Programming and Data Science for Biology (PDSB)

Session 3
Spring 2018

Using git to submit assignments:

Best practices for avoiding conflicts (multiple people editing a file with the **same name**).

Best practices for undoing mistakes (adding, committing, or pushing changes to a file that you didn't mean to change).

Submitting assignments

```
Lecture/

Notebooks/

Code-Review/

Assignment/
```

```
## Do not commit edits to these dirs
Notebooks/
Lecture/

## Use `cp` to make copies of notebooks
## and put your username in the name
cp Notebooks/3.1.ipynb \
Assignment/myname-3.1.ipynb
```

Submitting assignments

Lecture/

Notebooks/

Code-Review/

Assignment/

```
## add and commit only some files
git add Assignment/nb-3.1.ipynb

## do not add all files without checking
git add Assignment/*

## e.g., some hidden files get tracked
git rm -r Assignment/.ipynb_checkpoints/
```

Renaming committed files

Lecture/

Notebooks/

Code-Review/

Assignment/

```
## rename file in git tracking
git mv file2.py file3.py

## Check status of files and filenames
git status

## remove file from being tracked
git rm file3.ipynb
```

Undoing commits

Lecture/

Notebooks/

Code-Review/

Assignment/

```
## What if you made changes to file?
git checkout -- <filename>

## What if you committed changes?
git checkout origin/master <filename>

## What if you pushed changes to master?
## add upstream branch and checkout
git remote add upstream https://github..
git checkout upstream/master <filename>
git add <filename>
git commit -m “reverted <filename>”
```

Installing software part II: pip

The program `pip` is in the predecessor to conda, and can be used to install python packages. However, unlike conda pip can only install python packages and not other binaries. Some packages are currently only packaged for pip and not yet for conda, so we sometimes still use it. In fact, if pip is installed *by* conda then pip installs into the conda directory.

```
## conda commands
conda install pip
conda uninstall pip

## equivalent pip commands
pip install numpy
pip uninstall numpy

## see installed packages
conda list
```

Today's repo

```
Lecture/
  +   PDSB-4-Lecture.pdf

Notebooks/
  +   debruijn.py
  +   eulerian.py
  +   nb-4.0-answers.ipynb
  +   nb-4.1-dicts.ipynb
  +   nb-4.2-debruijn.ipynb
  +   nb-4.3-classes.ipynb
  +   nb-4.4-scripts.ipynb

Assignment/
Code-Review/
```

```
## Install toyplot plotting library
pip install toyplot

## clone and cd into the repo
git clone https://github.com/programm...
cd ./4-Python-advanced/

## start a notebook server
jupyter-notebook
```

Answers from last assignment

Open: Notebooks/4.0-answers.ipynb

Common errors I observed:

+ print versus return
+ Py2 versus Py3 (e.g., integer division)
+ fine-print (return % similar not %diff)
+ complete assignments on time please

```
## function to print (not very useful)
def f(x):
    print(x + 3)

## result can only be seen (OK for
## debugging while writing code.
f(3)

## function to return (very useful)
def f(x):
    return x + 3

## result can be stored
x = f(3)
```

Python advanced: dictionaries, sets, Classes, and imports

Python dicts

Open: Notebooks/4.1-dicts.ipynb

Dictionaries are incredibly useful:

+ Super fast queries
+ Flexible data storage

```
## create an empty dict
data = dict()
data = {}

## fill dict with key, val pairs
for i in range(100):
    key = random.randint(0, 10)
    if key not in data:
        data[key] = 1
    else:
        data[key] += 1
```

Python dicts

Open: Notebooks/4.1-dicts.ipynb

Dictionaries are incredibly useful:

+ Super fast queries
+ Flexible data storage

```
## create an empty dict
data = dict()
data = {}

## fill dict with key, val pairs
for i in range(100):
    key = random.randint(0, 10)
    if key not in data:
        data[key] = [i]
    else:
        data[key].append(i)
```

