Python 3: Child processes

Bob Dowling rjd4@cam.ac.uk

29 October 2012

Prerequisites

This self-paced course assumes that you have a knowledge of Python 3 equivalent to having completed one or other of

- Python 3: Introduction for Absolute Beginners, or
- Python 3: Introduction for Those with Programming Experience

Some experience beyond these courses is always useful but no other course is assumed.

The course also assumes that you know how to use a Unix text editor (`gedit`, `emacs`, `vi`, …).

Facilities for this session

The computers in this room have been prepared for these self-paced courses. They are already logged in with course IDs and have home directories specially prepared. Please do not log in under any other ID.

At the end of this session the home directories will be cleared. Any files you leave in them will be deleted. Please copy any files you want to keep.

The home directories contain a number of subdirectories one for each topic.

For this topic please enter directory `subprocess`. All work will be completed there:

```
$ cd subprocess
$ pwd
/home/x250/subprocess
$
```

These courses are held in a room with two demonstrators. If you get stuck or confused, or if you just have a question raised by something you read, please ask!

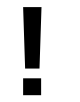
These handouts and the prepared folders to go with them can be downloaded from
`www.ucs.cam.ac.uk/docs/course-notes/unix-courses/pythontopics`

The formal Python 3 documentation for the topics covered here can be found online at
`docs.python.org/release/3.2.3/library/subprocess.html`

Table of Contents

Notation

Warnings



Warnings are marked like this. These sections are used to highlight common mistakes or misconceptions.

Exercises

Exercise 0

Exercises are marked like this. You are expected to complete all exercises. Some of them do depend on previous exercises being successfully completed.

Input and output

Material appearing in a terminal is presented like this:

```
$ more lorem.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
--More--(44%)
```

The material you type is presented like this: **`ls`**. (Bold face, typewriter font.)

The material the computer responds with is presented like this: “`Lorem ipsum`”. (Typewriter font again but in a normal face.)

Keys on the keyboard

Keys on the keyboard will be shown as the symbol on the keyboard surrounded by square brackets, so the “A key” will be written “[A]”. Note that the return key (pressed at the end of every line of commands) is written “[↲]”, the shift key as “[⇧]”, and the tab key as “[↹]”. Pressing more than one key at the same time (such as pressing the shift key down while pressing the A key) will be written as “[⇧]+[A]”. Note that pressing [A] generates the lower case letter “a”. To get the upper case letter “A” you need to press [⇧]+[A].

Content of files

The content[1] of files (with a comment) will be shown like this:

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur.                                  This is a comment about the line.
```

1 The example text here is the famous “lorem ipsum” dummy text used in the printing and typesetting industry. It dates back to the 1500s. See http://www.lipsum.com/ for more information.

What’s in this course

This course is for people who want to launch other programs from their Python scripts, either one at a time, several at the same time, or several all linked together.

1. Running a single program
2. Writing to files
3. Reading from files
4. Catching the results in a string
5. Running programs in the background
6. Piping programs together
7. Communicating with running processes

Running a simple program

Let's start simple. Suppose we want to run "`ls -l`" from within Python (and we don't want to roll our own with the `os` module).

The module for managing subprocesses is called "`subprocess`" and contains a function "`call()`" which allows you to call another program from within Python. The script in your directory, `example01.py`, illustrates exactly this process:

```
import subprocess
print('About to run ls.')
subprocess.call(['ls', '-l'])
print('Finished running ls.')
```

and performs like this:

```
$ python3 example01.py
About to run ls.
total 735
drwx------ 0 rjd4 rjd4      0 Oct 17 15:04 alpha
drwx------ 0 rjd4 rjd4      0 Oct 17 15:04 beta
-rwx------ 0 rjd4 rjd4    103 Oct 17 15:04 example01.py
-rwx------ 0 rjd4 rjd4    138 Oct 17 15:04 example02.py
-rwx------ 0 rjd4 rjd4    158 Oct 17 15:04 example03.py
-rwx------ 0 rjd4 rjd4    150 Oct 17 15:04 example04.py
-rwx------ 0 rjd4 rjd4    183 Oct 17 15:04 example05.py
-rwx------ 0 rjd4 rjd4    174 Oct 17 15:04 example06.py
-rwx------ 0 rjd4 rjd4    120 Oct 17 15:04 example07.py
-rwx------ 0 rjd4 rjd4    173 Oct 17 15:04 example08.py
-rwx------ 0 rjd4 rjd4    342 Oct 17 15:04 example09.py
-rwx------ 0 rjd4 rjd4    237 Oct 17 15:04 example10.py
-rwx------ 0 rjd4 rjd4     42 Oct 17 15:20 exercise01.py
-rwx------ 0 rjd4 rjd4   7544 Oct 17 15:15 iterator
-rwx------ 0 rjd4 rjd4  75674 Oct 17 15:04 notes.odt
-rwx------ 0 rjd4 rjd4 283961 Oct 17 15:04 notes.pdf
-rwx------ 0 rjd4 rjd4    534 Oct 17 15:04 plot_points
-rwx------ 0 rjd4 rjd4 364589 Oct 17 15:04 treasure.txt
Finished running ls.
$
```

This is a really trivial example but illustrates a few points:

- The final `print()` statement isn't run until the run of "`ls -l`" is finished.
- The output of `ls` goes to the same place as the output of `print()`.
- The command launched needs to be split into a list of arguments; it is not just a string.

This script uses a fixed command line. Obviously you could use Python's list manipulations to change it.

Exercise 1

Edit the script `exercise01.py` to run the command

