Generating commands and jobs in Python

HORT 530

Lecture/Lab 13

Instructor: Kranthi Varala

When to use supercomputers

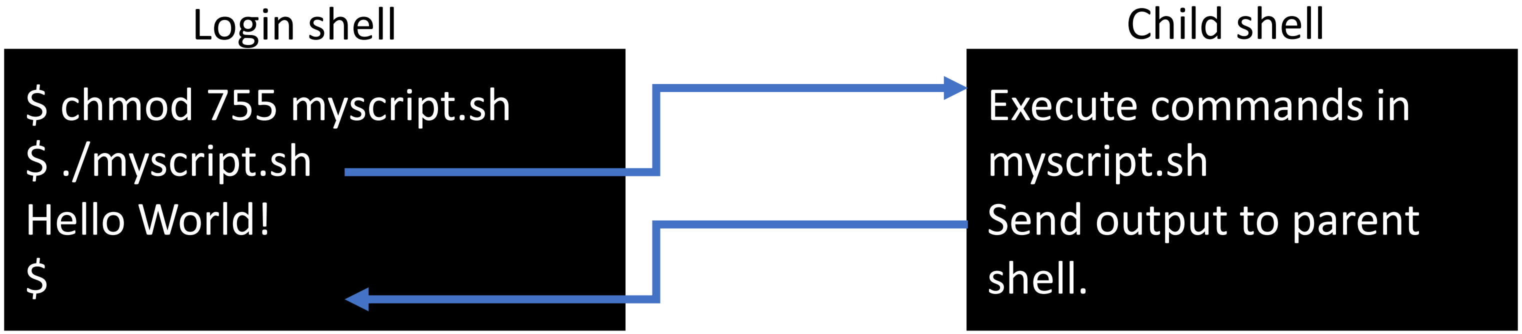
- Need to run hundreds to thousands of similar jobs.
- Need to run a few large jobs quickly.
- Tasks can be divided into smaller portions and run in parallel.

Parallelization

- Refers to the ability of dividing a large task into smaller parts that can all be run in parallel.
- E.g., Correlation matrix of 10,000 genes.
- Can be divided into 10,000 jobs where each job works on one gene.

Script creates its own Shell

- A script is always executed in its own shell, i.e., when you execute a shell script it starts a new child shell within the shell you executed the program from.



Login shell

```
$ chmod 755 myscript.sh
$ ./myscript.sh
Hello World!
$
```

Child shell

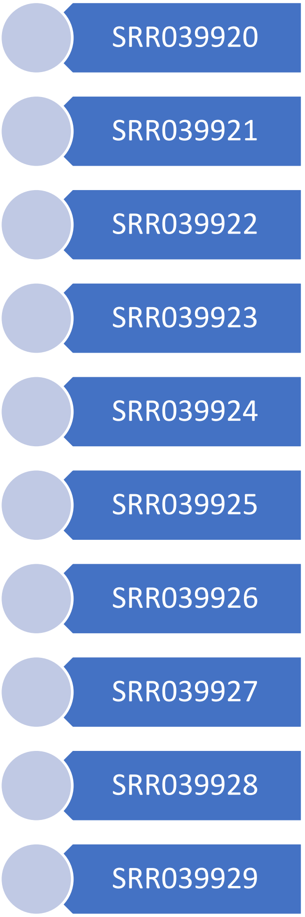
Execute commands in myscript.sh

Send output to parent shell.

Task: Download all sequencing runs from GSE18110

- https://www.ncbi.nlm.nih.gov/Traces/study/?acc=SRP002313
- Download the "Accession List"
- Use scp to copy file over to scholar.rcac.purdue.edu
- This file contains the list of SRR IDs.
- The command to download the results from a sequencing result is:
  - `fastq-dump SRR039929`
- This command needs the bioinfo and sra-toolkit modules.

