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
```

```
21      parser.add_argument(
22          'name', metavar='str', nargs='+', help='Names for hello.sh')
23
24      parser.add_argument(
25          '-p',
26          '--program',
27          help='Program to run',
28          metavar='str',
29          type=str,
30          default='./hello.sh')
31
32      return parser.parse_args()
33
34
35  # --------------------------------------------------
36  def warn(msg):
37      """Print a message to STDERR"""
38      print(msg, file=sys.stderr)
39
40
41  # --------------------------------------------------
42  def die(msg='Something bad happened'):
43      """warn() and exit with error"""
44      warn(msg)
45      sys.exit(1)
46
47
48  # --------------------------------------------------
49  def main():
50      """Make a jazz noise here"""
51      args = get_args()
52      prg = args.program
53
54      if not os.path.isfile(prg):
55          die('Missing expected program "{}"'.format(prg))
56
57      for name in args.name:
58          cmd = '{} "{}"'.format(prg, name)
59          rv, out = getstatusoutput(cmd)
60          if rv != 0:
61              warn('Failed to run: {}\nError: {}'.format(cmd, out))
62          else:
63              print('Success: "{}"'.format(out))
64
65      print('Done.')
66
```

```
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"
```

```
    17
    18  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.

CD-HIT

Let's take the `cd-hit` cluster exercise and extend it to where we take the proteins FASTA, run cd-hit, and find the unclustered proteins all in one go. First things first, we need to ensure `cd-hit` is on our system. It's highly unlikely that it is, so let's figure out how to install it.

If you search on the Internet for `cd-hit`, you might end up at http://weizhongli-lab.org/cd-hit/ from which you go to the download page (http://weizhongli-lab.org/cd-hit/download.php) which directs you to the GitHub releases for the `cd-hit` repository (https://github.com/weizhongli/cdhit/releases). From there, we can download the source code tarball (`.tar.gz` file). For instance, I right-click on the link to copy the line address, then go to my HPC into my "downloads" directory and then use `wget` to retrieve the tarball. Next use `tar xvf` to "extract" in a "verbose" fashion the "file" (followed by the tarball). Finally you should have a directory like `cd-hit-v4.8.1-2019-0228` into which you should `cd`.

If you look at the README, you'll see the way to compile this is to just type

`make`. On my Mac laptop, I needed to compile without multi-threading support, so I used `make openmp=no`. That will run for a few seconds and look something like this:

```
$ make openmp=no
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-common.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-utility.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit.o cdhit-common.o cdhit-utility.o -lz -o cd-hit
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-est.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-est.o cdhit-common.o cdhit-utility.o -lz -o cd-hit-e
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-2d.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-2d.o cdhit-common.o cdhit-utility.o -lz -o cd-hit-2d
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-est-2d.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-est-2d.o cdhit-common.o cdhit-utility.o -lz -o cd-hi
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-div.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-div.o cdhit-common.o cdhit-utility.o -lz -o cd-hit-d
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-454.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-454.o cdhit-common.o cdhit-utility.o -lz -o cd-hit-4
```

Often Makefiles will include an `install` target that will copy the new programs into a directory like `/usr/local/bin`. This one does not, so you'll have to manually copy the programs (e.g., `cd-hit`, `cd-hit-2d`, etc.) to whatever location you like. On an HPC (like Ocelote), you will not have permissions to copy to `/usr/local/bin`, so I'd recommend you create a directory like `$HOME/.local/bin` which you add to your `$PATH` and copy the binaries to that location.

Ensure you have a `cd-hit` binary you can use:

```
$ which cd-hit
/Users/kyclark/.local/bin/cd-hit
$ cd-hit -h | head
        ====== CD-HIT version 4.8.1 (built on Apr  9 2019) ======

Usage: cd-hit [Options]

Options

   -i   input filename in fasta format, required, can be in .gz format
   -o   output filename, required
   -c   sequence identity threshold, default 0.9
    this is the default cd-hit's "global sequence identity" calculated as:
```

