Extracted from:

Intuitive Python

Productive Development for Projects that Last

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Intuitive Python

Productive Development for Projects that Last

David Muller

edited by Adaobi Obi Tulton

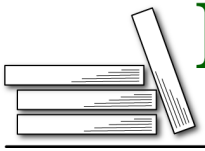
Intuitive Python

Productive Development for Projects that Last

David Muller

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Adaobi Obi Tulton
Copy Editor: Karen Galle
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-823-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2021

Calling Other Programs with subprocess

You can use the subprocess standard library module to invoke other command line programs available on your system and process their results in Python. You might find the subprocess useful if, for example, you want to call a program like git from inside Python to learn about the current state of your version controlled project. Indeed, some examples in this section will combine subprocess and git to do just that.

Note, if you don't have git installed on your machine or are not currently in a directory with a git repository, some of the following examples may not successfully execute on your computer. You can download git if you like,[9] but it's also OK to just follow along with the examples here so you get a sense of the abilities of the subprocess module (you don't need any special git knowledge).

You can use subprocess with git to retrieve the name of the current git branch you are on:

```
>>> import subprocess
>>> command = ["git", "branch", "--show-current"]
>>> result = subprocess.run(command, capture_output=True)
>>> result.returncode
0
>>> result.stdout
b'main\n'
>>> result.stdout.decode("utf-8").strip()
'main'
>>>
```

In the preceding example, you call subprocess.run with two arguments. The first argument (command) is a list of strings specifying the command line program you want to run. In general, you can think of Python as telling the operating system to execute as if there were spaces separating them. In other words ["git", "branch", "--show-current"] is roughly translated to git branch --show-current and executed by the operating system. (Python automatically handles any necessary escaping and quoting for you. This escaping and quoting can be helpful if, for example, one of the arguments in your list is a file name with a space in it.)

The second argument capture_output flag instructs Python to record stdout and stderr and make them available to you in the CompletedProcess object[10] returned by subprocess.run. In the preceding example, the result variable is bound to the

9. https://git-scm.com
10. https://docs.python.org/3/library/subprocess.html#subprocess.CompletedProcess

CompletedProcess object. result.returncode indicates that the git command you ran exited with a 0 code. Accessing result.stdout returns bytes with the output of the underlying git command we ran. Decoding the bytes as utf-8[11] and calling strip[12] to remove the trailing newline \n character leaves us with a Python string indicating my current branch name: 'main'.

Let subprocess.run Automatically Decode bytes for You

On Python 3.7 or higher, you can pass text=True to subprocess.run. subprocess.run will then automatically coerce the resulting bytes in stdout and stderr to strings and save you from having to call decode on the bytes yourself.

So far, we've only worked with an example where the underlying command we called worked. What happens if the underlying command failed and returned a non-0 returncode?

Handling Exceptional Cases with subprocess

Python can automatically raise an exception for you if the underlying command didn't exit with a returncode of 0. If you pass check=True to subprocess.run, it will raise an Exception if the underlying command fails:

```
>>> import subprocess
>>> subprocess.run(["git", "oops"], check=True)
git: 'oops' is not a git command. See 'git --help'.

The most similar command is
    notes
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.9/subprocess.py", line 528, in run
    raise CalledProcessError(retcode, process.args,
subprocess.CalledProcessError: Command '['git', 'oops']' returned
non-zero exit status 1.
>>>
```

git does not support a subcommand named oops, so when you try to execute git oops, git complains and returns a non-0 returncode. By including the check=True argument in your subprocess.run call, Python automatically raises a CalledProcessError exception for you indicating the failure. This CalledProcessError exception can be useful if you want your program to exit or otherwise fail if the underlying command you call doesn't work.

11. https://en.wikipedia.org/wiki/UTF-8
12. https://docs.python.org/3/library/stdtypes.html#str.strip

Timing Out Commands Run By subprocess

You can also instruct subprocess to automatically kill the underlying command if it has not completed after a certain amount of time using the timeout argument to subprocess.run:

```
>>> import subprocess
>>> subprocess.run(["sleep", "2"], timeout=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.9/subprocess.py", line 507, in run
    stdout, stderr = process.communicate(input, timeout=timeout)
  File "/usr/local/lib/python3.9/subprocess.py", line 1134, in communicate
    stdout, stderr = self._communicate(input, endtime, timeout)
  File "/usr/local/lib/python3.9/subprocess.py", line 2007, in _communicate
    self.wait(timeout=self._remaining_time(endtime))
  File "/usr/local/lib/python3.9/subprocess.py", line 1189, in wait
    return self._wait(timeout=timeout)
  File "/usr/local/lib/python3.9/subprocess.py", line 1911, in _wait
    raise TimeoutExpired(self.args, timeout)
subprocess.TimeoutExpired: Command '['sleep', '2']' timed out after
0.9996613000002981 seconds
>>>
```