Generating UNIX commands from Python



SRR039920

SRR039921

SRR039922

SRR039923

SRR039924

SRR039925

SRR039926

SRR039927

SRR039928

SRR039929

Option 1	Option 2	Option 3
Run each download command individually.	Generate commands in Python.	Use Python to generate multiple SLURM scripts.
10 jobs have to be created manually.	1 script can run jobs sequentially.	Each SLURM job runs in background.
No record of parameters give to command.	Script keeps track of parameters given to command.	Script keeps track of parameters given to command.
Run time for commands is cumulative and needs user attention.	Run time for commands is cumulative and DOESN'T need user attention.	Run time for commands will be <= cumulative and DOESN'T need user attention.

Controlling processes from command line

- Foreground: Default mode for running commands. The shell waits on the process to finish.
  - Process retains control of the command line.
  - Key input is directed to the active process.
- Background: Process is initiated and pushed to the background.
  - Control of command-line is returned to the user.
  - Key input and other interactions are no longer passed to the process.
  - Processes can be pushed to background at initiation using &

Controlling processes from Python

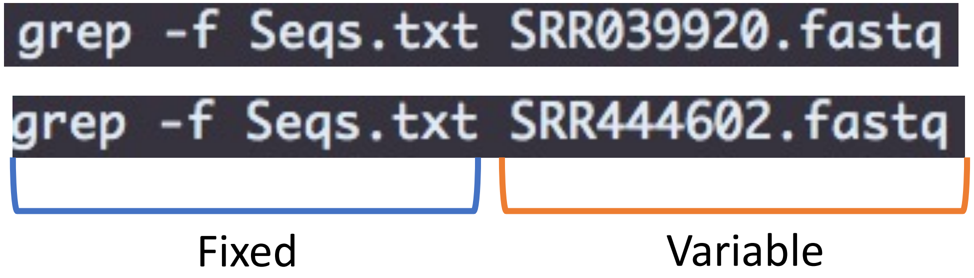
- Wait: Default mode for running commands. The python interpreter waits on the process to finish.
    - Process retains control of the shell.
    - Both os.system() and subprocess.call() do this by default
- Submit and forget: Command is initiated in a separate process.
    - Processes can be pushed to background at initiation using &
    - subprocess.Popen(), by default, does not wait for child process to finish

Creating command strings vs. scripts

- Command strings: Use python to generate the command string with a combination of fixed strings and variables.
  - Submit command using os.system() or subprocess.call()
  - Submit command using subprocess.Popen() if you need to capture output.
- Scripts: Use file handle object to create a new script file with commands and parameters embedded.
  - subprocess.Popen() to submit the script as a job
  - Remember to make shell scripts executable (chmod 755)
  - SLURM scripts need not be executable
- When submitting multiple jobs, creating scripts helps keep track of the exact command and parameters used.

Generating UNIX commands from Python

- Python can be used to generate repetitive UNIX commands that operate over multiples of a set
  - For example, find a given sequence in all fastq files
- Key: A UNIX command is a string with fixed words, such as command name, and variable words, such as name of input file(s)



