

Scripting for Biologists

(or, how to get by just fine without the fancy CS degree)

Vijay Ramani, PhD
Principal Investigator / Sandler Fellow
Dept. of Biochemistry & Biophysics, UCSF

Curriculum adapted from:
Seungsoo Kim, Shendure Lab; Cecilia Noecker, Borenstein Lab
University of Washington

About me!

From:

Short Hills, NJ

Education:

BSE, Chemical Engineering (2012)

PhD, Genome Sciences (2017)

My lab is interested in:

- High-throughput methods
- Single-cell sequencing technology development
- Transcriptional regulation; chromatin architecture
- Metabolic control of chromatin & transcription

Don't hesitate to e-mail me (vijay.ramani@ucsf.edu / vij.ramani@gmail.com) if you have questions about this material, science, grad school, rotations, *etc.*

Goals for Today's Session

- I. Intro to the command line / conda
- II. Learn the basics of coding in Python
- III. Apply some of those basics to elementary bioinformatics problems
- IV. Learn about Numpy, Scipy, Pandas and other data analysis / visualization tools available to you!

Don't hesitate to stop me to ask questions!

Syllabus for Today's Session

- I. Why code and what the heck is a **command line**?
(30 min.)
[5 min. break]
- II. Introduction to **Coding & Python**
(30 min.)
[5 min. break]
- III. Introduction to **Functions and Scripting**
(30 min.)
[5 min. break]
- IV. **Practice Problems / Wrap-up**
(30 - 45 min.)

Syllabus for Today's Session

Download slides for today to follow along here!

<https://github.com/VRam142/bootcamp>

Navigating the **command line**

```
(base) Vijays-MacBook-Pro:~ vijayramani$
```

How to:

- 1.) Change directories and navigate a filesystem
- 2.) Install conda and activate a conda virtual environment
- 3.) Install packages using the conda command line interface

ls: list the contents of a directory

```
(base) Vijays-MacBook-Pro:Desktop vijayramani$ ls
190419_NS500257_0093_AHFFHHBGXB The Jackbox Party Pack 5.app  junk
(base) Vijays-MacBook-Pro:Desktop vijayramani$ ls -l
total 80
drwxrwxrwx   18 vijayramani  staff    612 Apr 23  2019 190419_NS500257_0093_AHFFHHBGXB
-rw-r--r--@   1 vijayramani  staff   36888 Sep  8 11:06 Screen Shot 2020-09-08 at 11.06.49 AM.png
drwxr-xr-x    3 vijayramani  staff    102 Apr 24 19:17 The Jackbox Party Pack 5.app
drwxr-xr-x  355 vijayramani  staff  12070 Sep  8 11:06 junk
(base) Vijays-MacBook-Pro:Desktop vijayramani$
```

ls: list the contents of a directory

```
(base) Vijays-MacBook-Pro:Desktop vijayramani$ ls
190419_NS500257_0093_AHFFHHBGXB The Jackbox Party Pack 5.app    junk
(base) Vijays-MacBook-Pro:Desktop vijayramani$ ls -l
total 80
drwxrwxrwx   18 vijayramani  staff    612 Apr 23  2019 190419_NS500257_0093_AHFFHHBGXB
-rw-r--r--@   1 vijayramani  staff   36888 Sep  8 11:06 Screen Shot 2020-09-08 at 11.06.49 AM.png
drwxr-xr-x    3 vijayramani  staff    102 Apr 24 19:17 The Jackbox Party Pack 5.app
drwxr-xr-x  355 vijayramani  staff   12070 Sep  8 11:06 junk
(base) Vijays-MacBook-Pro:Desktop vijayramani$
```

ls: the ls command can be modified to provide more information about the contents of a directory (e.g. file size, read / write / execute permissions, file ownership). I often use `ls -ltr`, which conveniently lists files in order of when they were last created / modified.

cd: change directory

pwd: path to working directory (where am I now?)

```
(base) Vijays-MacBook-Pro:Desktop vijayramani$ pwd
/Users/vijayramani/Desktop
(base) Vijays-MacBook-Pro:Desktop vijayramani$ cd ~/Downloads/
(base) Vijays-MacBook-Pro:Downloads vijayramani$ pwd
/Users/vijayramani/Downloads
(base) Vijays-MacBook-Pro:Downloads vijayramani$ █
```