By passing the timeout=1 argument to subprocess.run, you are instructing subprocess.run to raise a TimeoutExpired exception after approximately one second has passed and the underlying command hasn't completed.[13] Since the sleep 2 command just waits for two seconds, it should never complete in one second and, indeed, you see a TimeoutExpired exception raised in the preceding output. Notably—as you may have surmised from the output—the timeout operation is made on a best effort basis and may be above or below the timeout value you request. In this case, for example, it was actually just under one second before the TimeoutExpired exception was raised.

You've now seen how to call external programs, handle errors they might return, and kill them if they are taking too long. Next, you'll learn how you can invoke programs that might need to have data sent to them over stdin.

Passing Input to External Programs with subprocess

Sometimes, it's useful to pass input to command line programs via stdin—either because the underlying program requires it, or you have a significant amount of data that you don't want to load into RAM.

13. https://docs.python.org/3/library/subprocess.html#subprocess.TimeoutExpired

Use the input Argument to Pass bytes to stdin

For simple cases when you want to pass data to the stdin of a program, you can use the input argument to subprocess.run. For example, you can search a sequence of input bytes with grep:

```
>>> import subprocess
>>> to_grep = b"Alpha\nBeta\nGamma"
>>> command = ["grep", "eta"]
>>> result = subprocess.run(command, input=to_grep, capture_output=True)
>>> result.stdout
b'Beta\n'
>>>
```

In this example, we define an input sequence of bytes that we want to grep through: b"Alpha\nBeta\nGamma". Next, we define our grep command as grep eta—we are searching for lines in the input that contain eta. Using the input argument to subprocess.run, we pass our to_grep bytes to our grep eta command as stdin. grep responds that it found one matching line b'Beta'\n. You have successfully passed stdin to a child program using subprocess!

Notably, when you use the input argument to subprocess.run you need all the data you want to pass as stdin to be loaded into your application's RAM. What if the data you wanted to pass through stdin was large—large enough, for example, that you wouldn't want to store it in the RAM of your Python application?

Use the stdin Argument to Pass Data Stored in Files to stdin

It turns out that subprocess.run also supports an argument besides input for passing values to stdin. subprocess.run actually includes an argument named stdin, which can accept file objects like those produced by the built-in open command. That was a lot to unpack, but let's try with an example where we pass the contents of a file to grep:

subprocess_with_stdin.py

```
import subprocess

with open("example.txt", mode="w") as f:
    contents = "example\ntext\nfile"
    f.write(contents)

with open("example.txt", mode="r") as f:
    result = subprocess.run(
        ["grep", "l"], stdin=f, capture_output=True, check=True
    )

stdout_str = result.stdout.decode("utf-8").strip()
print(stdout_str)
```

In the preceding example, the file example.txt is created and has the strings example, text, and file written into it with each word on its own line. Then, example.txt is opened for reading (mode="r") and its file object[14] bound to f is passed as the value for subprocess.run's stdin argument. The command in subprocess.run is ["grep", "l"], which translates roughly to, "find lines in the input that include l in them." The result of the subprocess.run call is bound to result, and the captured stdout value is decoded from bytes into a string, stripped of its trailing newline, and printed.

If you run python3 subprocess_with_stdin.py, you should see output like this:

```
example
file
```

grep has found all the lines in our file which had l in them, and the output indicates this. Importantly, you were able to pass stdin to grep without loading the entire contents of example.txt into RAM, which might be problematic if you are working with large files. In this example, example.txt was a file of trivial size, but you can imagine working with files much larger than example.txt.

The stdout and stderr arguments to subprocess.run[15] also support being passed as file objects. You can use file objects in place of the pipes[16] and redirects[17] you may be more familiar with from traditional shells. As you just learned about in Creating Temporary Workspaces with tempfile, on page ?, you can even take advantage of NamedTempfile and TemporaryDirectory to use as short-lived workspaces for your endeavors with subprocess.

In this section, you learned how to dispatch commands to the underlying operating system. In the next section, you'll learn how to use the sqlite3 standard library module to gain access to another powerful tool: sqlite.

14. https://docs.python.org/3/glossary.html#term-file-object
15. https://docs.python.org/3/library/subprocess.html#subprocess.run
16. https://en.wikipedia.org/wiki/Pipeline_%28Unix%29
17. https://en.wikipedia.org/wiki/Redirection_(computing)