Programming and Data Science for Biology (PDSB)

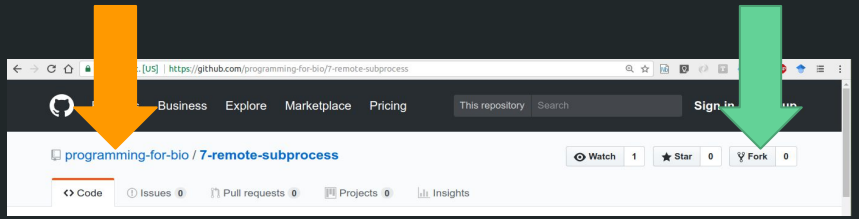
Session 7
Spring 2018

Working with git

1. Fork the upstream repo (a repo on the course GitHub account).

This creates a **new repo on your GitHub account** that is a clone of the original.

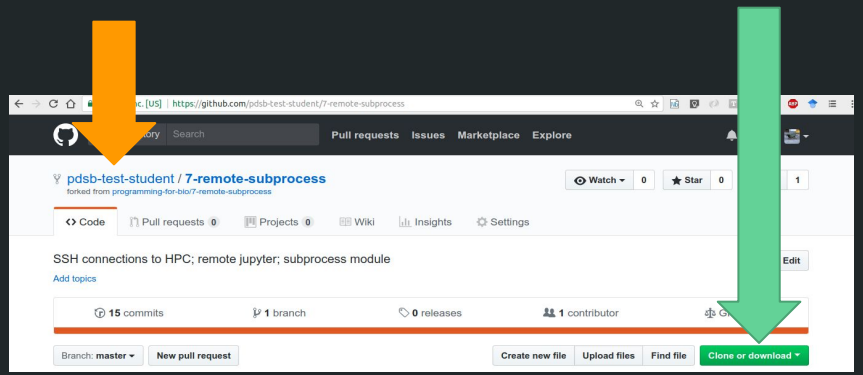
1. Fork the class repo (programming-for-bio)



Working with git

2. Clone your new repo so that you have a working copy on the computer that you are working on. Now you can make changes to it, add, commit, and push those changes and they will be synced back to your GitHub version.

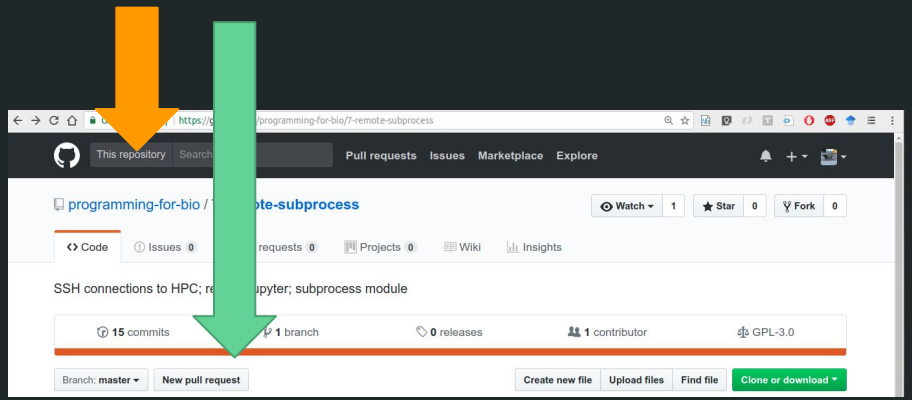
2. Clone your repo (your username)



Working with git

3. Pull request
This will ask the original (upstream) account to merge your changes with it. You should only have made changes to the requested files.

2. Make a pull request (programming-for-bio)



Python Classes: self?

Python Classes

Object oriented programming.