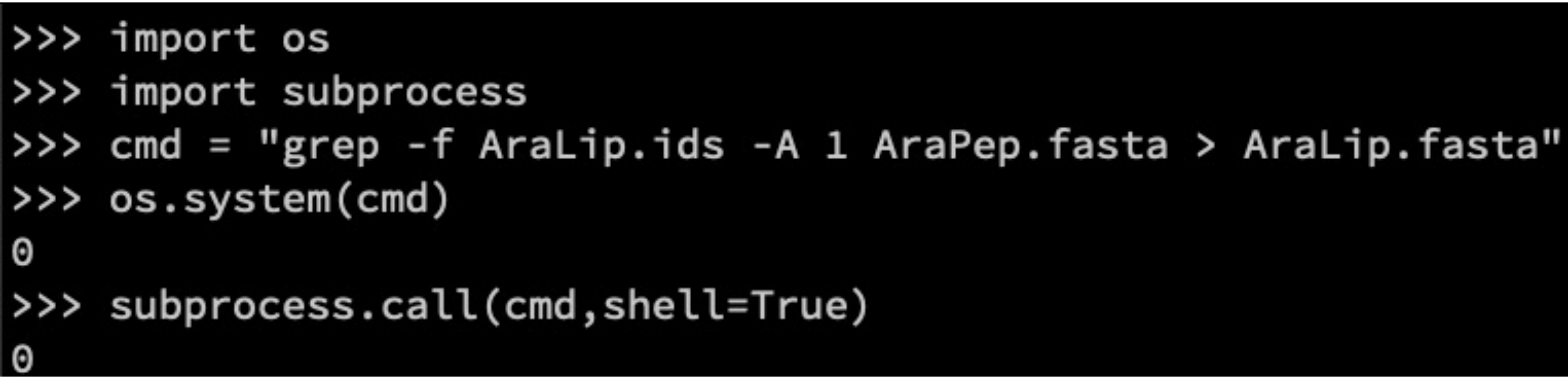
```
grep -f Seqs.txt SRR039920.fastq
```

```
grep -f Seqs.txt SRR444602.fastq
```

Making system calls from python

- We can use python to make calls to the system i.e., call commands and scripts available on the system command line.
- The 'os' and 'subprocess' module are the two main ways to interact with the system command line.
- The 'os' module is "deprecated", which means it's the old way of doing things and will not be supported in the future.

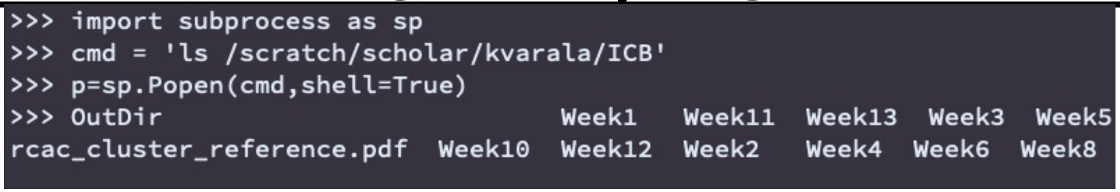
Running UNIX commands from Python



```
>>> import os
>>> import subprocess
>>> cmd = "grep -f AraLip.ids -A 1 AraPep.fasta > AraLip.fasta"
>>> os.system(cmd)
0
>>> subprocess.call(cmd,shell=True)
0
```

- os.system & subprocess.call() sent the 'cmd' command to the operating system.
- 'cmd' was run on the operating system and its output was dumped to the screen.
- Result: A new file called AraLip.fasta was created on the file system in the same directory

