**The ever useful and neat subprocess module**

29 Apr, 2012

Python's [subprocess](http://docs.python.org/library/subprocess.html) module is one of my favourite modules in the standard library. If you have ever done some decent amount of coding in python, you might have encountered it. This module is used for dealing with external commands, intended to be a replacement to the old [os.system](http://docs.python.org/library/os.html#os.system) and the like.

The most trivial use might be to get the output of a small shell command like ls or ps. Not that this is the best way to get a list of files in a directory (think [os.listdir](http://docs.python.org/library/os.html#os.listdir)), but you get the point.

I am going to put my notes and experiences about this module here. Please note, I wrote this with Python 2.7 in mind. Things **are** slightly different in other versions (even 2.6). If you find any errors or suggestions, please let me know.

* [A simple usage](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#a-simple-usage)
* [Popen class](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#popen-class)
* [Running via the shell](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#running-via-the-shell)
* [Getting the return code (aka exit status)](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#getting-the-return-code-aka-exit-status)
* [IO Streams](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#io-streams)
  + [Reading error stream](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#reading-error-stream)
  + [Watching both stdout and stderr](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#watching-both-stdout-and-stderr)
* [Passing an environment](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#passing-an-environment)
  + [Merge with current environment](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#merge-with-current-environment)
  + [Unicode](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#unicode)
* [Execute in a different working directory](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#execute-in-a-different-working-directory)
* [Killing and dieing](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#killing-and-dieing)
  + [Auto-kill on death](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#auto-kill-on-death)
* [Launch commands in a terminal emulator](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#launch-commands-in-a-terminal-emulator)
  + [Linux](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#linux)
  + [Windows](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#windows)
* [Conclusion](http://sharats.me/the-ever-useful-and-neat-subprocess-module.html#conclusion)

**A simple usage**

For the sake of providing context, lets run the ls command from subprocess and get its output

import subprocess

ls\_output = subprocess.check\_output(['ls'])

I'll cover getting output from a command in detail later. To give more command line arguments,

subprocess.check\_output(['ls', '-l'])

The first item in the list is the executable and rest are its command line arguments (argv equivalent). No quirky shell quoting and complex nested quote rules to digest. Just a plain python list.

However, not having shell quoting implies you don't also have the shell niceties. Like piping for one. The following won't work the way one would expect it to.

subprocess.check\_output(['ls', '|', 'wc', '-l'])

Here, the ls command gets its first command as | and I have no idea what ls would do with it. Perhaps complain that no such file exists. So, instead, we have to use the shell boolean argument. More later down in the article.

**Popen class**

If there's just one thing in the subprocess module that you should be concerned with, its the [Popen](http://docs.python.org/library/subprocess.html#subprocess.Popen) class. The other functions like [call](http://docs.python.org/library/subprocess.html#subprocess.call), [check\_output](http://docs.python.org/library/subprocess.html#subprocess.check_call), and [check\_call](http://docs.python.org/library/subprocess.html#subprocess.check_call) use Popen internally. Here's the signature from the docs.

class subprocess.Popen(args, bufsize=0, executable=None, stdin=None,

stdout=None, stderr=None, preexec\_fn=None, close\_fds=False, shell=False,

cwd=None, env=None, universal\_newlines=False, startupinfo=None,

creationflags=0)

I suggest you read the docs for this class. As with all python docs, its really good.

**Running via the shell**

Subprocess can also run command-line instructions via a shell program. This is usually dash/bash on Linux and cmd on windows.

subprocess.call('ls | wc -l', shell=True)

Notice that in this case we pass a string, not a list. This is because we want the shell to interpret the whole of our command. You can even use shell style quoting if you like. It is up to the shell to decide how to best split the command line into executable and command line arguments.

On windows, if you pass a list for args, it will be turned into a string using the same rules as the MS C runtime. See the doc-string for subprocess.list2cmdline for more on this. Whereas on unix-like systems, even if you pass a string, its turned into a list of one item :).

The behaviour of the shell argument can sometimes be confusing so I'll try to clear it a bit here. Something I wished I had when I first encountered this module.

Firstly, lets consider the case where shell is set to False, the default. In this case, if args is a string, it is assumed to be the name of the executable file. Even if it contains spaces. Consider the following.

subprocess.call('ls -l')

This won't work because subprocess is looking for an executable file called ls -l, but obviously can't find it. However, if args is a list, then the first item in this list is considered as the executable and the rest of the items in the list are passed as command line arguments to the program.

subprocess.call(['ls', '-l'])

does what you think it will.

Second case, with shell set to True, the program that actually gets executed is the OS default shell, /bin/sh on Linux and cmd.exe on windows. This can be changed with the executable argument.