Common questions:

- What is self?
- What is __init__?
- What is being returned or stored?

```
## a simple class with an init function
class Simple:
     def __init__(self, name):
          self.name = name

## an instance of the class object
x = Simple('deren')
x.name
deren
```

Python Classes

Self is an *instance* of the class

```
## a simple class with an init function
class Simple:
     def __init__(self, name):
          self.name = name

## self is an instance of the class
self = Simple('deren')
self.name
deren
```

Design

import seqlib	(needs setup.py)
s = seqlib.Seqlib(10, 100)	(init func)
s.maf	(is an attribute)
s.calculate_statistics()	(is a function)
s.filter()	(is a function)

```
## init function
class Seqlib:
     def __init__(self, ninds, nsites):
          self.ninds = ninds
          self.nsites = nsites

## an instance of the seqlib class
self = Seqlib(10, 100)
self.ninds
10
```

Design

import seqlib	(needs setup.py)
s = seqlib.Seqlib(10, 100)	(init func)
s.maf	**(maf attribute)**
s.calculate_statistics()	(function)
s.filter()	(function)

```
## init function
class Seqlib:
     def __init__(self, ninds, nsites):
          self.ninds = ninds
          self.nsites = nsites
          self.maf = self._get_maf()

     def _get_maf(self):
          ...

## an instance of the seqlib class
self = Seqlib(10, 100)
self.maf
array([0.0, 0.1, 0.1, 0.5, 0.0, ...
```

Design

The design requested that we be able to see the minor allele frequency of the initiated sequence array, so that means we should probably calculate it in `__init__()`.

The `._get_maf()` function could be very simple like our example in the notebook, however, for it to work correctly we needed it to skip sites that are “N” (missing data). I hinted at this by asking *does the order of filters matter?*

```
## init function
class Seqlib:
     def __init__(self, ninds, nsites):
          self.ninds = ninds
          self.nsites = nsites
          self.maf = self._get_maf()

     def _get_maf(self):
          ...

## an instance of the seqlib class
self = Seqlib(10, 100)
self.maf
array([0.0, 0.1, 0.1, 0.5, 0.0, ...
```

Design

This is just *one* way to write this function. For example, we could have used **`set()`** to find the two most common alleles in a column and their frequency.

Each step asks: *what part of this array do I need to view, and which do I need to hide?*

```
def _get_maf(self):
    "returns the maf of the full seqarray while not counting Ns"
    ## init an array to fill and iterate over columns
    maf = np.zeros(self.nsites)
    for col in range(self.nsites):
        ## select this column of bases
        thiscol = self.seqs[:, col]

        ## mask "N" bases and get new length
        nmask = thiscol != "N"
        no_n_len = np.sum(nmask)

        ## mask "N" bases and get the first base
        first_non_n_base = thiscol[nmask][0]

        ## calculate maf of "N" masked bases
        freq = np.sum(thiscol[nmask] != first_non_n_base) / no_n_len
        if freq > 0.5:
            maf[col] = 1 - freq
        else:
            maf[col] = freq
    return maf
```

Design

One of the requested outputs was *very hard* to actually produce, since the example code returned a **ndarray**, while the requested function returned a modified **seqlib class object.**

I cheated / wrote a separate function to do this: `.filter_seqlib()`

We needed it to return a copy of the seqlib object in which all of its objects are also copied (e.g., its seqlib ndarray). The function **copy.deepcopy()** from the standard library can do this.

```
import copy

class Seqlib:
     def __init__(self, ninds, nsites):
          self.ninds = ninds
          self.nsites = nsites
          self.maf = self._get_maf()

     def filter_seqlib(self):
          newself = copy.deepcopy(self)
          ...
          return newself

## an instance of the seqlib class
self = Seqlib(10, 100)
self.filter_seqlib.calculate_statistics()
```

Design

Installing so the package is importable (setup.py)

```
project/
  +   setup.py
  +   package/
        +   __init__.py
        +   seqlib.py
```

We didn't need a __main__.py file because we were not writing a CLI. In fact, __main__.py is not a special filename the way that __init__.py is, and when writing a CLI we could name this file whatever we want as long as in setup.py we point to it in the `console_scripts` section.

```
#!/usr/bin/env python

# from package/seqlib.py import Seqlib
from seqlib.seqlib import Seqlib

-----------------------------------

# from the directory with setup.py
cd project/
pip install .
```

Secure Shell (SSH): secure remote login. Cluster computing. Linux.

