Writing Pipelines in Python

> Falling in love with code means falling in love with problem solving and being a part of a forever ongoing conversation. – Kathryn Barrett

You might be surprised at how far you can push humble `make` to write analysis pipelines. I'd encourage you to really explore Makefiles, reading the docs and looking at other people's examples. You'll save yourself many hours if you learn to use `make` well, even if you are just documenting how you ran your Python program. Beyond `make`, there are many other frameworks for writing pipelines such as Nextflow, Snakemake, Taverna, Pegasus and many more (cf https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5429012/), many of which are probably far superior to rolling your own in Python; however, we will do just that as you will learn many valuable skills along the way. After all, hubris is one of the three virtues of a great programmer:

> According to Larry Wall, the original author of the Perl programming language, there are three great virtues of a programmer:
>
> **Laziness**: The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful and document what you wrote so you don't have to answer so many questions about it.
>
> **Impatience**: The anger you feel when the computer is being lazy. This makes you write programs that don't just react to your needs, but actually anticipate them. Or at least pretend to.
>
> **Hubris**: The quality that makes you write (and maintain) programs that other people won't want to say bad things about.
>
> *Programming Perl*, 2nd Edition, O'Reilly & Associates, 1996

A "pipeline" is chaining the output of one program or function as the input to the next as many times as necessary to arrive at an end product. Sometimes the whole pipeline can be written inside Python, but often in bioinformatics what we have is one program written in Java/C/C++ we install from source that creates some output that needs to be massaged by a program we write in bash or Python that gets fed to a Perl script you found on BioStars that produces some text file that we read into R to create some visualization. We're going to focus on how to use Python to take input, call external programs, check on the status, and feed the output to some other program.

Hello

In this first example, we'll pretend this "hello.sh" is something more interesting than it really is:

```
$ cat -n hello.sh
     1	#!/usr/bin/env bash
     2	
     3	if [[ $# -lt 1 ]]; then
     4	    printf "Usage: %s NAME\n" $(basename $0)
     5	    exit 1
     6	fi
     7	
     8	NAME=$1
     9	
    10	if [[ $NAME == 'Lord Voldemort' ]]; then
    11	    echo "Upon advice of my counsel, I respectfully refuse to say that name."
    12	    exit 1
    13	fi
    14	
    15	echo "Hello, $1!"
$ ./hello.sh
Usage: hello.sh NAME
$ ./hello.sh Jan
Hello, Jan!
$ ./hello.sh "Lord Voldemort"
Upon advice of my counsel, I respectfully refuse to say that name.
```

We'll write a Python program to feed names to the "hello.sh" program and monitor whether the program ran successfully.

```
$ cat -n run_hello.py
     1	#!/usr/bin/env python3
     2	"""
     3	Author : kyclark
     4	Date   : 2019-03-28
     5	Purpose: Run "hello.sh"
     6	"""
     7	
     8	import argparse
     9	import os
    10	import sys
    11	from subprocess import getstatusoutput
    12	
    13	
    14	# --------------------------------------------------
    15	def get_args():
    16	    """get command-line arguments"""
    17	    parser = argparse.ArgumentParser(
    18	        description='Simple pipeline',
    19	        formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    20	
    parser.add_argument(
        'name', metavar='str', nargs='+', help='Names for hello.sh')

    parser.add_argument(
        '-p',
        '--program',
        help='Program to run',
        metavar='str',
        type=str,
        default='./hello.sh')

    return parser.parse_args()


# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""
    print(msg, file=sys.stderr)


# --------------------------------------------------
def die(msg='Something bad happened'):
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    prg = args.program

    if not os.path.isfile(prg):
        die('Missing expected program "{}"'.format(prg))

    for name in args.name:
        cmd = '{} "{}"'.format(prg, name)
        rv, out = getstatusoutput(cmd)
        if rv != 0:
            warn('Failed to run: {}\nError: {}'.format(cmd, out))
        else:
            print('Success: "{}"'.format(out))

    print('Done.')