```
./iterator 0.60
```

The script should generate a lot of numerical data on the screen.

All commands return a simple numerical value to indicate whether they completed successfully. This is called the "return code" for the run. A value of zero (0) indicates that the program completed successfully. A non-zero value indicates that there was a problem. Most programs return one (1) if there was any problem; a few

have various different non-zero values depending on what the problem was. A return code of 127 means that the command could not be found.

The `subprocess.call()` function returns this code to the script, as shown in `example02.py`:

```
import subprocess
print('About to run ls.')
rc = subprocess.call(['ls', '-l'])
print('Finished running ls.')
print('RC = {:d}'.format(rc))
```

```
$ python3 example02.py
About to run ls.
Total 36
-rwx------. 1 rjd4 rjd4   103 Oct 14 15:54 example01.py
-rwx------. 1 rjd4 rjd4   138 Oct 14 15:56 example02.py
-rw-r--r--. 1 rjd4 rjd4 28209 Oct 14 15:48 notes.odt
Finished running ls.
RC = 0
$
```

Obviously we may not want to have our output displayed to the screen. There are two common alternative ways to capture the output:

- to a file
- as a text object

Capturing the output in a file

The shell has redirection operators like “>” to divert standard output to a file. Python’s `subprocess` module has something equivalent if not so syntactically elegant.

First we have to open a file for writing to to put our output in. We will call ours `ls_out.txt` for these examples:

```
ls_output = open('ls_out.txt', 'w')
```

This gives us a Python file object. Next we have to tell `subprocess.call()` to use it with the `stdout` parameter:

```
rc = subprocess.call(['ls', '-l'], stdout=ls_output)
```

After the program is complete, we need to close the file:

```
ls_output.close()
```

An example of this exists in `example03.py` (which no longer has the “about to…” and “finished…” lines):

```
$ python3 example03.py
RC = 0
$ more ls_out.txt
total 44
-rwx------. 1 rjd4 rjd4   103 Oct 14 15:54 example01.py
-rwx------. 1 rjd4 rjd4   138 Oct 14 15:56 example02.py
-rw-rw-r--. 1 rjd4 rjd4   158 Oct 14 16:14 example03.py
-rw-rw-r--. 1 rjd4 rjd4     0 Oct 14 16:15 ls_out.txt
-rw-r--r--. 1 rjd4 rjd4 28767 Oct 14 16:12 notes.odt
$
```

Apart from our printing of the return code the example script was completely silent; all the output went to the file.

Exercise 2

Copy the script `exercise01.py` to `exercise02.py` and edit it to divert the standard output (the numerical data) to a file called `output.dat`.

You need to complete this exercise before attempting the next one, as you will be using the `output.dat` file.

Reading the input from a file

The converse of writing output to a file is reading the command's input from a file.

We have already noted that commands have a standard output that goes to the terminal by default and which we can divert to an open Python file object. Similarly, they have a "standard input" which reads from the keyboard by default, but which can be told to read from an open file object.

We will switch from using the ls program for this example, as it does not read in any input. Instead we will use the "wc" program which, if used without any arguments, reads its standard input and prints out the numbers of lines, words and characters it receives. Just as there is a `stdout=…` parameter in `subprocess.call()` to specify standard output there is a `stdin=…` parameter to specify standard input.

This is demonstrated in `example04.py` which reads in the text of "Treasure Island" and feeds it to `wc`:

