BCH2203 Python - 9. Subprocess

Ramses van Zon

13 March 2024

In this lecture

Last lecture, we saw how to call blast from Python using Bio.Blast.Applications.

```
>>> from Bio.Blast.Applications import NcbiblastnCommandline
_init__.py:40: BiopythonDeprecationWarning: The Bio.Application modules and modules relying on it have be

Due to the on going maintenance burden of keeping command line application
wrappers up to date, we have decided to deprecate and eventually remove these
modules.

We instead now recommend building your command line and invoking it directly
with the subprocess module.
  warnings.warn(
>>> blastn_cmdline = NcbiblastnCommandline(query="dna.fasta", db="nt", evalue=0.001,
                                            outfmt=5, out="dna.xml")
>>> stdout, stderr = blastn_cmdline()
```

Looks like we got it wrong, this is now deprecated.

Instead, the Biopython documentation recommends using subprocess.

Subprocess

First, what are processes?

When Python runs, it is a process. When blastn runs, it is a process. When the shell runs, it is a process.

- Processes are managed and started by the operating system (OS).
  Many processes can run at the same time.
- The OS gives each process its own, private resources like memory.
- The OS reclaims those resources when the process ends.
- Processes can start other processes. These are called child processes or subprocesses.
  E.g. the shell can start python.

Inter-Process Communication

There are various ways for processes to communicate (including with subprocesses)

Input

1. Commmand line arguments when starting the (sub)process
2. Reading files.
   Usually the the start of a (sub)process.
3. Pipes, which use shared memory. This is a generalization of keyboard input.

Output

1. Printing to screen.
   Turns out to work via these Pipes too.
2. Output to a file. Usually the end of a (sub) process
3. Returning an exit code. (a number between 0 and 255).

How these work exactly depends on the OS, but there is a Python module that abstracts that out.

The subprocess module

```
import subprocess
```

- This Python module's purpose is to launch subprocesses. These do not need to be python scripts.
- It contains a "Popen" class to handle subprocesses.
- Often, one uses this class through one of the following functions:

  ```
  subprocess.run()
  subprocess.check_output()
  subprocess.call()
  subprocess.check_call()
  ```

  The first two are most common for python 3.5+, the latter two were useful for older python versions.
- These functions have many arguments, mostly to do with the interprocess communication.

Note:

`subprocess` replaces older ways that have issues, e.g. os.system, os.spawn, os.popen, popen2.

Let's start simple

```
>>> import subprocess
>>> subprocess.run("ls")
darktheme.tex               lecture09_subprocess.pdf   mds
lecture09_subprocess.tex  SciNetLogoTransparent.png  lecture09_subprocess.theme
lecture09_subprocess.md   Makefile
CompletedProcess(args='ls', returncode=0)
>>>
```

Here, `subprocess.run("ls")` called the `ls` application.

This suggests to just give a string with a command. But no:

```
>>> subprocess.run("ls mds")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.10/subprocess.py", line 503, in run
    with Popen(*popenargs, **kwargs) as process:
  File "/usr/lib/python3.10/subprocess.py", line 971, in __init__
    self._execute_child(args, executable, preexec_fn, close_fds,
  File "/usr/lib/python3.10/subprocess.py", line 1863, in _execute_child
    raise child_exception_type(errno_num, err_msg, err_filename)
FileNotFoundError: [Errno 2] No such file or directory: 'ls mds'
```

Command line arguments

- Cannot give application with spaces to pass arguments.
- When getting command line arguments in our python scripts, we use `sys.argv`, which is a list.
- Likewise, to give command line arguments to `subprocess.run`, you pass it a list:

```
>>> subprocess.run(["ls","mds"])
mds
CompletedProcess(args=['ls', 'mds'], returncode=0)
```

We see we get two kinds of output:

Output to console

Whatever the command would have printed on console, still is.

A 'CompletedProcess' with a returncode

If zero, this mean all went well.

If non-zero, there was an error.

Our scripts can also give exit codes with `exit(NUMBER)`.

Capturing the console output

What if we want to store the console output in a variable?

This does not work:

```
>>> result = subprocess.run(["ls","mds"])
mds
>>> print(result)
CompletedProcess(args=['ls', 'mds'], returncode=0)
```

Because we just get the `CompletedProcess` in `result`. The output still goes to the console.