```

```
67
68  # --------------------------------------------------
69  if __name__ == '__main__':
70      main()
```

In `get_args` we establish that we expect one or more positional arguments on the command line along with an optional `-p|--program` to run with those as arguments. One of the first items to check is if the `program` exists (we are expecting a full path with `./hello.sh` being the default), so line 54 checks this and calls `die` if it does not exist.

The main event starts on line 57 where we loop through the name arguments. On line 58, we create a command by making a string with the name of the program and the argument. Then we use `subprocess.getstatusoutput` to run this command and give us the return value (`rv`) and the output from the command (both STDERR and STDOUT get combined). If the return value is not zero ("zero errors"), then we use `warn` to report on STDERR that there was a failure, else we print "Success" along with the output from `hello.sh`.

If we run this, we see it stops when given a bad `program`:

```
$ ./run_hello.py -p foo Ken
Missing expected program "foo"
```

And we see it correctly reports the results for our inputs:

```
$ ./run_hello.py Jan Marcia "Lord Voldemort" Cindy
Success: "Hello, Jan!"
Success: "Hello, Marcia!"
Failed to run: ./hello.sh "Lord Voldemort"
Error: Upon advice of my counsel, I respectfully refuse to say that name.
Success: "Hello, Cindy!"
Done.
```

If you were submitting this job to run on an HPC, it would be launched by the job scheduler sometime later than when you submit it and would be run in an automated fashion. You would quickly learn that it's better to capture errors to an error file rather than let them comingle with STDOUT.

```
$ ./run_hello.py Jan Marcia "Lord Voldemort" Cindy 2>err
Success: "Hello, Jan!"
Success: "Hello, Marcia!"
Success: "Hello, Cindy!"
Done.
$ cat err
Failed to run: ./hello.sh "Lord Voldemort"
Error: Upon advice of my counsel, I respectfully refuse to say that name.
```

Parallel Hello

This works fairly well, but what if there are potentially dozens, hundreds, or thousands of names to greet? We are processing these in a serial fashion, but it's common that even laptops have more than one CPU that could we could use. Even with just 2 CPUs, we'd accomplish the task 2X faster than using just one. It's common to have 60-90 CPUs (or "cores") on HPC machines. If you aren't using them, you're wasting time!

The GNU `parallel` program (https://www.gnu.org/software/parallel/) provides a simple way to use more than one CPU to complete a batch of jobs. It takes as input the commands that need to be run and spins them out to all available CPUs (or as many as you limit it to), watching for jobs that fail, starting up new jobs when older ones finish.

To see it in action, let's compare these two programs in the "examples/gnu_parallel" directory. The first one simply prints the number 1-30 in order:

```
$ cat -n run.sh
     1	#!/usr/bin/env bash
     2
     3	for i in $(seq 1 30); do
     4	    echo $i
     5	done
     6
     7	echo "Done."
```

The second one uses `parallel` to print them. While this is a trivial case, imagine something more intense like BLAST jobs.

```
$ cat -n run_parallel.sh
     1	#!/usr/bin/env bash
     2
     3	JOBS=$(mktemp)
     4
     5	for i in $(seq 1 30); do
     6	    echo "echo $i" >> "$JOBS"
     7	done
     8
     9	NUM_JOBS=$(wc -l "$JOBS" | awk '{print $1}')
    10
    11	if [[ $NUM_JOBS -gt 0 ]]; then
    12	    echo "Running $NUM_JOBS jobs"
    13	    parallel -j 8 --halt soon,fail=1 < "$JOBS"
    14	fi
    15
    16	[[ -f "$JOBS" ]] && rm "$JOBS"
```

```

echo "Done."
```

And here is they look like when they are run:

```
$ ./run.sh
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
Done.
$ ./run_parallel.sh
Running 30 jobs
7
9
8
10
11
12
13
14
```

```
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
6
5
30
4
3
2
1
Done.
```

The `parallel` version looks out of order because the jobs are run as quickly as possible in whatever order that happens.