```
import subprocess
wc_input = open('treasure.txt', 'r')
rc = subprocess.call(['wc'], stdin=wc_input)
wc_input.close()
print('RC = {:d}'.format(rc))
```

This behaves as follows:

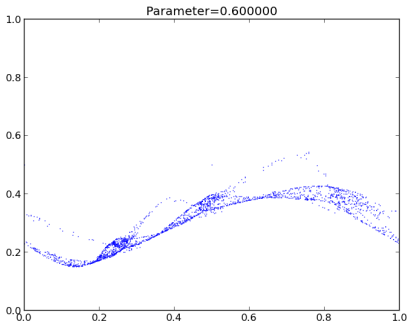
```
$ python3 example04.py
  8734  67945 364589
RC = 0
$
```

There is nothing to stop us specifying files for both standard input and standard output.

Exercise 3

Note: You should already have a data file `output.dat` from the previous exercise. You will need it for this one.

There is a program called "`plot_points`" in your directory that takes the data from the `iterator` program on its standard input and plots it on a graph in a file.

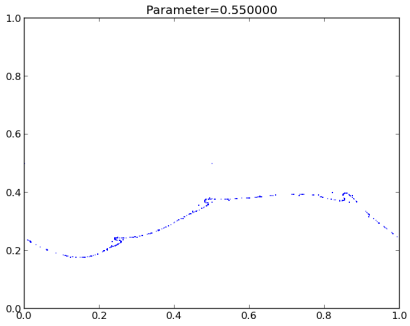


Create a Python script called `exercise03.py` which runs the program `./plot_points` with its standard input read from the file `output.dat`.

If it runs correctly it will create a file called `graph_0.600000.png` which you can view with the `eog` program, or by double-clicking on its icon in the file browser.

```
$ eog graph_0.600000.png
```

Exercise 4



Combine `exercise02.py` and `exercise03.py` to create a Python script, `exercise04.py`, that runs `./iterator` with parameter 0·55 to create a new `output.dat` and then runs `./plot_points` to generate a new graph file, `graph_0.550000.png`.

Don't forget to close the file after writing it before opening it for reading the data back again.

```
$ rm output.dat
$ python3 exercise05.py
$ eog graph_0.550000.png
```

Capturing the output in a string

There is an alternative to `subprocess.call()` called "`subprocess.check_output()`" which diverts the standard output into an array of bytes, which can then be converted into a standard Python text object. The script `example05.py` illustrates this approach:

```
import subprocess

ls_output_raw = subprocess.check_output(['ls', '-l'])
ls_output_text = ls_output_raw.decode('UTF-8')

print(ls_output_text)
```

All modern system use the UTF-8 character encoding.

If you want to work through the output line by line then you may find it convenient to split your block of text into its lines:

```
ls_output_lines = ls_output_text.splitlines()
```

This method automatically adapts for the line terminators on your particular system.

Exercise 5

Copy the script `example01.py` to `exercise05.py` and edit it to capture the output as text. Split the text into lines and print all bar the first line in the list.

NB: Start from **`example01.py`**.

Like `subprocess.call()`, `subprocess.check_output()` does not return until the command is finished. If it is going to generate a lot of text output you might want to get at it early. There are ways to do that and we will see them when we run commands in the background.

Running programs in the background

Suppose we have a number of programs we want to run at the same time, each of which might take some significant amount of time. We want to launch them all to run at the same time, in parallel, and then have the script wait for all the background tasks to finish before it finishes itself.

Some of the examples in this section involve the output of two commands running at the same time intermingling. If you don't see exactly the same output as the printed examples don't be surprised. If you don't see the expected behaviour at all, try running the command again. If you persistently don't see it, ask a demonstrator.

To do this we replace `subprocess.call()` with a different function `subprocess.Popen()` (NB the capital "P") that launches the program but does not wait for it to finish before moving on to the next line of the Python script. Obviously the function can't give the function's return code because the function hasn't completed yet. What it returns is a "handle" on the running process which we will examine in more detail shortly.

Consider these two lines in `example06.py`:

```
alpha_rc = subprocess.call(['ls', '-l', 'alpha'])
beta_rc  = subprocess.call(['ls', '-l', 'beta'])
```

These run "`ls -l`" on the `alpha` directory and then the `beta` directory:

```
$ python3 example06.py
total 0
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 alpha_01
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 alpha_02
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 alpha_03
 ...
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 beta_97
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 beta_98
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 beta_99
$
```

Now compare them with the `subprocess.Popen()` equivalent lines in `example07.py`:

```
import subprocess
alpha_p  = subprocess.Popen(['ls', '-l', 'alpha'])
beta_p   = subprocess.Popen(['ls', '-l', 'beta'])
```

which runs the two `ls` processes at the same time, intermingling their outputs:

```
$ python3 example07.py
total 0
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 alpha_01
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 alpha_02

total 0
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 beta_01
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 alpha_03
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 alpha_04
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 beta_02
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 beta_03
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 alpha_05

...
$

...
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 beta_98
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 alpha_99
-rw-rw-r--. 1 rjd4 rjd4 0 Oct 16 20:57 beta_99
```

Note also that the Unix system prompt, "$", appears in the middle of the output. This indicates that the Python script has finished before the two `ls` processes have!

Next we are going to arrange for the Python script to wait for the two `ls` processes (known as "child processes") to finish before it does.

The `subprocess.Popen()` function returns a handle on the running function, rather than the command's return code. We can use that handle to wait for the running process to come to an end. The handle is a Python object with a method called "`wait()`" which pauses until the corresponding process completes. This is called "waiting for your children to die" in the deliciously morbid Unix terminology.

The script `example08.py` extends `example07.py` to include two `wait()` calls:

```
import subprocess

alpha_p  = subprocess.Popen(['ls', '-l', 'alpha'])
beta_p   = subprocess.Popen(['ls', '-l', 'beta'])

alpha_rc = alpha_p.wait()
beta_rc  = beta_p.wait()
```

This still causes the two runs of `ls` to intermingle but the Unix prompt (indicting the end of the script) does not come back until they are both complete (and have been `wait`ed for).

A few notes on `wait()`:

1. The `wait()` methods return the corresponding processes' return codes. Now that the processes are complete, this information is available and this is how it is passed over.
2. The `subprocess.call()` function can be thought of as a `subprocess.Popen()` immediately followed by a call to the `wait()` method:

   (i) `rc = subprocess.call(…)` → `rc = subprocess.Popen.wait(…)`

   (ii) `rc = subprocess.call(…)` →
   ```
   process = subprocess.Popen(…)
   rc = process.wait(…)
   ```
3. It doesn't matter what order the `wait()` methods get called in, so long as all background processes are `wait`ed for.
4. The `wait()` method pauses for the corresponding process to complete. If it has already completed then `wait()` returns immediately. Child processes that have finished and are just hanging around so that they can hand over their return codes to a `wait()` call are called "zombie processes".

Exercise 6

Copy your script `exercise04.py` to `exercise06.py` and edit it to replace each use of `subprocess.call()` with a call to `subprocess.Popen()` (recording the process handle) immediately followed by a call to the `wait()` method on that process handle.

This gives the pattern `Popen()`…`wait()`…`Popen()`…`wait()`… which is not particularly useful in itself but sets us up for the next exercise.

What might go wrong if you had scripted `Popen()`…`Popen()`…`wait()`… `wait()`…?

Piping programs together

To date, we have run commands with `subprocess.call()`, `subprocess.check_output()`, or `subprocess.Popen()` and we have had to wait for the program to complete before moving on to the next line of Python in the script. As a result, if we wanted to combine the `iterator` program (to generate some numerical output) and the `plot_points` program (to plot those points) we have to run `iterator` first to generate the data file, and then run `plot_points` to create the graph from the data file. If we wanted to leave no trace of the intermediate data file we would have to explicitly remove it afterwards.

Now we are going to run the two commands in parallel (with `subprocess.Popen()`) and connect the standard output of one with the standard input of the other. In Unix terminology this is called “piping” and the connector we will build between the two running processes is called a “pipe”.

For our worked examples we will do something singularly pointless; we will take the output of `ls` and feed it in as the input of `wc`. The exercises you do will connect `iterator` and `plot_points`, which is far more productive.

The script `example09.py` runs both `ls` and `wc` with a file as the interchange:

```
import subprocess

filename = 'interchange.dat'

# Run ls

output_file = open(filename, 'w')
alpha_p  = subprocess.Popen(['ls', '-l'], stdout=output_file)
alpha_rc = alpha_p.wait()
output_file.close()

# Run wc

input_file = open(filename, 'r')
beta_p   = subprocess.Popen(['wc'], stdin=input_file)
beta_rc  = beta_p.wait()
input_file.close()
```

and works as expected:

```
$ python3 example09.py
  21  182 1139
$
```

Now, we will start work on eliminating the interchange file. We will convert from `example09.py` to `example10.py` (which implements this) step-by-step.

We start by eliminating the lines that declare, open, or close the interchange file, of course.

Next, we turn to the run of `ls` where we need to declare that its output is to go to a pipe. There is a special object in the `subprocess` module specifically for this purpose, `subprocess.PIPE`, and we declare the standard output to be a pipe like this:

```
alpha_p  = subprocess.Popen(['ls', '-l'], stdout=subprocess.PIPE)
```

Pipes have a very important property: they can fill up.

We are writing to a relatively small buffer. Because we have not started the `wc` process yet, nothing is reading from (or "consuming") the pipe. As soon as the pipe fills the process pauses, waiting for more room to be made in the pipe. In `example09.py` we wait for the process to finish before we start `wc`, so this can lead to what is called "**deadlock**".

Avoid deadlock: when a script pipes processes together it must not wait for *any* of them until they are *all* started.

So we will move the line

```
alpha_rc = alpha_p.wait()
```

further down the script, below the line that launches `wc`.

Next, we need to tell the `subprocess.Popen()` that launches `wc` to get its standard input from the same pipe as `ls` is writing to. We want its standard input to be `ls`'s standard output. We do this by using the handle on the `ls` process, `alpha_p`, and asking it for its standard output, `alpha_p.stdout`:

```
beta_p   = subprocess.Popen(['wc'], stdin=alpha_p.stdout)
```

This leaves us with the script `example10.py`:

```
import subprocess

# Run ls & wc

alpha_p  = subprocess.Popen(['ls', '-l'], stdout=subprocess.PIPE)
beta_p   = subprocess.Popen(['wc'], stdin=alpha_p.stdout)

# Wait for children to die

alpha_rc = alpha_p.wait()
beta_rc  = beta_p.wait()
```

which runs in exactly the same way except that it does not create the intermediate file:

```
$ rm interchange.dat
$ python3 example10.py
     22     191    1196
$ ls interchange.dat
ls: cannot access interchange.dat: No such file or directory
$
```

More than one command joined together like this is called a "pipeline" and pipelines can have as many commands string together as you want.

Exercise 7

Copy your script `exercise06.py` to `exercise07.py` and edit it to use a pipe rather than an intermediate file.

Reading output from running processes

We have seen how to launch two (or more) programs and have them communicate. What we will look at next is launching one program and reading its output from the Python script itself.

There is a program in your directory called `collatz` which runs essentially for ever, generating lines of numerical output. Because this program never stops, we have to run it in the background. How can we read its data?

The file `example11.py` demonstrates the approach we have to take. It reads in the first 10,000 lines of output from a running process.

First, we have to launch it in the background, redirecting its standard output to a pipe:

```
collatz_p = subprocess.Popen(['./collatz'], stdout=subprocess.PIPE)
```

We treat `process.stdout` as a standard Python file object and read in lines from it, using the `next()` function and converting the raw data into a text string:

```
line_raw = next(collatz_p.stdout)
line_txt = line_raw.decode('UTF-8')
```

In the `example11.py` script this sits inside a loop to do this 10,000 times.

If the `./collatz` program never got as far as 10,000 lines of output, but never closed its output either (e.g. by ending) then the script would hang waiting for another line of output.

Future versions of Python 3 will have a `timeout=…` parameter to prevent this.

If our Python script simply took its 10,000 lines of data, did what it wanted to do and then quit it would leave behind a running background process that never quits. Obviously, the Python script needs to kill off the running `collatz` process. To do this, we introduce another method on the process handle, `terminate()`:

```
collatz_p.terminate()
```

The `terminate()` method sends a “polite” request to the running process to shut down. Unless the process has got stuck in some way this is typically sufficient. There is an alternative method, `kill()`, which is less polite.

Exercise 8

Edit the script `exercise08.py` to read in 10,000 lines of input from a running `./collatz` process and pass them to the plotting function provided.

Passing data into a background process

We have demonstrated how to pass command line arguments to the launched commands. We have seen how to read data from launched commands’ standard outputs into our Python scripts. What we have not looked at is how to send data from our Python scripts to a backgrounded process’ standard input.

To do this we will run a process (wc, the line/word/character counting program) whose standard input is defined to be a pipe:

```
wc_proc = subprocess.Popen(['wc'], stdin=subprocess.PIPE)
```

and then writing lines of data to it with `wc_proc.stdin.write()`.

Recall that when we were reading data from a process’s standard output we received raw data and had to convert it to text by calling a `decode('UTF-8')` method. We have to do the inverse when we want to write text to a process: we have to encode it into raw format before sending it on with an `encode()` method:

```
wc_proc.stdin.write(line.encode())
```

The file `example12.py` illustrates this process.

The `write()` call pauses (“blocks” in the technical vernacular) until the process is ready to read data.

If `subprocess.Popen()` is used with both standard input and standard output set to be pipes that are both accessed from the script be very careful. This is a recipe for deadlock.