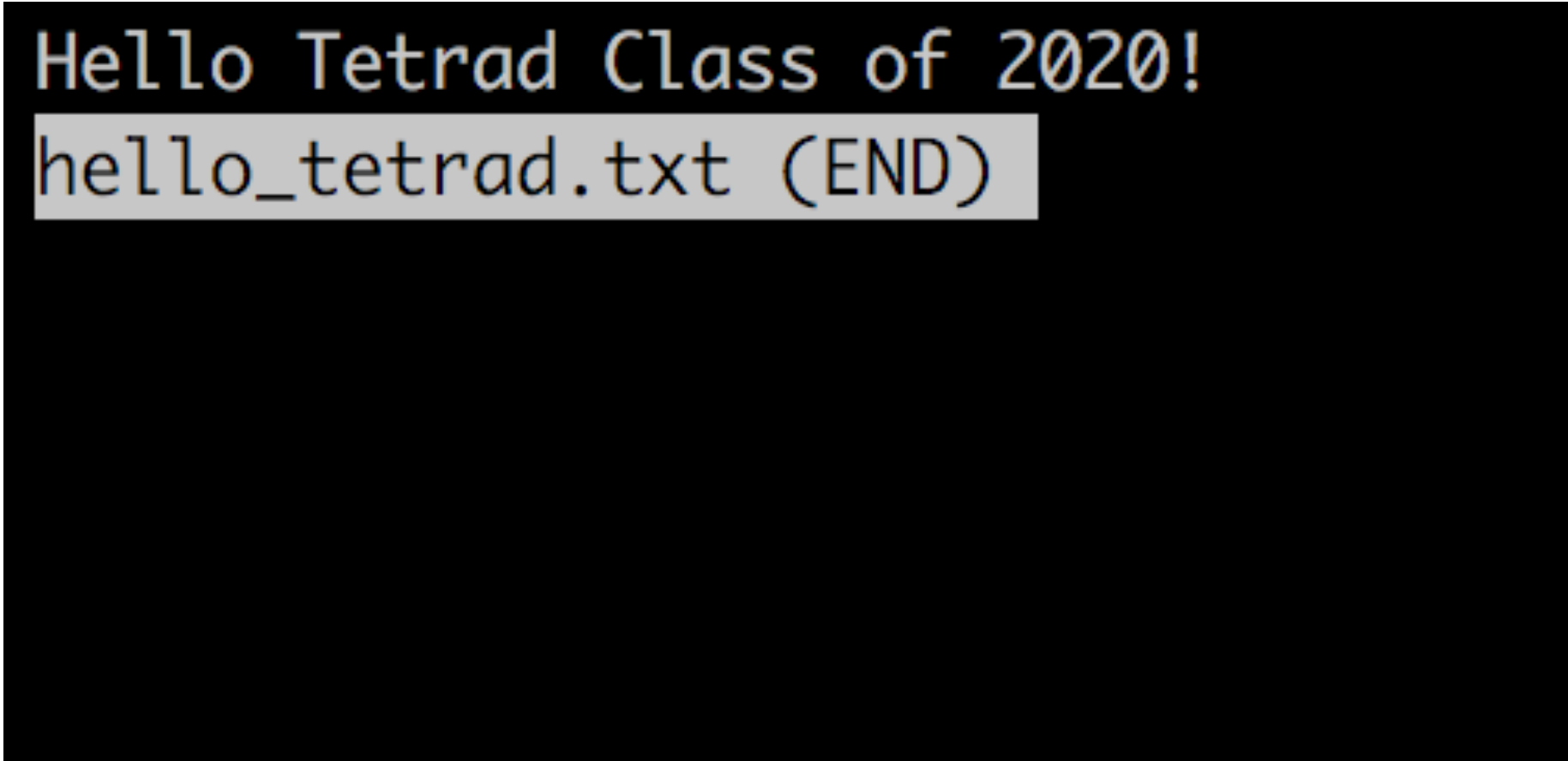
cd: change directory to specified directory (where am I going?)

less: quickly view a text file

```
(base) Vijays-MacBook-Pro:Desktop vijayramani$ less hello_tetrad.txt
```

less: a better version of `more` (haha.) that allows us to quickly read and page through any text file.