Instead, we should add the option `capture_output=True`:

```
>>> result = subprocess.run(["ls","mds"],capture_output=True)
>>> print(result)
CompletedProcess(args=['ls', 'mds'], returncode=0, stdout=b'mds\n', stderr=b'')
>>> print(result.stdout)
b'mds\n'
>>> print(result.stdout.decode())
mds

>>>
```

Shortcut to capture output

The following are equivalent

```
>>> result = subprocess.run(["ls","mds"], capture_output=True).stdout
>>> print(result)
b'mds\n'
>>> print(result.decode())
mds

>>>
```

and

```
>>> result = subprocess.check_output(["ls","mds"])
>>> print(result)
b'mds\n'
>>> print(result.decode())
mds

>>>
```

Shortcut to capture output, except. . .

- With `check_output`, you don't get the exit code, nor any error messages.
- But if `check_output` detects an error, an exception is thrown (whereas `run` does not).

Shortcut to get output as a string

The above are further equivalent to:

```
>>> result = subprocess.run(["ls","mds"], capture_output=True, text=True).stdout
>>> print(result)
mds

>>>
```

and

```
>>> result = subprocess.check_output(["ls","mds"], text=True)
>>> print(result)
mds

>>>
```

Input

So we can get the output into a string in python.
What it the application we are calling needs (keyboard) input?

Both run and check_output have an ‘input’ parameters for that:

```
>>> import sys
>>> result = subprocess.run([sys.executable,
                             "-c",
                             "print(input())"],
                             text=True,
                             capture_output=True,
                             input="Hello\n").stdout
>>> # or, with check_output:
>>> result = subprocess.check_output([sys.executable,
                             "-c",
                             "print(input())"],
                             text=True,
                             input="Hello\n")
>>> print(result)
Hello

>>>
```

Note that here:

- `sys.executable` is the python interpreter used by this script.
- `-c` is an option to tell python to execute some following code.
- Here, that code takes an input and prints it.

More about streams

- Each process has one input stream, called stdin.

  When you use `input(...)`, you are reading from this input stream.
- Each process has an output stream, called stdout.

  When you use `print(...)`, you are writing to this output stream.
- In addition, each process has an error stream, called stderr.

  Here's where errors go.

  When you use `print(..., file=sys.stderr)`, you are writing to this error stream.

We have seen that we can attach a string to the stdin stream.

We have seen that we can capture the stdout from a CompletedProcess returned by `run`.

Similarly, we can get the stderr as a member of the result of `run`.

Pipes

- You can attach the output channel of one process (A) to the input channel of another process (B).
- This connection is called a pipe.
- Both process A and B need to be running at the same time.
- Data produced by A will continuously flow into B.

Unfortately, all functions we have seen so far are blocking;

They will wait for the process to finish.

To be able to get more that one process simultaneously, we need to use the Popen object directly.

In addition, we need to make it so that data flows through the pipe instead of to the console.

Pipes with Popen

Let P be a producer process that prints a string to its stdout.

Let C be a consumer process that prints some input from its stdin.

We want C to ingest from P, so we redirect P's output to a PIPE:

```
>>> from subprocess import PIPE
>>> P = subprocess.Popen([sys.executable,"-c","print('Hello')"], text=True, stdout=PIPE)
```

The producer P needs to know where its stdout should go, i.e., to A.

```
>>> C = subprocess.Popen([sys.executable,"-c","print(input())"], text=True, stdin=P.stdout)
>>> Hello
```

The appearance of "Hello" shows it is working, but still goes to the console.

Also, less apparently here, the processes C and P may still be running.
To be sure that they are done, you need:

```
>>> returncodeC = C.wait()
>>> returncodeP = P.wait()
```

Capturing the console output from Popen

So the output from C still goes to the console.

To capture it, we need to attach a PIPE to its output as well:

```
>>> import sys
>>> from subprocess import PIPE, Popen
>>> P = Popen([sys.executable,"-c","print('Hello')"], text=True, stdout=PIPE)
>>> C = Popen([sys.executable,"-c","print(input())"], text=True, stdin=P.stdout, stdout=PIPE)
>>>
```

Nothing seems to happen, because C's stdout is not attached to anything.

We can attach it back to python with its communicate method:

```
>>> result = C.communicate()
>>> print(result)
('Hello\n', None)
```

Note that communicate returns both the stdout and stderr.
(To actually get the latter, should add stderr=PIPE)

One more thing: shell=True?