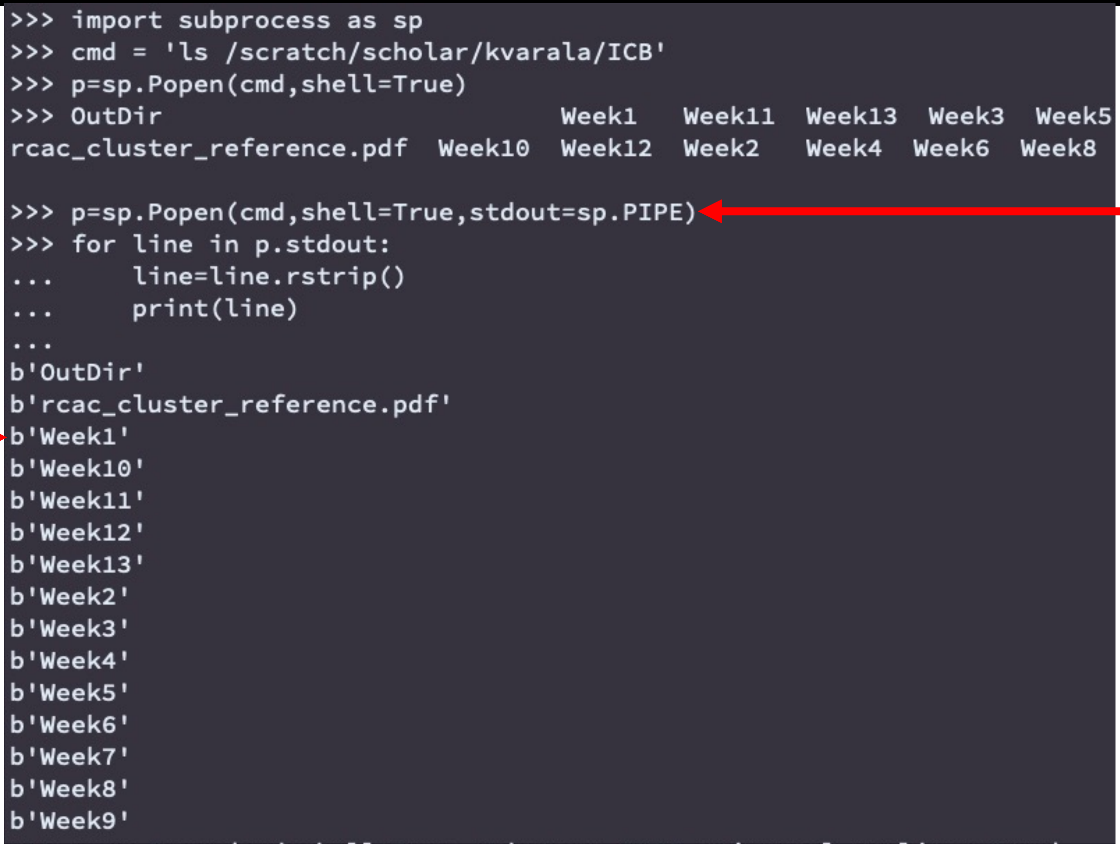
Capturing output of UNIX commands from Python



```
>>> import subprocess as sp
>>> cmd = 'ls /scratch/scholar/kvarala/ICB'
>>> p=sp.Popen(cmd,shell=True)
>>> OutDir                          Week1   Week11  Week13  Week3  Week5
rcac_cluster_reference.pdf  Week10  Week12  Week2   Week4  Week6  Week8
```

Output is not captured in 'p'