`less`: quickly view a text file

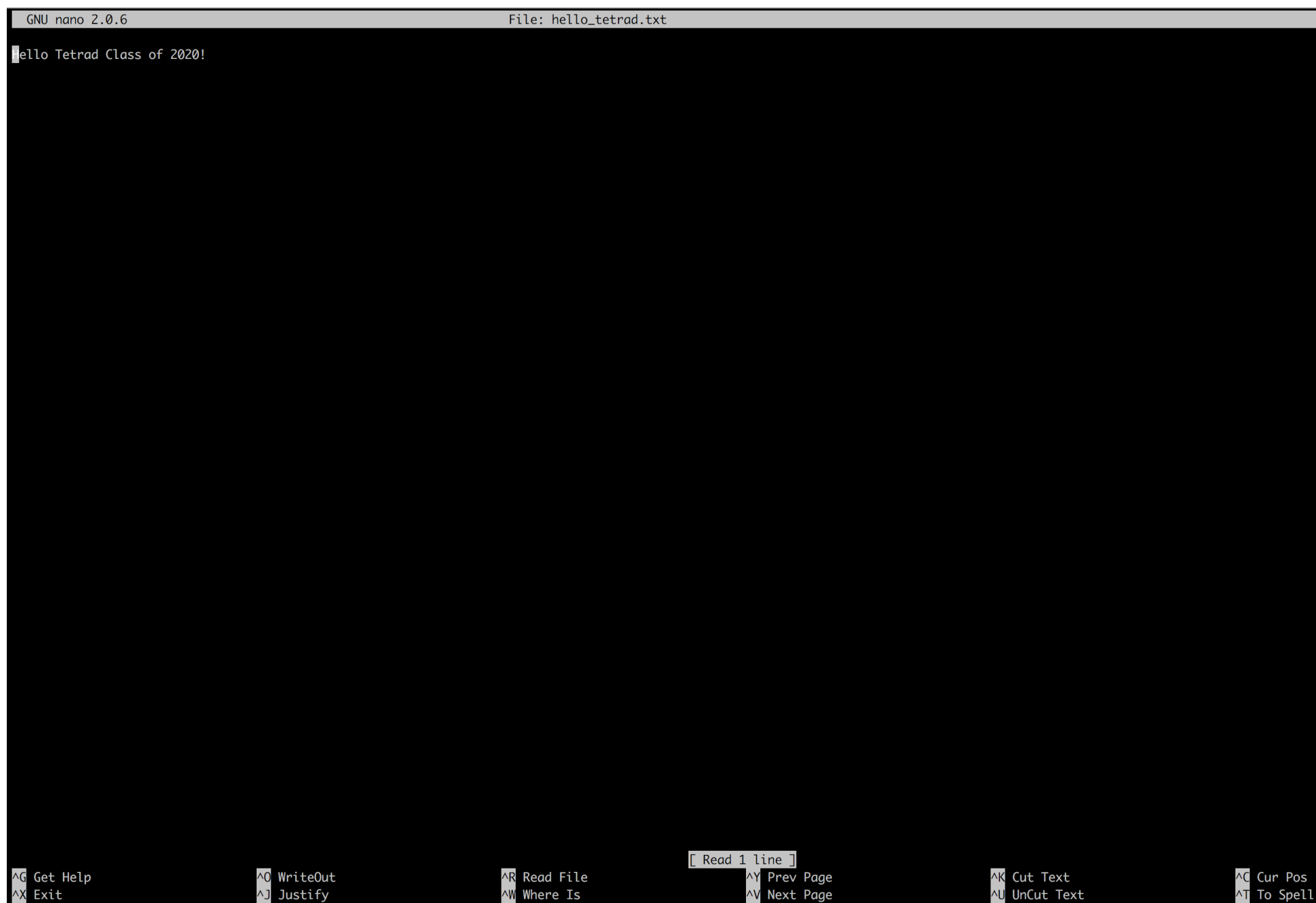
A terminal window with a black background. The first line of text is "Hello Tetrad Class of 2020!" in a light blue font. The second line is "hello_tetrad.txt (END)" in a light blue font, and this line is highlighted with a light gray background.

```
Hello Tetrad Class of 2020!  
hello_tetrad.txt (END)
```

`less`: a better version of `more` (haha.) that allows us to quickly read and page through any text file.

Text editors: nano, vim, emacs

`nano [filename]` allows us to edit that text file. Different text editors have different graphical user interfaces (GUIs).