- Many examples that you may find have the option "`shell=True`" to `.run()`.
- Instead of running the process directly, this calls the shell as an intermediate process, and runs a command in that shell.
- This can be useful if e.g., you need wildcards ('*', etc), or other constructs that only work in a shell.
- Unless you have a reason to use this, though, don't.

Back to BLAST+

How now can we get this to work:

```
>>> from Bio.Blast.Applications import NcbiblastnCommandline
>>> blastn_cmdline = NcbiblastnCommandline(query="dna.fasta", db="nt", evalue=0.001,
                                           outfmt=5, out="dna.xml")
>>> stdout, stderr = blastn_cmdline()
```

We have to build the command line ourselves, so we need to know the options and syntax of `blastn`.

We can get the syntax with the "`blastn -h`" command, which we can enter on the shell prompt, or. . .

Get syntax

```
>>> print(subprocess.check_output(["blastn","-h"],text=True))
USAGE
  blastn [-h] [-help] [-import_search_strategy filename]
    [-export_search_strategy filename] [-task task_name] [-db database_name]
    [-dbsize num_letters] [-gilist filename] [-seqidlist filename]
    [-negative_gilist filename] [-negative_seqidlist filename]
    [-taxids taxids] [-negative_taxids taxids] [-taxidlist filename]
    [-negative_taxidlist filename] [-no_taxid_expansion]
    [-entrez_query entrez_query] [-db_soft_mask filtering_algorithm]
    [-db_hard_mask filtering_algorithm] [-subject subject_input_file]
    [-subject_loc range] [-query input_file] [-out output_file]
    [-evalue evalue] [-word_size int_value] [-gapopen open_penalty]
    [-gapextend extend_penalty] [-perc_identity float_value]
    [-qcov_hsp_perc float_value] [-max_hsps int_value]
    [-xdrop_ungap float_value] [-xdrop_gap float_value]
    [-xdrop_gap_final float_value] [-searchsp int_value] [-penalty penalty]
    [-reward reward] [-no_greedy] [-min_raw_gapped_score int_value]
    [-template_type type] [-template_length int_value] [-dust DUST_options]
    [-filtering_db filtering_database]
    [-window_masker_taxid window_masker_taxid]
    [-window_masker_db window_masker_db] [-soft_masking soft_masking]
    [-ungapped] [-culling_limit int_value] [-best_hit_overhang float_value]
    [-best_hit_score_edge float_value] [-subject_besthit]
```

Back to BLAST+: Queries

How now can we get this to work:

```
>>> from Bio.Blast.Applications import NcbiblastnCommandline
>>> blastn_cmdline = NcbiblastnCommandline(query="dna.fasta", db="nt", evalue=0.001,
                                           outfmt=5, out="dna.xml")
>>> stdout, stderr = blastn_cmdline()
```

We have to build the command line ourselves, so we need to know the options and syntax of `blastn`.

From "`blastn -h`" command, we see that there are command line options for everything we want. There's a "`-query`" option, a "`-db`" option, and "`-evalue`" option. and "`-outfmt`" option, and a "`-out`" option.

So:

```
>>> our_cmdline = ["blastn", "-db", "nt", "-evalue", "0.001", "-outfmt", "5",
                   "-out","dna.xml", "-query", "dna.fasta"]
>>> result = subprocess.check_output(our_cmdline, text=True)
>>> # or: result = subprocess.run(our_cmdline, text=True).stdout
```

The real result is in `dna.xml`, to be read like we saw in the last lecture, using `xml.etree.ElementTree`.

Back to BLAST+: Creating a database

When working with our own database, we had to first index it.

Before, with BioPython, you could do this:

```
>>> from Bio.Blast.Applications import NcbimakeblastdbCommandline
>>> makeblastdb_cmdline = NcbimakeblastdbCommandline(input_file="my.fasta",out="chicken", dbtype="nucl")
>>> stdout, stderr = makeblastdb_cmdline()
```

Looking at the help in `makeblastdb -h`, we see that we need to do:

```
makeblastdb -in nt -out nt -dbtype nucl

>>> our_cmdline = ["makeblastdb", "-in","my.fasta", "-out", "chicken", "-dbtype","nucl"]
>>> result = subprocess.check_output(our_cmdline)
>>> print(result)
```