When using the shell, args is usually a string, something that will be parsed by the shell program. The args string is passed as a command line argument to the shell (with a -c option on Linux) such that the shell will interpret it as a shell command sequence and process it accordingly. This means you can use all the shell builtins and goodies that your shell offers.

subprocess.call('ls -l', shell=True)

is similar to

$ /bin/sh -c 'ls -l'

In the same vein, if you pass a list as args with shell set to True, all items in the list are passed as command line arguments to the shell.

subprocess.call(['ls', '-l'], shell=True)

is similar to

$ /bin/sh -c ls -l

which is the same as

$ /bin/sh -c ls

since /bin/sh takes just the argument next to -c as the command line to execute.

**Getting the return code (aka exit status)**

If you want to run an external command and its return code is all you're concerned with, the [call](http://docs.python.org/library/subprocess.html#subprocess.call) and [check\_call](http://docs.python.org/library/subprocess.html#subprocess.check_call) functions are what you're looking for. They both return the return code after running the command. The difference is, check\_call raises a CalledProcessError if the return code is non-zero.

If you've read the docs for these functions, you'll see that its not recommended to use stdout=PIPE or stderr=PIPE. And if you don't, the stdout and stderr of the command are just redirected to the parent's (Python VM in this case) streams.

If that is not what you want, you have to use the Popen class.

proc = Popen('ls')

The moment the Popen class is instantiated, the command starts running. You can wait for it and after its done, access the return code via the [returncode](http://docs.python.org/library/subprocess.html#subprocess.Popen.returncode) attribute.

proc.wait()

print proc.returncode

If you are trying this out in a python REPL, you won't see a need to call [.wait()](http://docs.python.org/library/subprocess.html#subprocess.Popen.wait) since you can just wait yourself in the REPL till the command is finished and then access the returncode. Surprise!

>>> proc = Popen('ls')

>>> file1 file2

>>> print proc.returncode

None

>>> # wat?

The command is definitely finished. Why don't we have a return code?

>>> proc.wait()

0

>>> print proc.returncode

0

The reason for this is the returncode is not automatically set when a process ends. You have to call .wait or [.poll](http://docs.python.org/library/subprocess.html#subprocess.Popen.poll) to realize if the program is done and set the returncode attribute.

**IO Streams**

The simplest way to get the output of a command, as seen previously, is to use the [check\_output](http://docs.python.org/library/subprocess.html#subprocess.check_call) function.

output = subprocess.check\_output('ls')

Notice the check\_ prefix in the function name? Ring any bell? That's right, this function will raise a CalledProcessError if the return code is non-zero.

This may not always be the best solution to get the output from a command. If you do get a CalledProcessError from this function call, unless you have the contents of stderr you probably have little idea what went wrong. You'll want to know what's written to the command's stderr.

**Reading error stream**

There are two ways to get the error output. First is redirecting stderr to stdout and only being concerned with stdout. This can be done by setting the stderr argument to [subprocess.STDOUT](http://docs.python.org/library/subprocess.html#subprocess.STDOUT).

Second is to create a Popen object with stderr set to [subprocess.PIPE](http://docs.python.org/library/subprocess.html#subprocess.PIPE) (optionally along with stdout argument) and read from its stderr attribute which is a readable file-like object. There is also a convenience method on Popen class, called .communicate, which optionally takes a string to be sent to the process's stdin and returns a tuple of (stdout\_content, stderr\_content).

**Watching both stdout and stderr**

However, all of these assume that the command runs for some time, prints out a couple of lines of output and exits, so you can get the output(s) in strings. This is sometimes not the case. If you want to run a network intensive command like an svn checkout, which prints each file as and when downloaded, you need something better.

The initial solution one can think of is this.

proc = Popen('svn co svn+ssh://myrepo', stdout=PIPE)

for line in proc.stdout:

print line

This works, for the most part. But, again, if there is an error, you'll want to read stderr too. It would be nice to read stdout and stderr simultaneously. Just like a shell seems to be doing. Alas, this remains a not so straightforward problem as of today, at least on non-Linux systems.

On Linux (and where its supported), you can use the [select](http://docs.python.org/library/select.html) module to keep an eye on multiple file-like stream objects. But this isn't available on windows. A more platform independent solution that I found works well, is using threads and a [Queue](http://docs.python.org/library/queue.html#queue-objects).

from subprocess import Popen, PIPE

from threading import Thread

from Queue import Queue, Empty

io\_q = Queue()

def stream\_watcher(identifier, stream):

for line in stream:

io\_q.put((identifier, line))

if not stream.closed:

stream.close()

proc = Popen('svn co svn+ssh://myrepo', stdout=PIPE, stderr=PIPE)

Thread(target=stream\_watcher, name='stdout-watcher',

args=('STDOUT', proc.stdout)).start()

Thread(target=stream\_watcher, name='stderr-watcher',

args=('STDERR', proc.stderr)).start()

def printer():

while True:

try:

# Block for 1 second.

item = io\_q.get(True, 1)

except Empty:

# No output in either streams for a second. Are we done?

if proc.poll() is not None:

break

else:

identifier, line = item

print identifier + ':', line

Thread(target=printer, name='printer').start()

Fair bit of code. This is a typical producer-consumer thing. Two threads producing lines of output (one each from stdout and stderr) and pushing them into a queue. One thread watching the queue and printing the lines until the process itself finishes.