Python dicts

Open: Notebooks/4.1-dicts.ipynb

Dictionaries are incredibly useful:

+ Super fast queries
+ Flexible data storage

```
## create an empty dict
d1 = {‘a’: 1, ‘c’: 2}
d2 = {‘a’: 3, ‘b’: 4}

## update adds one dict to another and
## replaces values for keys that overlap
d1.update(d2)
{‘a’:3, ‘b’: 4, ‘c’: 2}
```

Python dicts and sets

In: Notebooks/4.1-dict.ipynb

Using %%timeit (IPython magic)
There are a number of *magic* commands that work in ipython/jupyter to perform specific tasks in cells. These start with one or two % symbols. Timeit is useful for comparing efficiency of different implementations of code.

```
## hashed values can be searched quickly
s1 = set(range(int(1e6)))
500000 in s1

## lists are searched slowly
l1 = list(range(int(1e6)))
500000 in l1
```

denovo Genome assembly

Open: Notebooks/4.2-debruijn.ipynb

kmers are all possible substrings of length k in a string. For example, a 4-mer is all length 4 elements of a string.

Kmers can be compared very quickly.
e.g., “ABBA” == “AABB”

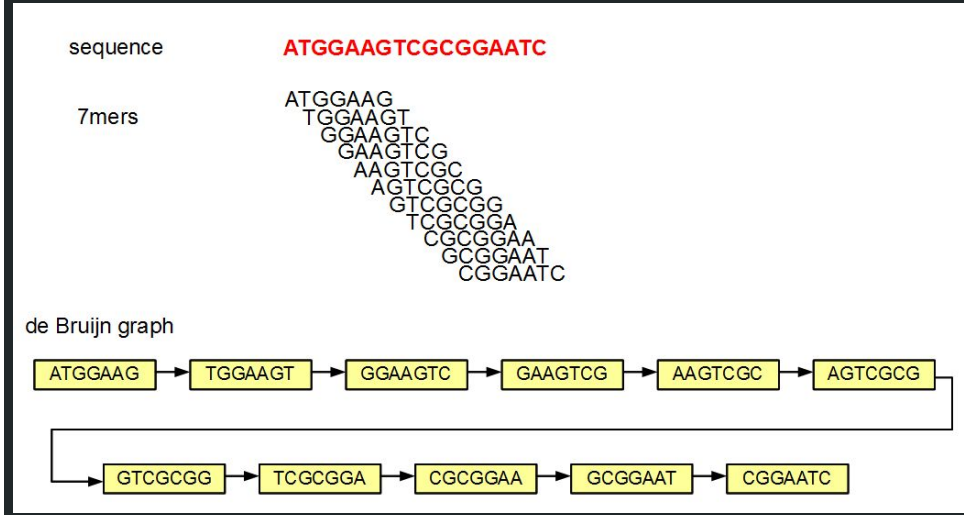
```
## a long string
long1 = “abcdefgh”

## all 4-mers in the string
kmers = [
    “abcd”,
    “bcde”,
    “cdef”,
    “defg”,
    “efgh”,
]
```

denovo Genome assembly

Open: Notebooks/4.2-debruijn.ipynb

de Bruijn graph is a directed graph representing overlaps between symbols. They are used in genome assembly to measure the overlap among kmers. If sufficient kmers exist to cover an entire sequence such that they all overlap by n-1 then the graph can contain information to reconstruct the sequence.



denovo Genome assembly

Open: Notebooks/4.2-debruijn.ipynb

Eulerian path: The correct assembly of the sequence is represented by a walk through the graph such that each edge is visited once. However, if there are **repeats** in the sequence then there may be more than one best path which leads to errors.

```
## Example target sequence
s1 = “AAABBBBA”
```

Python scripts (.py files)

Open: Notebooks/4.2-debruijn.ipynb

Python scripts are files that contain Python code and can be either imported or executed. There is a standard format for the order of elements in a script.

