An Introduction to Python Subprocess: Basics and Examples

Explore our step-by-step guide to running external commands using Python's subprocess module, complete with examples.

Feb 2023 · 15 min read

CONTENTS

SHARE

You can use the Python subprocess module to create new processes, connect to their input and output, and retrieve their return codes and/or output of the process. In this article, we'll explore the basics of the subprocess module, including how to run external commands, redirect input and output, and handle errors. Whether you're a beginner or an experienced Python developer, this tutorial will provide you with the knowledge you need to use the subprocess module effectively in your projects.

What is Python Subprocess

The Python subprocess module is a tool that allows you to run other programs or commands from your Python code. It can be used to open new programs, send them data and get results back.

It's like giving commands to your computer using Python instead of typing them directly into the command prompt. This module makes it easy to automate tasks and integrate other programs with your Python code.

For example, you can use the subprocess module to run a shell command, like "ls" or "ping", and get the output of that command in your Python code. You can also use it to run other Python scripts or executables, like .exe files on Windows.

Additionally, the subprocess module can redirect the input and output of the process, meaning you can control what data is sent to the process and what data is received from it.

One of the most useful capabilities of the subprocess module is that it enables the user to handle the inputs, outputs, and even the errors that are generated by the child process from within the Python code.

This feature is considered to be one of the module's most powerful aspects. Because of this feature, it is now possible to make the process of calling subprocesses more powerful and versatile. For example, it is now possible to use the output of the subprocess as a variable throughout the rest of the Python script.

In this tutorial, we will learn how to use the subprocess module to run other programs from our Python code, how to send data to them, and how to get results back. It will be helpful for both beginners and experienced Python developers.

When to use Python Subprocess

Automating system tasks

The subprocess module can be used to automate various system tasks, such as running backups, starting and stopping services, and scheduling cron jobs. For example, you can use the subprocess module to run the "cp" command to create a backup of a file or the "service" command to start and stop services on Linux systems.

You can also use the subprocess module to schedule tasks to run at specific intervals using

You can also use the subprocess module to schedule tasks to run at specific intervals using cron or other scheduling tools.

Running command-line tools

The subprocess module can be used to run command-line tools such as grep, sed, and awk, and process their output in your Python code. For example, you can use the subprocess module to run the "grep" command to search for a specific pattern in a file and then process the output in your Python code. This can be useful for tasks such as log analysis, data processing, and text manipulation.

Running external executables

The subprocess module can run other executables, such as .exe files on Windows, and control their behavior. For example, you can use the subprocess module to run an executable that performs a specific task and then use the output of that executable in your own code. This can be useful for image processing, data analysis, and machine learning tasks.

Running scripts as background processes

You can use the subprocess module to run scripts as background processes so that they continue running after the main program exits. For example, you can use the subprocess module to run a script that performs a specific task and exit the main program without waiting for the script to complete. This can be useful for monitoring, logging, and data collection.

Running scripts with non-Python interpreter

The subprocess module can help you run scripts written in other languages like Perl, Ruby, and Bash. For example, you can use the subprocess module to run a Perl script that performs a specific task and then use the output of that script in your own code. This can be useful for data processing, text manipulation, and system administration.

Parallel processing

The subprocess module can be used to run multiple processes in parallel, which can be useful for tasks such as image processing, data analysis, and machine learning. For example, you can use the subprocess module to run multiple instances of the same script, each processing a different part of the data and then combine the results.

Python Subprocess Examples

python subprocess run

The `subprocess.run()` method is a convenient way to run a subprocess and wait for it to complete. It lets you choose the command to run and add options like arguments, environment variables, and input/output redirections. Once the subprocess is started, the `run()` method blocks until the subprocess completes and returns a CompletedProcess object, which contains the return code and output of the subprocess.

The `subprocess.run()` method takes several arguments, some of which are:

- `args`: The command to run and its arguments, passed as a list of strings.
- `capture_output`: When set to True, will capture the standard output and standard error.
- `text`: When set to True, will return the stdout and stderr as string, otherwise as bytes.
- `check`: a boolean value that indicates whether to check the return code of the subprocess, if check is true and the return code is non-zero, then subprocess `CalledProcessError` is raised.
- `timeout`: A value in seconds that specifies how long to wait for the subprocess to complete before timing out.
- `shell`: A boolean value that indicates whether to run the command in a shell. This means that the command is passed as a string, and shell-specific features, such as wildcard expansion and variable substitution, can be used.