**Passing an environment**

The env argument to Popen (and others) lets you customize the environment of the command being run. If it is not set, or is set to None, the current process's environment is used, just as documented.

You might not agree with me, but I feel there are some subtleties with this argument that should have been mentioned in the documentation.

**Merge with current environment**

One is that if you provide a mapping to env, whatever is in this mapping is all that's available to the command being run. For example, if you don't give a TOP\_ARG in the env mapping, the command won't see a TOP\_ARG in its environment. So, I frequently find myself doing this

p = Popen('command', env=dict(os.environ, my\_env\_prop='value'))

This makes sense once you realize it, but I wish it were at least *hinted at* in the documentation.

**Unicode**

Another one, is to do with Unicode (Surprise surprise!). And windows. If you use unicodes in the env mapping, you get an error saying you can *only* use strings in the environment mapping. The worst part about this error is that it only seems to happen on windows and not on Linux. If its an error to use unicodes in this place, I wish it break on both platforms.

This issue is very painful if you're like me and use unicode*all the time*.

from \_\_future\_\_ import unicode\_literals

That line is present in all my python source files. The error message doesn't even bother to mention that you have unicodes in your env so it's very hard to understand what's going wrong.

**Execute in a different working directory**

This is handled by the cwd argument. You set the location of the directory which you want as the working directory of the program you are launching.

The docs do mention that the working directory is changed *before* the command even starts running. But that you *can't* specify program's path relative to the cwd. In reality, I found that you *can* do this.

Either I'm missing something with this or the docs really are inaccurate. Anyway, this works

subprocess.call('./ls', cwd='/bin')

Prints out all the files in /bin. Of course, the following doesn't work when the working directory is not /bin.

subprocess.call('./ls')

So, if you are giving something explicitly to cwd and are using a relative path for the executable, this is something to keep in mind.

**Killing and dieing**

A simple

proc.terminate()

or for some dramatic umphh!

proc.kill()

will do the trick to end the process. As noted in the documentation, the former sends a SIGTERM and later sends a SIGKILL on unix, but both do some native windows-y thing on windows.

**Auto-kill on death**

The processes you start in your python program, stay running even after your program exits. This is *usually* what you want, but when you want all your sub processes killed automatically on exit with Ctrl+C or the like, you have to use the [atexit](http://docs.python.org/library/atexit.html) module.

procs = []

@atexit.register

def kill\_subprocesses():

for proc in procs:

proc.kill()

And add all the Popen objects created to the procs list. This is the only solution I found that works best.

**Launch commands in a terminal emulator**

On one occasion, I had to write a script that would launch multiple svn checkouts and then run many ant builds (~20-35) on the checked out projects. In my opinion, the best and easiest way to do this is to fire up multiple terminal emulator windows each running an individual checkout/ant-build. This allows us to monitor each process and even cancel any of them by simply closing the corresponding terminal emulator window.

**Linux**

This is pretty trivial actually. On Linux, you can use xterm for this.

Popen(['xterm', '-e', 'sleep 3s'])

**Windows**

On windows, its not as straight forward. The first solution for this would be

subprocess.Popen(['cmd','/K','git pull origin master'])

/K option tells cmd to run the command and keep the command window from closing. You may use /C instead to close the command window after the command finishes.

As simple as it looks, it has some weird behavior. I don't completely understand it, but I'll try to explain what I have. When you try to run a python script with the above Popen call, in a command window like this

python main.py

you *don't* see a new command window pop up. Instead, the sub command runs in the same command window. I have no idea what happens when you run multiple sub commands this way. (I have only limited access to windows).

If instead you run it in something like an IDE or IDLE (F5), you have a new command window open up. I believe one each for each command you run this way. Just the way you expect.

But I gave up on cmd.exe for this purpose and learnt to use the [mintty](https://code.google.com/p/mintty/) utility that comes with [cygwin](http://www.cygwin.com/) (I think 1.7+). mintty is awesome. Really. Its been a while since I felt that way about a command line utility on windows.

Popen(['mintty', '--hold', 'error', '--exec', 'command'])

This. A new mintty console window opens up running the command and it closes automatically, *if* the command exits with zero status (that's what --hold error does). Otherwise, it stays on. Very useful.

**Conclusion**

The subprocess module is a very useful thing. Spend some time understanding it better. This is my attempt at helping people with it, and turned out to be way longer than I'd expected. If there are any inaccuracies in this, or if you have anything to add, please leave a comment.