1. hashbang
2. comment
3. imports
4. code
5. __main__

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

“This is an example .py file”

import random

def func1():
    ...
def func2():
    ...

if __name__ == “__main__”:
  ——func1(func2())
```

Python scripts (.py files)

Open: Notebooks/4.2-debruijn.ipynb

Hashbang: labels the file as Python code

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

“This is an example .py file”

import random

def func1():
    ...
def func2():
    ...

if __name__ == “__main__”:
  ——func1(func2())
```

Python scripts (.py files)

Open: Notebooks/4.2-debruijn.ipynb

Comment: tell others what this file is for.

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

“This is an example .py file”

import random

def func1():
    ...
def func2():
    ...

if __name__ == “__main__”:
  ——func1(func2())
```

Python scripts (.py files)

Open: Notebooks/4.2-debruijn.ipynb

Imports: imports should come before any other code. Load in all of the standard library or third-party libraries you will need.

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

“This is an example .py file”

import random

def func1():
    ...
def func2():
    ...

if __name__ == “__main__”:
  ——func1(func2())
```

Python scripts (.py files)

Open: Notebooks/4.2-debruijn.ipynb

Code: Write your Classes and functions here. Try to write short atomic functions and join the functions together to accomplish larger tasks.

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

“This is an example .py file”

import random

def func1():
    ...
def func2():
    ...

if __name__ == “__main__”:
  ——func1(func2())
```

Python scripts (.py files)

Open: Notebooks/4.2-debruijn.ipynb

__main__: This is a weird looking part that goes at the end of the script. It isn't actually required. This is the code that will be executed when you execute this script, not when it is imported. This is important for separating imported parts of the code from executed parts of the code. We'll return to this more later.

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

“This is an example .py file”

import random

def func1():
    ...
def func2():
    ...

if __name__ == “__main__”:
  ——func1(func2())
```

Python scripts (.py files)

Open: Notebooks/4.2-debruijn.ipynb

Let's write a Python script: debruijn.py

Copy all of your functions from 4.2-debruijn.ipynb to a new text file in the jupyter text editor. Save the file in the Notebooks directory (this is important for now) as **debruijn.py**. Then reload notebook 4.2 and try importing the functions from debruijn.py

```
## local import from debruijn.py file
import debruijn

## funcs can now be accessed like this
debruijn.random_sequence()
debruijn.get_kmers()
debruijn.get_reads()
debruijn.reads_to_kmers()
```

Class (object-oriented) programming

Todo: Notebooks/4.3-classes.ipynb

Organizing code into Classes. This streamlines workflows and allows for much more readable and user-friendly code.

```
## writing installable packages
import debruijn

## we'll create a Class object to
## organize function calls
obj = debruijn.Assembler(tlen=100)
obj.assemble(nreads=100, k=4)
obj.draw_graph(width=400)
print(obj.contig)
```

Writing python packages

Todo: Notebooks/4.4-scripts.ipynb

How to write a package that can be *installed* and therefore imported from anywhere. In the example of our *debruijn.py* script so far we can only import it if we are in the same directory.

```
## writing installable packages
import debruijn

## writing executable programs
> python debruijn.py
```

Pull updated repo

For this week's assignment you will need to pull in updates that I have made to the repo **after** you initially cloned it in class. For this you need to add the *upstream branch* as a **remote** and pull from it.

This will pull in Assignments/
Notebooks/4.3-classes.ipynb
Notebooks/4.4-scripts.ipynb

```
## add upstream repo as a remote branch
git remote add upstream https://gith..

## Tuesday or later pull from upstream
## this will pull in changes committed to
## upstream since your fork
git pull upstream master
```

Assignments and readings

Assignment is due Friday at 5pm

Code review is due Monday by class.

Assignment: Link to Session 3 repo

Readings: See syllabus

Collaborate: Work together in this gitter chatroom