Now we can try out our new code:

```
$ cat -n cdhit_unclustered.py
     1  #!/usr/bin/env python3
     2  """
```

```
 3  Author : kyclark
 4  Date   : 2019-02-20
 5  Purpose: Run cd-hit, find unclustered proteins
 6  """
 7
 8  import argparse
 9  import datetime
10  import logging
11  import os
12  import re
13  import signal
14  import sys
15  from subprocess import getstatusoutput
16  from shutil import which
17  from Bio import SeqIO
18
19
20  # --------------------------------------------------
21  def get_args():
22      """get command-line arguments"""
23      parser = argparse.ArgumentParser(
24          description='Run cd-hit, find unclustered proteins',
25          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
26
27      parser.add_argument(
28          '-p',
29          '--proteins',
30          help='Proteins FASTA',
31          metavar='str',
32          type=str,
33          required=True)
34
35      parser.add_argument(
36          '-c',
37          '--seq_id_threshold',
38          help='cd-hit Sequence identity threshold',
39          metavar='float',
40          type=float,
41          default=0.9)
42
43      parser.add_argument(
44          '-o',
45          '--outfile',
46          help='Output file',
47          metavar='str',
48          type=str,
```

```
49          default='unclustered.fa')
50
51      parser.add_argument(
52          '-l',
53          '--logfile',
54          help='Log file',
55          metavar='str',
56          type=str,
57          default='.log')
58
59      parser.add_argument('-d', '--debug', help='Debug', action='store_true')
60
61      return parser.parse_args()
62
63
64  # --------------------------------------------------
65  def die(msg='Something bad happened'):
66      """log a critical message() and exit with error"""
67      logging.critical(msg)
68      sys.exit(1)
69
70
71  # --------------------------------------------------
72  def run_cdhit(proteins_file, seq_id_threshold):
73      """Run cd-hit"""
74      cdhit = which('cd-hit')
75
76      if not cdhit:
77          die('Cannot find "cd-hit"')
78
79      out_file = os.path.basename(proteins_file) + '.cdhit'
80      out_path = os.path.join(os.path.dirname(proteins_file), out_file)
81
82      logging.debug('Found cd-hit "{}"'.format(cdhit))
83      cmd = '{} -c {} -i {} -o {} -d 0'.format(cdhit, seq_id_threshold,
84                                               proteins_file, out_file)
85      logging.debug('Running "{}"'.format(cmd))
86      rv, out = getstatusoutput(cmd)
87
88      if rv != 0:
89          die('Non-zero ({}) return from "{}"\n{}\n'.format(rv, cmd, out))
90
91      if not os.path.isfile(out_file):
92          die('Failed to create "{}"'.format(out_file))
93
94      logging.debug('Finished cd-hit, found cluster file "{}"'.format(out_file))
```

```
 95
 96      return out_file
 97
 98
 99  # --------------------------------------------------
100  def get_unclustered(cluster_file, proteins_file, out_file):
101      """Find the unclustered proteins in the cd-hit output"""
102
103      if not os.path.isfile(cluster_file):
104          die('cdhit "{}" is not a file'.format(cluster_file))
105
106      logging.debug('Parsing "{}"'.format(cluster_file))
107
108      clustered = set([rec.id for rec in SeqIO.parse(cluster_file, 'fasta')])
109
110      # Alternate (longer) way:
111      # clustered = set()
112      # for rec in SeqIO.parse(cluster_file, 'fasta'):
113      #     clustered.add(rec.id)
114
115      logging.debug('Will write to "{}"'.format(out_file))
116      out_fh = open(out_file, 'wt')
117      num_total = 0
118      num_unclustered = 0
119
120      for rec in SeqIO.parse(proteins_file, 'fasta'):
121          num_total += 1
122          prot_id = re.sub(r'\|.*', '', rec.id)
123          if not prot_id in clustered:
124              num_unclustered += 1
125              SeqIO.write(rec, out_fh, 'fasta')
126
127      logging.debug(
128          'Finished writing unclustered proteins'.format(num_unclustered))
129
130      return (num_unclustered, num_total)
131
132
133  # --------------------------------------------------
134  def main():
135      """Make a jazz noise here"""
136      args = get_args()
137      proteins_file = args.proteins
138      out_file = args.outfile
139      log_file = args.logfile
140
```

```
141     if not os.path.isfile(proteins_file):
142         die('--proteins "{}" is not a file'.format(arg_name, proteins_file))
143 
144     logging.basicConfig(
145         filename=log_file,
146         filemode='a',
147         level=logging.DEBUG if args.debug else logging.CRITICAL)
148 
149     def sigint(sig, frame):
150         logging.critical('INT: Exiting early!')
151         sys.exit(0)
152 
153     signal.signal(signal.SIGINT, sigint)
154 
155     banner = '#' * 50
156     logging.debug(banner)
157     logging.debug('BEGAN {}'.format(str(datetime.datetime.today())))
158 
159     cluster_file = run_cdhit(proteins_file, args.seq_id_threshold)
160     num_unclustered, num_total = get_unclustered(cluster_file, proteins_file,
161                                                  out_file)
162 
163     msg = 'Wrote {:,d} of {:,d} unclustered proteins to "{}"'.format(
164         num_unclustered, num_total, out_file)
165 
166     print(msg)
167     logging.debug(msg)
168     logging.debug('FINISHED {}'.format(str(datetime.datetime.today())))
169     logging.debug(banner)
170 
171 
172 # --------------------------------------------------
173 if __name__ == '__main__':
174     main()
```