Pimp my Terminal - An Introduction to "Oh My Zsh"
- Handling Unix Signals in Python
- Lists vs Tuples in Python

Sajjad Heydari *Author*

Marcus Sanatan
Editor