The subprocess.run() method also returns a CompletedProcess object, which contains the following attributes:

- `args`: The command and arguments that were run.
- `returncode`: The return code of the subprocess.
- `stdout`: The standard output of the subprocess, as a bytes object.

- `stderr`: The standard error of the subprocess, as a bytes object.

Example 1: Running shell commands:

```
import subprocess

result = subprocess.run(["dir"], shell=True, capture_output=True, text=True)

print(result.stdout)
```

Explain code | Run code | POWERED BY datalab

Output:

```
Volume in drive C has no label.

Volume Serial Number is E414-A41C

 Directory of C:\Users\owner

01/25/2023  10:56 AM    <DIR>          .

01/25/2023  10:56 AM    <DIR>          ..

07/19/2021  01:19 PM    <DIR>          .anaconda

07/19/2021  01:19 PM    <DIR>          .astropy

07/19/2021  01:19 PM    <DIR>          .aws

09/12/2022  08:48 AM               496 .bash_history

03/27/2022  03:08 PM    <DIR>          .cache

09/26/2021  06:58 AM    <DIR>          .conda

09/26/2021  06:59 AM                25 .condarc

…
```

POWERED BY datalab POWERED BY datalab

Linux or mac users replace “dir” to “ls” and get rid of the `shell` argument.

Example 2: Running Python scripts:

You can also run a python script using the subprocess.run() method. Let’s start by creating a simple Python script in .py file

```
print(“This is the output from subprocess module”)
```

POWERED BY datalab POWERED BY datalab

Save this file as `my_python_file.py`

Now you can use subprocess module to run this file:

```
import subprocess

result = subprocess.run(["python", "my_python_file.py"], capture_output=True, tex

print(result.stdout)
```

Explain code | Run code | POWERED BY datalab

Output:

```
This is the output from subprocess module
```

Explain code | Run code | POWERED BY datalab

Example 3: Running Python code directly from a function:

For simple use-cases, you can directly pass a python command in the subprocess.run() function. Here is how:

```
result = subprocess.run(["C:/Users/owner/anaconda3/python", "-c", "print('Th is
```

```
print(result.stdout)
```

Explain code | Run code | POWERED BY datalab

Output:

```
This is directly from a subprocess.run() function
```

Explain code | Run code | POWERED BY datalab

In the `args` list, the first element `C:/Users/owner/anaconda3/python` is a path to executable Python (your path may be different). The second element, `-c` is a Python tag that allows the user to write Python code as text to the command line. The third element, `print(...)` is the Python command itself.

Example 4: Using the check argument

The check argument is an optional argument of the subprocess.run() function in the Python subprocess module. It is a boolean value that controls whether the function should check the return code of the command being run.

When check is set to True, the function will check the return code of the command and raise a CalledProcessError exception if the return code is non-zero. The exception will have the return code, stdout, stderr, and command as attributes.

When check is set to False (default), the function will not check the return code and will not raise an exception, even if the command fails.

```
import subprocess

result = subprocess.run(["python", "file_donot_exist.py"], capture_output=True, t

print(result.stdout)

print(result.stderr)
```

Explain code | Run code | POWERED BY datalab

Output:

```
---------------------------------------------------------------------------
CalledProcessError                        Traceback (most recent call last)
<ipython-input-81-503b60184db8> in <module>
      1 import subprocess
----> 2 result = subprocess.run(["python", "file_donot_exist.py"], capture_output
      3 print(result.stdout)
      4 print(result.stderr)
~\anaconda3\lib\subprocess.py in run(input, capture_output, timeout, check, *pope
    514         retcode = process.poll()
    515         if check and retcode:
--> 516             raise CalledProcessError(retcode, process.args,
    517                                      output=stdout, stderr=stderr)
    518     return CompletedProcess(process.args, retcode, stdout, stderr)
CalledProcessError: Command '['python', 'file_donot_exist.py']' returned non-zero
```

Explain code | Run code | POWERED BY datalab

Notice that the command failed because `file_donot_exist.py` does not exist. As opposed to when you set `check=True`, your process won't fail; instead, you will get the error message in `stdout`.

```
import subprocess

result = subprocess.run(["python", "my_python_file2.py"], capture_output=True, te

print(result.stdout)

print(result.stderr)
```

```
print(result.stderr)
```

Explain code | Run code | POWERED BY datalab

Output:

```
python: can't open file 'my_python_file2.py': [Errno 2] No such file or dir ry
```

Explain code | Run code | POWERED BY datalab

python subprocess Popen

`subprocess.Popen` is a lower-level interface to running subprocesses, while subprocess.run is a higher-level wrapper around Popen that is intended to be more convenient to use.