Remote access by secure shell

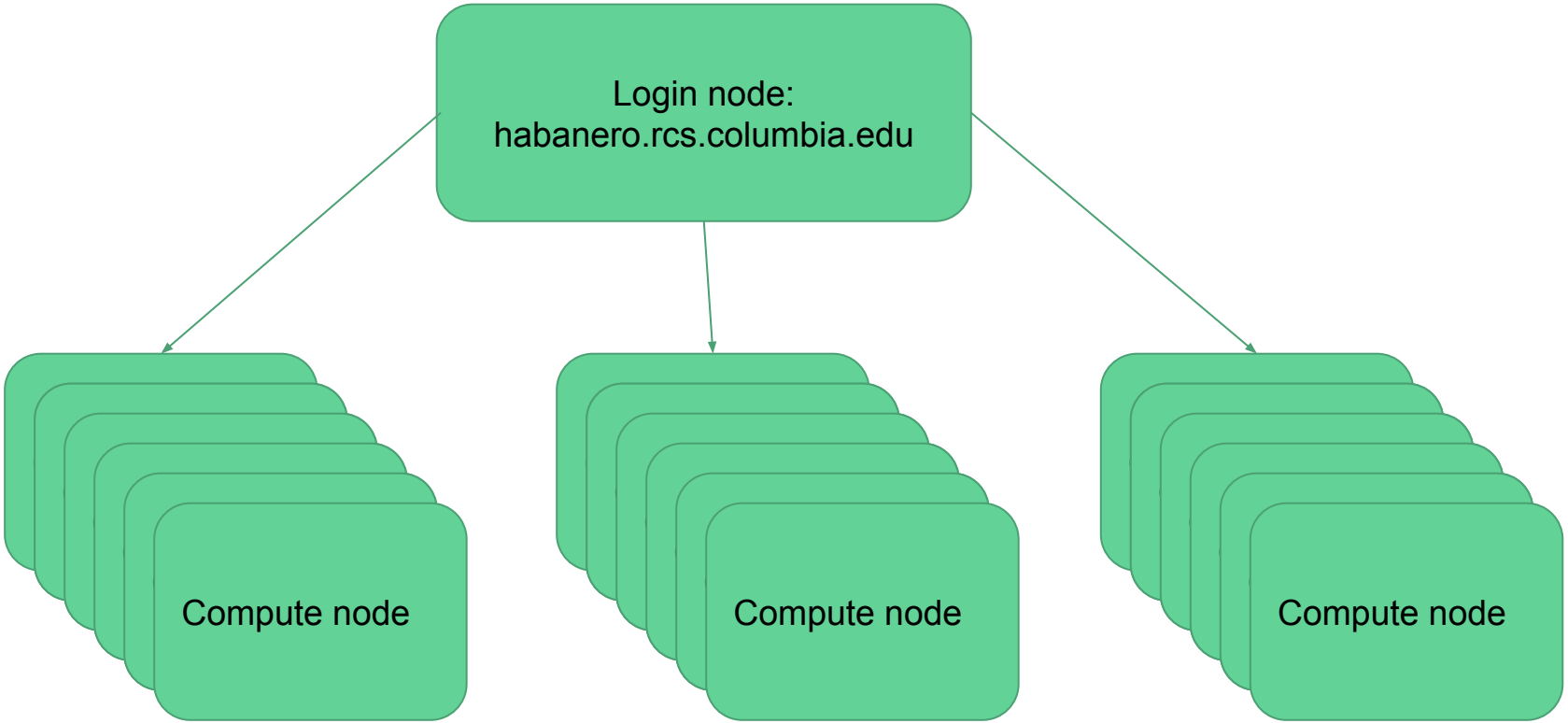
Stores a public/private key pair. By default the cluster server will authenticate your key using your UNI password.

```
## A server has an IP address
232.195.43.2.2

## And can also have a hostname
habanero.rcs.columbia.edu
```

Remote access by secure shell

Stores a public/private key pair. By default the cluster server will authenticate your key using your UNI password.

```
## format of ssh login command
ssh <user>@<hostname>

## our example
ssh de2356@habanero.rcs.columbia.edu
```


Login node:
habanero.rcs.columbia.edu
Compute node
Compute node
Compute node

High Performance Computing clusters

Remotely accessible servers for a connected array of computers.