The screenshot shows the GNU nano 2.0.6 text editor running in a terminal. The title bar at the top indicates the file being edited is `hello_tetrad.txt`. The main editing area contains the text `ello Tetrad Class of 2020!` with a cursor at the end of the line. The bottom status bar displays various keyboard shortcuts for editing and navigation, including `^G Get Help`, `^O WriteOut`, `^R Read File`, `^W Where Is`, `^Y Prev Page`, `^K Cut Text`, `^C Cur Pos`, `^X Exit`, `^J Justify`, `^V Next Page`, `^U UnCut Text`, and `^T To Spell`. A small status indicator shows `[Read 1 line]`.

Installing anaconda / conda

Install anaconda:

1.) Download https://repo.anaconda.com/archive/Anaconda3-2020.07-MacOSX-x86_64.sh

2.) Run:

```
bash ~/Downloads/Anaconda3-2020.07-MacOSX-x86_64.sh
```

3.) Follow the prompts and complete the installation.

4.) Profit.

Installing anaconda / conda

Create and activate your first conda environment

```
(base) Vijays-MacBook-Pro:Desktop vijayramani$ conda create -n Tetrad_Bootcamp
/Users/vijayramani/anaconda/lib/python2.7/site-packages/cryptography/hazmat/primitives/constant_time.py:26: CryptographyDeprecationWarning: Support for your Python version is deprecated. The next version
of cryptography will remove support. Please upgrade to a release (2.7.7+) that supports hmac.compare_digest as soon as possible.
  utils.PersistentlyDeprecated2018,
Collecting package metadata (current_repodata.json): done
Solving environment: done

==> WARNING: A newer version of conda exists. <==
  current version: 4.7.12
  latest version: 4.8.4

Please update conda by running

  $ conda update -n base -c defaults conda

## Package Plan ##

  environment location: /Users/vijayramani/anaconda/envs/Tetrad_Bootcamp

Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate Tetrad_Bootcamp
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) Vijays-MacBook-Pro:Desktop vijayramani$
```

Installing anaconda / conda

Create and activate your first conda environment

```
(base) Vijays-MacBook-Pro:Desktop vijayramani$ conda activate Tetrad_Bootcamp  
(Tetrad_Bootcamp) Vijays-MacBook-Pro:Desktop vijayramani$ █
```

Why should we bother with virtual environments?

Compartmentalizing your coding workflows into virtual environments allows others to reproduce your work by keeping all of the 'dependencies' of your code in one convenient location. Conda also solves program compatibility on the fly, so you can be sure all of your installed programs will behave as expected.

Use conda to install jupyter, salmon, tximport, and DESeq2

```
conda install -c anaconda jupyter
```

```
conda install -c bioconda salmon
```

```
conda install -c bioconda bioconductor-tximport
```

```
conda install -c bioconda bioconductor-deseq2
```


Break!



A Practical Introduction to Coding

Programming uses **language**...

A Practical Introduction to Coding

Programming uses **language**...

...and languages have **rules**

A Practical Introduction to Coding

Programming uses **language**...

...and languages have **rules**

Let's eat Grandma!

A Practical Introduction to Coding

Programming uses **language**...

...and languages have **rules**

Let's eat Grandma!

Let's eat, Grandma!

What is a program?

A program is a list of instructions,
written in a language your computer understands!

What is a program?

A program is a list of instructions,
written in a language your computer understands!

```
1: 98 degrees, 5 min.  
2: 98 degrees, 30 sec.  
3: 60 degrees, 30 sec.  
4: 72 degrees, 1 min.  
5: GOTO 2, 20 TIMES  
6: 72 degrees, 10 min.  
7: 4 degrees, FOREVER  
8: END
```

What is a program?

A program is a list of instructions,
written in a language your computer understands!

```
In [1]: import os,sys,re  
        print("Hello Tetrad!")  
Hello Tetrad!
```


Building blocks of a **python** program

Coding in python just requires some knowledge of the basic topics:

- Variables
- Data Types
- Control logic & loops
- Argument parsing, File I/O, functions

Open up **jupyter** to code along!

```
(Tetrad_Bootcamp) Vijays-MacBook-Pro:coding_bootcamp vijayramani$ jupyter notebook
```

Download the Tetrad Coding Bootcamp Notebook here:

Download the folder for this afternoon here:

<https://ucsf.box.com/s/yyz1tydhtvw7twuvw4i3y7107ckaij6a>

Variables in python

Variables are the “subjects” of the instructions you’re giving the computer:

```
In [4]: x = 1 #the variable "x" now has the value 1
        y = 2 #the variable "y" now has the value 2
        z = x + y # lets initialize a variable z that is x + y

        #And then let's print out all of those values!
        print(x)
        print(y)
        print(z)

1
2
3
```

Data types in **python**

We initialize variables with different **data types**, depending on what we'd like our program / script to do!