Popen allows you to start a new process and interact with its standard input, output, and error streams. It returns a handle to the running process that can be used to wait for the process to complete, check its return code, or terminate it.

run is a more convenient function that allows you to run a command and capture its output in a single call, without having to create a Popen object and manage the streams yourself. It also allows you to specify various options for running the command, such as whether to raise an exception if the command fails.

In general, you should use run if you just need to run a command and capture its output and Popen if you need more control over the process, such as interacting with its input and output streams.

The Popen class takes the same arguments as run(), including the args that specify the command to be run and other optional arguments such as stdin, stdout, stderr, shell, cwd, and env.

The Popen class has several methods that allow you to interact with the process, such as communicate(), poll(), wait(), terminate(), and kill().

```
import subprocess

p = subprocess.Popen(["python", "--help"], stdout=subprocess.PIPE, stderr=subproc

output, errors = p.communicate()

print(output)
```

Explain code | Run code | POWERED BY datalab

Output:

```
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...

Options and arguments (and corresponding environment variables):

-b     : issue warnings about str(bytes_instance), str(bytearray_instance)

         and comparing bytes/bytearray with str. (-bb: issue errors)

-B     : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x

-c cmd : program passed in as string (terminates option list)

-d     : debug output from parser; also PYTHONDEBUG=x

-E     : ignore PYTHON* environment variables (such as PYTHONPATH)

…
```

POWERED BY datalab | POWERED BY datalab

This will run the command `python –help` and create a new Popen object, which is stored in the variable p. The standard output and error of the command are captured using the communicate() method and stored in the variables output and errors, respectively.

`subprocess.Popen` is useful when you want more control over the process, such as sending input to it, receiving output from it, or waiting for it to complete.

python subprocess call

`subprocess.call()` is a function in the Python subprocess module that is used to run a command in a separate process and wait for it to complete. It returns the return code of the command, which is zero if the command was successful, and non-zero if it failed.

The call() function takes the same arguments as run(), including the args which specify the

command to be run, and other optional arguments, such as stdin, stdout, stderr, shell, cwd, and env.

The standard output and error of the command are sent to the same stdout and stderr as the parent process unless you redirect them using stdout and stderr arguments.

```
import subprocess

return_code = subprocess.call(["python", "--version"])

if return_code == 0:

    print("Command executed successfully.")

else:

    print("Command failed with return code", return_code)
```

Explain code | Run code | POWERED BY datalab

Output:

```
Command executed successfully.
```

Explain code | Run code | POWERED BY datalab

This will run the command `python –version` in a separate process and wait for it to complete. The command's return code will be stored in the return_code variable, which will be zero if the command was successful, and non-zero if it failed.

subprocess.call() is useful when you want to run a command and check the return code, but do not need to capture the output.

python subprocess check_output

check_output is a function in the subprocess module that is similar to run(), but it only returns the standard output of the command, and raises a CalledProcessError exception if the return code is non-zero.

The check_output function takes the same arguments as run(), including the args which specify the command to be run, and other optional arguments, such as stdin, stderr, shell, cwd, and env.

The check_output function returns the standard output of the command as a bytes object or string, if text=True is passed.

```
import subprocess

try:

    output = subprocess.check_output(["python", "--version"], text=True)

    print(output)

except subprocess.CalledProcessError as e:

    print(f"Command failed with return code {e.returncode}")
```

Explain code | Run code | POWERED BY datalab

Output:

```
Python 3.8.8
```

Explain code | Run code | POWERED BY datalab

Python Subprocess Pipe

Python subprocess module provides a way to create and interact with child processes, which can be used to run other programs or commands. One of the features of the subprocess module is the ability to create pipes, which allow communication between the parent and child processes.

A pipe is a unidirectional communication channel that connects one process's standard output to another's standard input. A pipe can connect the output of one command to the input of another, allowing the output of the first command to be used as input to the second command.

Pipes can be created using the subprocess module with the Popen class by specifying the stdout or stdin argument as subprocess.PIPE.

stdout or stdin argument as subprocess.PIPE.