Capturing output of UNIX commands from Python

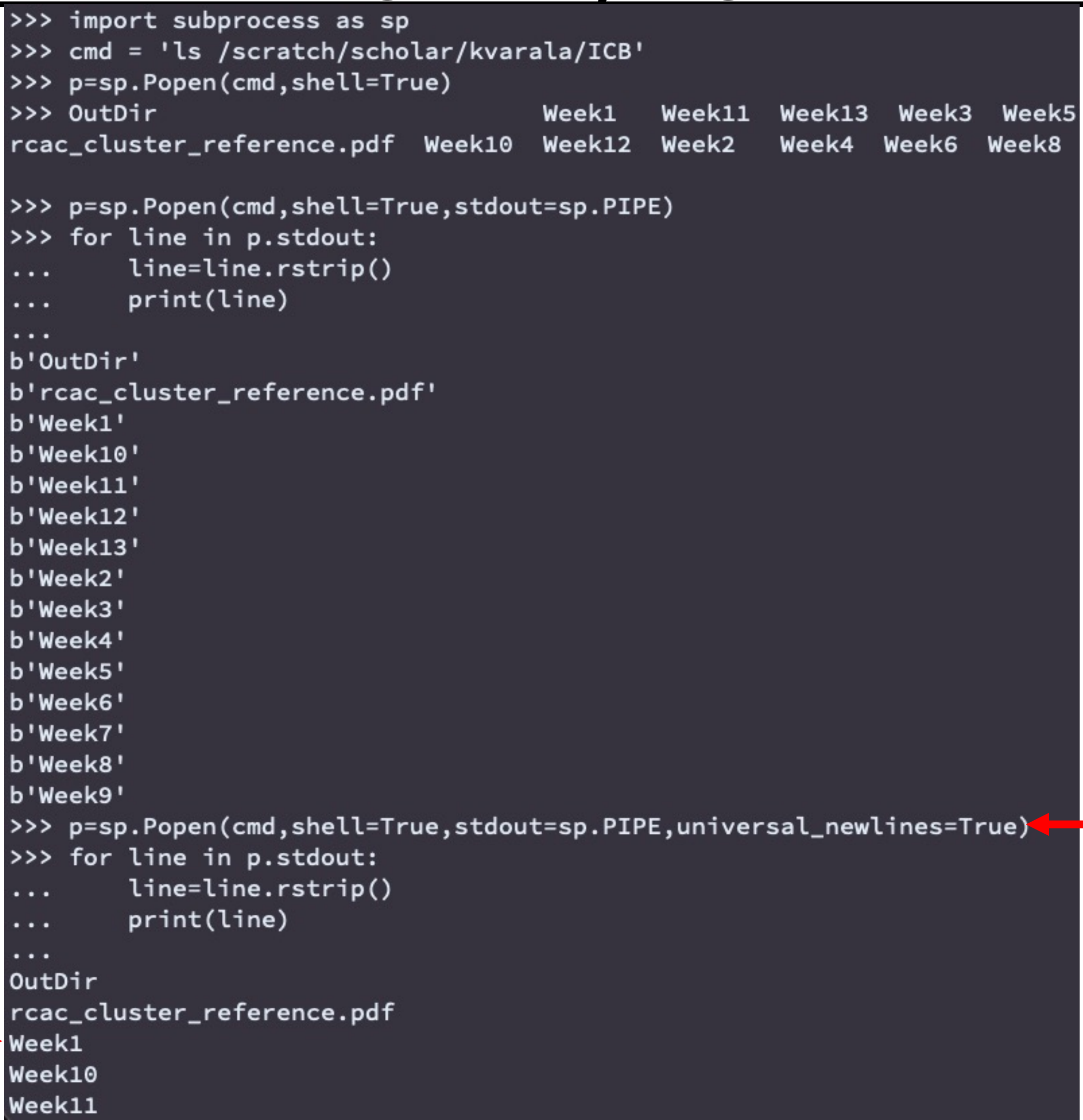


```
>>> import subprocess as sp
>>> cmd = 'ls /scratch/scholar/kvarala/ICB'
>>> p=sp.Popen(cmd,shell=True)
>>> OutDir                          Week1   Week11  Week13  Week3  Week5
rcac_cluster_reference.pdf  Week10  Week12  Week2   Week4  Week6  Week8

>>> p=sp.Popen(cmd,shell=True,stdout=sp.PIPE)
>>> for line in p.stdout:
...     line=line.rstrip()
...     print(line)
...
b'OutDir'
b'rcac_cluster_reference.pdf'
b'Week1'
b'Week10'
b'Week11'
b'Week12'
b'Week13'
b'Week2'
b'Week3'
b'Week4'
b'Week5'
b'Week6'
b'Week7'
b'Week8'
b'Week9'
```

Output is captured in 'p'