Some useful data types are: `string`, `int`,
`float`, `list`, `dict`

Let's go over them in some detail. In the interest of time I won't go over a bunch of other data types—read about them in the Python resources we sent over!

String, int, and float

`String` lets you represent text. When we print something out to a file, we're printing it out as a string. We denote strings through the use of double quotes (`"``"`).

```
In [1]: foo = "Hello world!"  
        print(foo)
```

```
Hello world!
```

String, int, and float

Int and float are numerical representations with differing *precision*. Integers cannot represent fractional values, while floating point numbers can.

```
In [4]: x = 1 #the variable "x" now has the value 1
        y = 2 #the variable "y" now has the value 2
        z = x + y # lets initialize a variable z that is x + y

        #And then let's print out all of those values!
        print(x)
        print(y)
        print(z)

1
2
3
```

Recasting variables and a warning!

Python is **dynamically typed**, meaning that values, not variables, are assigned data types! This is generally incredibly convenient, but can lead to some unwanted behaviors, too!

```
In [11]: string = "3"
integer = 3

string2int = int(string)
print(string2int + integer) #What happens when I sum integers?
print(string + string) #What happens when I sum strings?

int2string = str(integer)
print(int2string + integer) #What happens when I try to sum an integer and a string?
```

6
33

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-11-f946d4918ccb> in <module>()
      7
      8 int2string = str(integer)
----> 9 print int2string + integer #What happens when I try to sum an integer and a string?

TypeError: cannot concatenate 'str' and 'int' objects
```

list and dict

In many cases we would like to store data in a container: we can do this using special data types called “data structures”

Lists are similar to “vectors” or “arrays,” and are initialized using brackets `[]`, and elements can be addressed, reset, or subset using indexing (by selecting start and end indices and using the `:` operator).

Lists can be concatenated, and new items can be added using `+` or `list.append()`

list and dict

```
In [13]: #Lists can be comprised of many data types!
newList = [1,2,3,4,5,"cheese","crackers","wine",3.141592654]
print(newList)
#We can address different elements of the list using indexing
print(newList[0])
print(newList[5])
print(newList[-1])
#We can also subset lists through "slicing"
print(newList[:5])
print(newList[:5:-1])
print(newList[:-3])
#Finally, we can add stuff to lists using the + operator, or the append function

#Option 1
newObject = ["Add me!"]
newList += newObject
print(newList)

#Option 2
newObject = "Add me!"
newList.append(newObject)
print(newList)

[1, 2, 3, 4, 5, 'cheese', 'crackers', 'wine', 3.141592654]
1
cheese
3.141592654
[1, 2, 3, 4, 5]
[3.141592654, 'wine', 'crackers']
[1, 2, 3, 4, 5, 'cheese']
[1, 2, 3, 4, 5, 'cheese', 'crackers', 'wine', 3.141592654, 'Add me!']
[1, 2, 3, 4, 5, 'cheese', 'crackers', 'wine', 3.141592654, 'Add me!', 'Add me!']
```

list and dict

`Dict` are initialized using curly brackets `{ }`, and represent a special type of data structure: a “hash table.”

Dictionaries are comprised of **keys** and **values**, which are paired in a single data structure. Dictionary keys must be **immutable** objects (strings, ints, floats, etc.). Dictionary values can be **ANYTHING**. We look up values in a dict using brackets `[]`.

Dictionaries are very useful for “associating” data: key membership (*i.e.* is a key in the dict or not) checks and retrieving associated data is instantaneous!

list and dict

```
In [14]: myDict = {"apple": "Fruit", "orange": "Fruit", "tomato": "Fruit", "Broccoli": "Vegetable"}
print(myDict["apple"])
print(myDict["Broccoli"])
```