For example, the following code creates a pipe that connects the output of the ls command to the input of the grep command, which filters the output to show only the lines that contain the word "file":

```
import subprocess

ls_process = subprocess.Popen(["ls"], stdout=subprocess.PIPE, text=True)

grep_process = subprocess.Popen(["grep", "sample"], stdin=ls_process.stdout, stdo

output, error = grep_process.communicate()

print(output)

print(error)
```

Explain code | Run code | POWERED BY datalab

Output:

```
sample_data
```

Explain code | Run code | POWERED BY datalab

In this example, the Popen class is used to create two child processes, one for the ls command and one for the grep command. The stdout of the ls command is connected to the stdin of the grep command using subprocess.PIPE, which creates a pipe between the two processes. The communicate() method is used to send the output of the ls command to the grep command and retrieve the filtered output.

Conclusion

The Python `subprocess` module provides a powerful and flexible way to create and interact with child processes, allowing you to run other programs or issue commands from within your Python script. From simple commands like subprocess.call() to more advanced features like pipes, redirecting input and output, and passing environment variables, the subprocess module has something to offer for almost every use case. It is a great way to automate repetitive tasks, run system commands, and even interact with other programming languages and platforms.

While working with the subprocess module, it's important to remember that running external commands poses a security risk, especially when using the shell=True parameter or passing unsanitized input. It's always a good practice to use the subprocess.run() function that allows you to specify various options for how the command should be run, such as whether to raise an exception if the command fails.

If you are interested in doing a deep dive into endless possibilities for command line automation through Python, check out our **Command Line Automation** in Python course. In this course, you will learn to write an automation code that will browse a filesystem, look for files that follow a pattern, and then determine whether files are duplicates in one of the numerous cases. After finishing the course, you'll be able to manage and interact with Unix processes as well as automate a variety of routine file system activities.

Python Subprocess FAQs

What is the difference between subprocess.call() and subprocess.run()?

subprocess.call() is a function that runs a command and waits for it to complete, returning the return code of the command. subprocess.run() is a more powerful function that allows you to run a command, capture its output, and specify various options for how the command should be run, such as whether to raise an exception if the command fails.

How can I run a command and capture its output in Python using subprocess?

How can I pass variables as arguments to a command in a subprocess?

How can I redirect the output of a command to a file using subprocess?

Moez Ali
Data Scientist, Founder & Creator of PyCaret

TOPICS

Python

Python Multiprocessing Tutorial

How can I run a command in the background using subprocess?

How can I check the return code of a command run using subprocess?

How can I pass environment variables to a command run using subprocess?

TOPICS

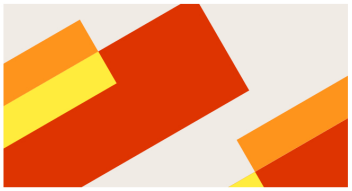
Python

Python Tutorial for Beginners

How to Run Python Scripts Tutorial

Python Count Tutorial

Related



TUTORIAL

Python Multiprocessing Tutorial

Discover the basics of multiprocessing in Python and the benefits it can bring to your workflows.

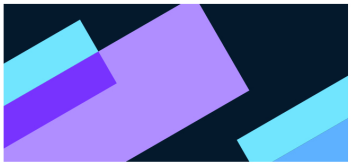
Kurtis Pykes • 6 min

TUTORIAL

Python Tutorial for Beginners

Get a step-by-step guide on how to install Python and use it for basic data science functions.

Matthew Przybyla • 12 min

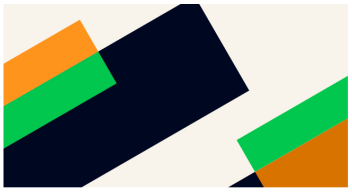


TUTORIAL

How to Run Python Scripts Tutorial

Learn how you can execute a Python script from the command line, and also how you can provide command line...

Aditya Sharma • 10 min

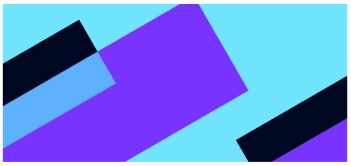


TUTORIAL

Python Count Tutorial

Learn how to use the counter tool with an interactive example.

DataCamp Team • 3 min

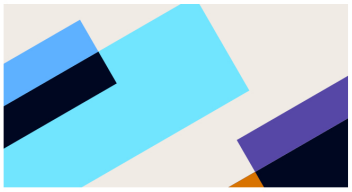


TUTORIAL

Python Setup: The Definitive Guide

In this tutorial, you'll learn how to set up your computer for Python development, and explain the basics for having the bes...

J. Andrés Pizarro • 15 min



TUTORIAL

Argument Parsing in Python

In this tutorial, learn how to parse one or more arguments from the command-line or terminal using the getopt, sys, and...

Sayak Paul • 10 min

See More →