+ Huge computing power (1000’s CPUs)
+ Large memory available (Gbs to Tbs)
+ Large disk storage (Tbs)
+ Managed/shared resource use

The Habanero cluster at Columbia

Total resources available

+ >5,000 CPUs
+ Each node has 128Gb RAM (or 512)
+ Each user has 1 Tb disk storage

The Habanero cluster at Columbia

Node structure (and reserved nodes)

+ 5,328 CPUs (222 nodes of 24 cores)
+ 128Gb RAM (176 nodes) or 512 (32 node)
+ 1 Tb disk storage

```
## A server has an IP address
232.195.43.2.2

## And can also have a hostname
habanero.rcs.columbia.edu
```

The Habanero cluster at Columbia

Because most nodes are reserved you often have a wait time until your job reaches the top of the queue of available resources. If you use more resources your next job will wait even longer.

```
## See the full queue of jobs
> squeue

## See just your jobs on the queue
> squeue -u de2356
```

SSH tutorial

Open on GitHub to view the notebook
`Notebooks/7.1-ssh.ipynb`

We won't be executing code *in* this notebook, just copying and pasting into a separate terminal.

subprocess

In: `Notebooks/7.2-subprocess.ipynb`

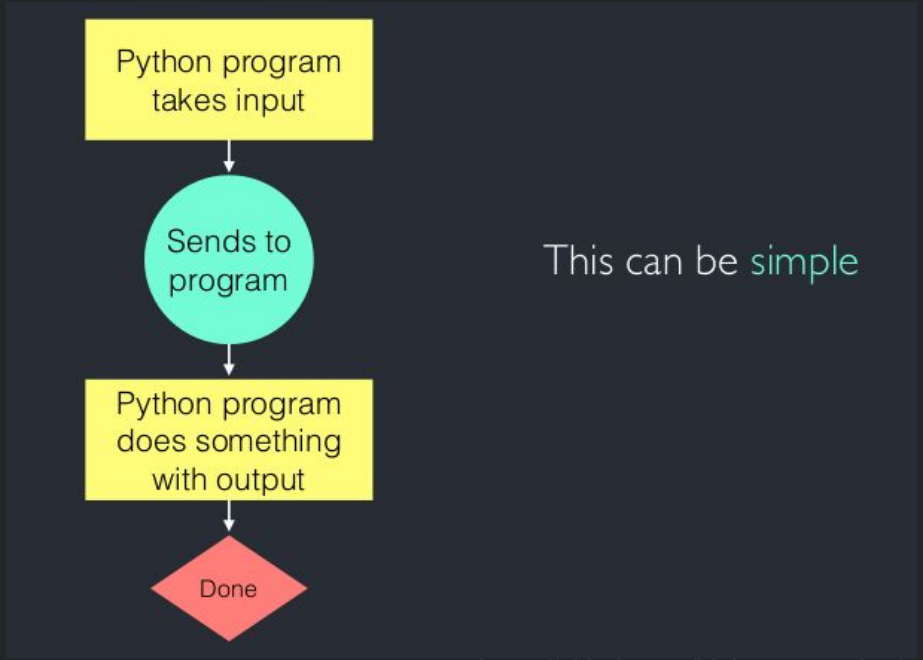
So far, we've been writing code to accomplish some task. But often, *there is already some program written to accomplish that task.*

```
## e.g., external genomics programs
bwa
abyss
vsearch
bowtie
muscle
samtools
```

subprocess

In: `Notebooks/7.2-subprocess.ipynb`

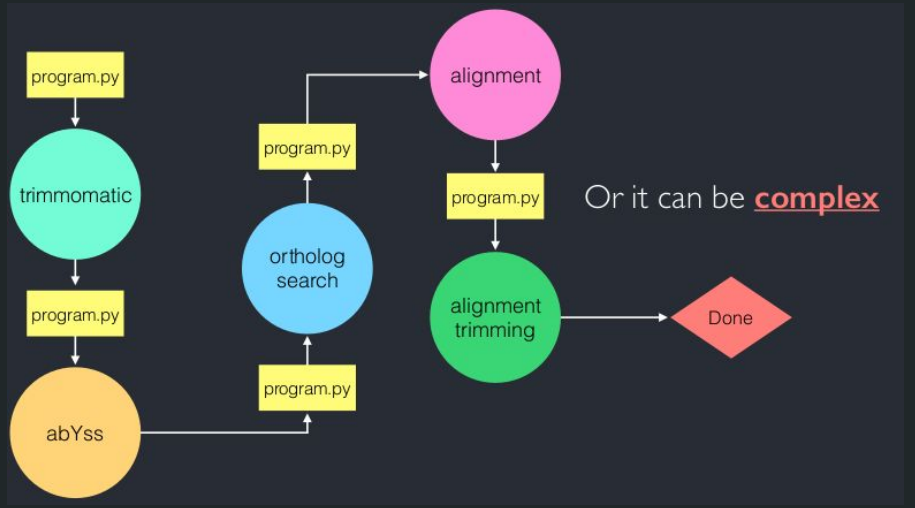
A python tool for calling other programs.



subprocess

In: `Notebooks/7.2-subprocess.ipynb`

A python tool for calling other programs.