```
Fruit
Vegetable
```

```
In [19]: #Lets initialize some dictionaries of dictionaries
dictOfDicts = {1:{}, 2: {}}
dictOfDicts[1]["Sox2"] = 35.0 #We can add key:value pairs to a dictionary like so
dictOfDicts[2]["Sox2"] = 50.0

print(dictOfDicts[1]["Sox2"])
print(dictOfDicts[2]["Sox2"])
```

```
35.0
50.0
```

```
In [26]: #Dictionaries need to have unique immutable keys, but can have non-unique, mutable values!
foo = [1,2,3]
badDict = {foo:2, 1:2, 1:3}
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-26-0ef5ac57fc86> in <module>()
      1 #Dictionaries need to have unique immutable keys, but can have non-unique, mutable values!
      2 foo = [1,2,3]
----> 3 badDict = {foo:2, 1:2, 1:3}

TypeError: unhashable type: 'list'
```

Loops & control logic

Now that we have some data types to play with, we need to do something with these!

Your first python scripts will 1.) read in data as variables, and then 2.) use logic & loops to process that data.

Lets go over loops and control logic.

Loops

`for` loops:

Loops allow us to efficiently carry out many operations.

Python very conveniently allows users to *iterate* over many objects using loops. The syntax is generally as follows (**note the whitespace!**):

```
for i in [list]:  
    [do something]
```

Lots of stuff can be iterated over, including lists, dicts, and strings!

Loops

```
In [27]: loopList = [1,2,3,4,5,6,7,8,9,10]

#Here is the syntax for a for loop. Note the whitespace!!
#Whitespace is important in Python
for i in loopList:
    j = i + 10
    print i
    print j
```

```
1
11
2
12
3
13
4
14
5
15
6
16
7
17
8
18
9
19
10
20
```

Control logic & while loops

Logic in python:

Boolean logic allows us to control how a program executes certain operations depending on the situation.

Syntactically, we use `if`, `else`, `and`, `or`, `elif` in combination with operators like `==`, `!=`, `<=`, `>=`, and `in`

Control logic & while loops

```
In [34]: loopList = [1,2,3,4,5,6,7,8,9,10]
loopDict = {1:"moo", 3:"baa", 5:"oink", 7:"roar"}

#Lets explore a function that only prints all of the even numbers
#in the list above
for i in loopList:
    if i % 2 == 0: #The % operator gives us the remainder of i / 2
        print i
    elif i > 5:
        print "Not even and also greater than 5"
    else:
        print "Not even and less than 5"

#Now lets try some other syntax:
for i in loopList:
    if i in loopDict:
        #here we're using string formatting to print something. The syntax for string formatting is:
        # "%s" % (variable name), and you can do this for as many variables as you'd like
        print "%s\t%s" % (i, loopDict[i])
```

```
Not even and less than 5
2
Not even and less than 5
4
Not even and less than 5
6
Not even and also greater than 5
8
Not even and also greater than 5
10
1      moo
3      baa
5      oink
7      roar
```


while loops

while loops:

while loops allow us to loop *ad infinitum* until certain conditions are met.

```
In [40]: i = 0  
while i < 5:  
    print i  
    i += 1
```

```
0  
1  
2  
3  
4
```

Argument parsing & File I/O

We are using the **jupyter notebook environment** to script in Python, but we should really only use notebooks for reproducible data exploration + visualization. Reproducible data analysis pipelines should be **standalone scripts**.

A script is a multi-line python file with the suffix *.py, which we execute in bash as:

```
python *.py [argument1] [argument2] [etc.]
```

Now we are going to cover how to code command-line arguments, read in, and write to files using custom Python scripts!

Argument parsing & File I/O

We read in command-line arguments using the `sys` module. These arguments can be stored in variables as such:

```
import sys
cmd1 = sys.argv[1] #this is argument 1
cmd2 = sys.argv[2] #this is argument 2
cmd3 = sys.argv[3] #this is argument 3
```

Argument parsing & File I/O

We read in command-line arguments using the `sys` module. These arguments can be stored in variables as such:

```
import sys
cmd1 = sys.argv[1]#this is argument 1
cmd2 = sys.argv[2]#this is argument 2
cmd3 = sys.argv[3]#this is argument 3
```

We read in files using the `open()` and `.close()` commands. Always remember to close your file handles!! In this example, the file we want to open is command-line argument #1.