Extra characters from new line character

Capturing output of UNIX commands from Python



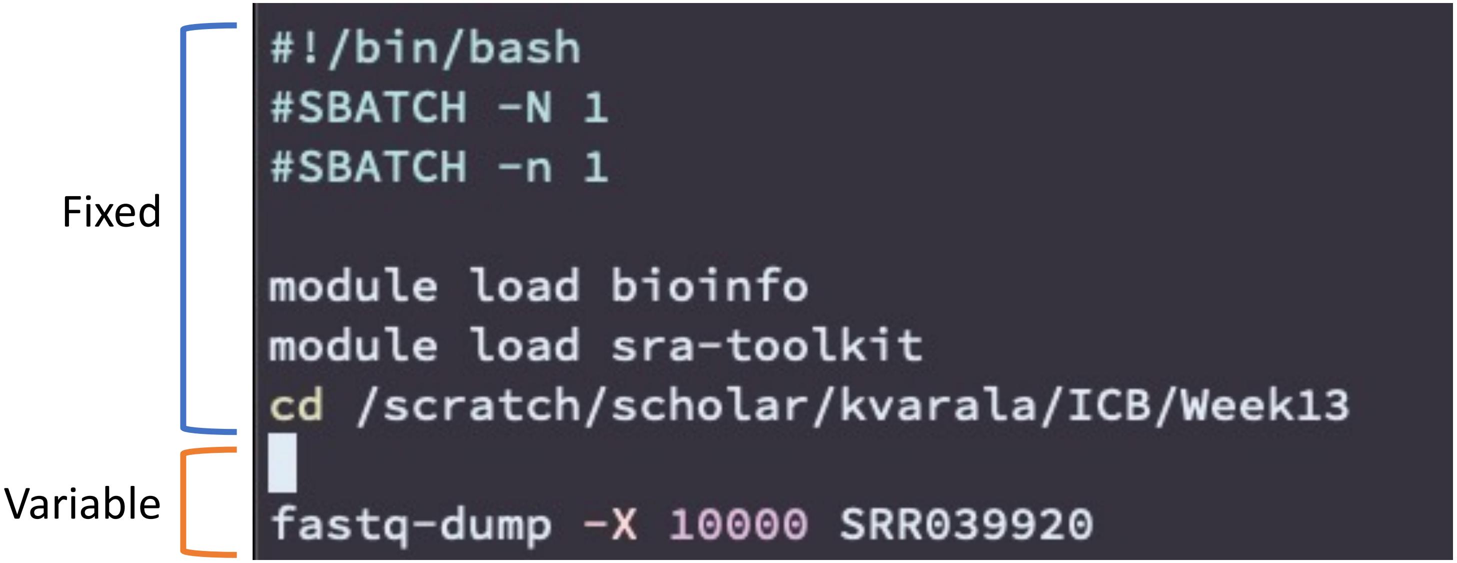
```
>>> import subprocess as sp
>>> cmd = 'ls /scratch/scholar/kvarala/ICB'
>>> p=sp.Popen(cmd,shell=True)
>>> OutDir                          Week1   Week11  Week13  Week3   Week5
rcac_cluster_reference.pdf  Week10  Week12  Week2   Week4  Week6  Week8

>>> p=sp.Popen(cmd,shell=True,stdout=sp.PIPE)
>>> for line in p.stdout:
...     line=line.rstrip()
...     print(line)
...
b'OutDir'
b'rcac_cluster_reference.pdf'
b'Week1'
b'Week10'
b'Week11'
b'Week12'
b'Week13'
b'Week2'
b'Week3'
b'Week4'
b'Week5'
b'Week6'
b'Week7'
b'Week8'
b'Week9'
>>> p=sp.Popen(cmd,shell=True,stdout=sp.PIPE,universal_newlines=True)
>>> for line in p.stdout:
...     line=line.rstrip()
...     print(line)
...
OutDir
rcac_cluster_reference.pdf
Week1
Week10
Week11
```

Output is captured in 'p'

Newline processed

Generating shell scripts from Python

- Python can be used to generate shell scripts that differ in few parameters
  - For example, SLURM scripts with different job parameters
- Key: A job script is made of fixed lines, such as job parameters, module loads etc. and variable lines, such as the lines specifying the input line.



```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 1

module load bioinfo
module load sra-toolkit
cd /scratch/scholar/kvarala/ICB/Week13

fastq-dump -X 10000 SRR039920
```

Generating shell scripts from Python

- Example 1: Write a python script that creates one SLURM job to fastq-dump the series of SRR IDs

- Example 2: Write a python script that creates one SLURM job for each SRR ID to fastq-dump the data