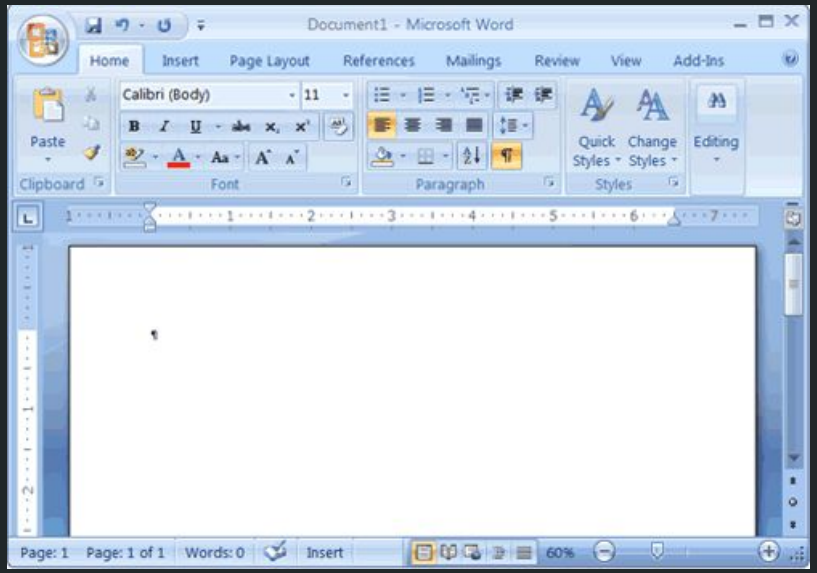


https://github.com/biolprogramming/

subprocess

Works for pretty much any scriptable program (i.e., a program with a CLI not a GUI).

Some GUI programs you are familiar with may have CLI options as well, e.g., ArcGIS or Arlequin. However, if they have a Python API then that's even better!



stdin, stdout, stderr

When you run a program it branches outputs into two different channels.

Good outputs go to stdout
Error outputs go to stderr

```
## bash code with result > stdout
echo “hello world” | cut -b 1-5

## bash code with error > stderr
echo “hello world” | cut -b 1:5

## explicit redirects
echo “hello world” | cut -b 1:5 1> file1
echo “hello world” | cut -b 1:5 2> file2

## separate redirects for error and results
echo “hello world” | cut -b 1-5 2> file2 1> file1

## some programs write logs to stderr
someprogram 2> logfile 1> resultfile
```

stdin, stdout, stderr

Calls to subprocess return separate outputs from stderr and stdout.

```
## Standard subprocess call
import subprocess as sps

## tell is where to write stdout (here to PIPE)
proc = sps.Popen([‘ls’, ‘-l’], stdout=sps.PIPE)

## PIPE connects stdout to the proc object
print(proc.stdout.decode())
```

Installing software on the remote cluster

Although up til this point we've gotten by fine using Linux, OSX, or windows, we will start to run into problems for windows when writing pipeline scripts.

This is because a lot of scientific software is written for unix-like systems, and so there is no windows version.

```
## conda installation of third party binaries
conda install muscle -c bioconda
conda install raxml -c bioconda
```

Python as glue

Pipelines are one of the most powerful uses of Python, and the ease with which Python can create pipelines very much makes up for the *relatively* slow speed of Python in comparison to languages like C++ or Fortran.

For many tasks Python is good enough, and has the advantage of being readable and easy to use.

Python remotely

One of the best new uses of Python is to work interactively with jupyter, and to do remotely

```
## running jupyter on HPC
jupyter notebook password
jupyter notebook --ip=$(hostname -i) --port=8888
```

Assignments and readings

Assignment is due Friday at 5pm

Code review is due Monday by class.

Assignment: Link to Session 7 repo

Readings: See syllabus

Collaborate: Work together in this gitter chatroom