```
fhi = open(sys.argv[1], 'r')
fhi.close()
```

Argument parsing & File I/O

We read in command-line arguments using the `sys` module. These arguments can be stored in variables as such:

```
import sys
cmd1 = sys.argv[1]#this is argument 1
cmd2 = sys.argv[2]#this is argument 2
cmd3 = sys.argv[3]#this is argument 3
```

We read in files using the `open()` and `.close()` commands. Always remember to close your file handles!! In this example, the file we want to open is command-line argument #1.

```
fhi = open(sys.argv[1], 'r')
fhi.close()
```

We write to files using the `open()` function with the `'w'` flag, and then use the `print` function to and `.close()` commands.

```
fho = open(sys.argv[2], 'w')
print("%s\t%s" % (string1, string2), file = fho)
fho.close()
```

Functions

Modular programming allows us to script functions that we think might be generally useful, and then call these functions farther down the line. The syntax for defining functions is:

```
def myFunction():
```

Functions can be used for anything! It's probably best to illustrate how functions work using an example. Copy the example in your jupyter notebooks into a new file using your favorite TextEditor, and save that file as `boot_camp.py`.

Functions

```
In [ ]: import os,sys,re

def myFunction(dictionary, outfile):
    '''This function takes in a dictionary of scores, and then
    prints the highest and lowest names and scores to an outfile'''
    val_max = 0
    val_min = 0
    for i in dictionary:
        if dictionary[i] < val_min:
            val_min = dictionary[i]
            lowest = i
        if dictionary[i] > val_max:
            val_max = dictionary[i]
            highest = i
    print >> outfile, "%s\t%s\tHighScore" % (highest, dictionary[highest])
    print >> outfile, "%s\t%s\tLowScore" % (lowest, dictionary[lowest])
```

Functions

```
In [ ]: import os,sys,re

def myFunction(dictionary, outfile):
    '''This function takes in a dictionary of scores, and then
    prints the highest and lowest names and scores to an outfile'''
    val_max = 0
    val_min = 0
    for i in dictionary:
        if dictionary[i] < val_min:
            val_min = dictionary[i]
            lowest = i
        if dictionary[i] > val_max:
            val_max = dictionary[i]
            highest = i
    print >> outfile, "%s\t%s\tHighScore" % (highest, dictionary[highest])
    print >> outfile, "%s\t%s\tLowScore" % (lowest, dictionary[lowest])

fhi = open(sys.argv[1]) #This allows us to open a file-handle for input
fho = open(sys.argv[2], 'w') #This allows us to open a file-handle for output; don't forget the 'w' flag!

score_dictionary = {}

for line in fhi: #Files can be iterated over as well!
    split = line.split('\t') #This very useful function allows us to split the line into a list
    score_dictionary[split[0]] = int(split[1])

myFunction(score_dictionary, fho)

fhi.close()
fho.close()
```


Functions

```
In [ ]: import os,sys,re

def myFunction(dictionary, outfile):
    '''This function takes in a dictionary of scores, and then
    prints the highest and lowest names and scores to an outfile'''
    val_max = 0
    val_min = 0
    for i in dictionary:
        if dictionary[i] < val_min:
            val_min = dictionary[i]
```

We now have the **building blocks** for programs.
What are some things we might want to do?

```
fhi = open(sys.argv[1]) #This allows us to open a file-handle for input
fho = open(sys.argv[2], 'w') #This allows us to open a file-handle for output; don't forget the 'w' flag!

score_dictionary = {}

for line in fhi: #Files can be iterated over as well!
    split = line.split('\t') #This very useful function allows us to split the line into a list
    score_dictionary[split[0]] = int(split[1])

myFunction(score_dictionary, fho)

fhi.close()
fho.close()
```

BREAK

Part II: Problem Solving

Problem #1: Counting substrings

There are going to be times where we have a given sequence (say, 'ACGT'), and are interested in counting the number of times this sequence occurs in a longer sequence (e.g. 'ACGTGTAGATACGT').

- a.) Write a script that takes in the 4-mer 'ACGT,' and finds the number of times it occurs in the longer sequence 'ACGTGTAGATACGT'.
- b.) Write a script that takes in the 6-mer 'CACGTG,' the file `chr1.txt`, and prints out the number of times that 6-mer occurs in the sense orientation.
- c.) Knowing the number of times a subsequence occurs in the context of a longer sequence is great, but there will also be times we need to find *where* that subsequence is. Write a script that takes in the same 6-mer and text file as above, and writes a file with the locations (1-indexed!) of all matches in the sense direction.

Problem #2: Counting all substrings

In sequence analysis, there are also going to be times where we might be interested in the relative abundance of all sequences of length k (k -mers). Sometimes, we will want to compute the extent to which certain k -mers are enriched or depleted with respect to random sequence.

a.) Write a script that takes in `chr1.txt` as a command-line argument, tabulates the relative abundance of 1-mers (*i.e.* A,C,G,T), 3-mers and 6-mers, and prints out these counts to three separate text files.

b.) These files are super helpful, because we can now compute a *background distribution* describing the frequency with which we expect to see a sequence of length k in our data. For simplicity's sake, let's just say that the odds of drawing a sequence of length k by chance is:

$$\prod_{i=1}^k P(A|C|G|T)_k$$

Using the previously computed 1-mer abundances, compute the probabilities of observing all possible 3 and 6-mers, and then calculate the relative enrichment and depletion ($\log_2(\text{observed} / \text{expected})$) of observed 3- and 6-mers with respect to this background distribution. What is the most enriched 6-mer? The most depleted 6-mer?

BREAK

Part III: Leveraging the Bioinformatics Ecosystem



Part III: Leveraging the Bioinformatics Ecosystem

The beauty of using computing languages like Python and R is that there are *tons* of packages & resources at your disposal—take advantage of them when exploring and visualizing your data!

Important packages for Python:

Numpy

Scipy

SciKit-Learn

Matplotlib

pandas

Important packages for R*:

tidyverse

glmnet

*: we'll go over a tiny bit of R before the end of the day

An aside: why use notebooks?

Notebooks give us a persistent, easy-to-navigate record of the code / analyses / visualizations we've already generated. This lets us quickly recap data experiments we've already performed, and allows others to quickly reproduce our work!

An aside: why use notebooks?

Notebooks give us a persistent, easy-to-navigate record of the code / analyses / visualizations we've already generated. This lets us quickly recap data experiments we've already performed, and allows others to quickly reproduce our work!

Jupyter (referencing Julia, Python, and R) supports kernels for three different languages, allowing us to seamlessly switch between them in the same notebook.

An aside: why use notebooks?

Notebooks give us a persistent, easy-to-navigate record of the code / analyses / visualizations we've already generated. This lets us quickly recap data experiments we've already performed, and allows others to quickly reproduce our work!

Jupyter (referencing Julia, Python, and R) supports kernels for three different languages, allowing us to seamlessly switch between them in the same notebook.

Note: package installation is handled by `anaconda`, which should already be installed on your laptops. Make sure `numpy`, `scipy`, `pandas`, and `matplotlib` are installed!

Messing around with `pandas` & friends

The best way to learn how to use these packages is to just dive in & use them, errors / bugs be damned.

I highly recommend using cookbook examples from Julia Evans (<https://jvns.ca/> ; <https://github.com/jvns>), which illustrate the core functionalities of `pandas` and provide examples for basic data visualization using `matplotlib`

A quick note on R

There are cases where you **should** be using R (*i.e.* certain bulk- and single-cell RNA-seq analysis tools; a large number of powerful statistical tools / tests); personally, I like to preprocess data in Python, then use R for these fantastic data visualization / statistical tools.

We don't have time to dive into it, but you should all check out `tidyverse` (<https://www.tidyverse.org/packages/>), Hadley Wickham's incredible suite of tools for data analysis in R. I make **all** of my publication-quality plots using `ggplot2` — I would recommend trying it out, using your Jupyter notebook!

Follow this link (<https://irkernel.github.io/>) to make sure your Jupyter install can switch between the Python and R kernels.

Closing remarks

We've covered a lot today, but hopefully the snippets of code you've come up with will come in handy further down the line.

Remember, a lot of bioinformatics is actually quite straightforward (*e.g.* read in a file, iterate through lines, reformat the data into a table ready for `pandas` or `R`, compute some summary statistics).

Many of the statistically rigorous pipelines for analyzing data have been well-worked out (*e.g.* DESeq for bulk RNA-seq; Monocle / Seurat for scRNA-seq), and even for new applications the concepts implemented there often represent good starting points.

Don't be afraid to build on (with attribution, of course) the work of bioinformaticians past! There's a reason why all of this code is open source.