PLEASE DOWNLOAD ANACONDA NOW

HTTPS://WWW.CONTINUUM.IO/DOWNLOADS

OR

GOOGLE: ANACONDA PYTHON

FEEL FREE TO USE YOUR OWN COMPUTER

DATA ANALYSIS IN PYTHON

A 5 HOUR TOUR OF PYTHON FOR SCIENTIFIC COMPUTING

OUTLINE

1. Fundamental Python
2. IPython: The I is for Interactive
3. Numpy: How to add science to python!
4. Pandas: The secret sauce for python data analysis
5. Matplotlib: Plots and figures.

MATERIALS

- http://cogmech.ucmerced.edu/bkerster/workshop.zip
- Make sure you have anaconda
  - Bundle that includes python and most of the libraries essential to scientific programming
  - https://www.continuum.io/downloads

ABOUT ME

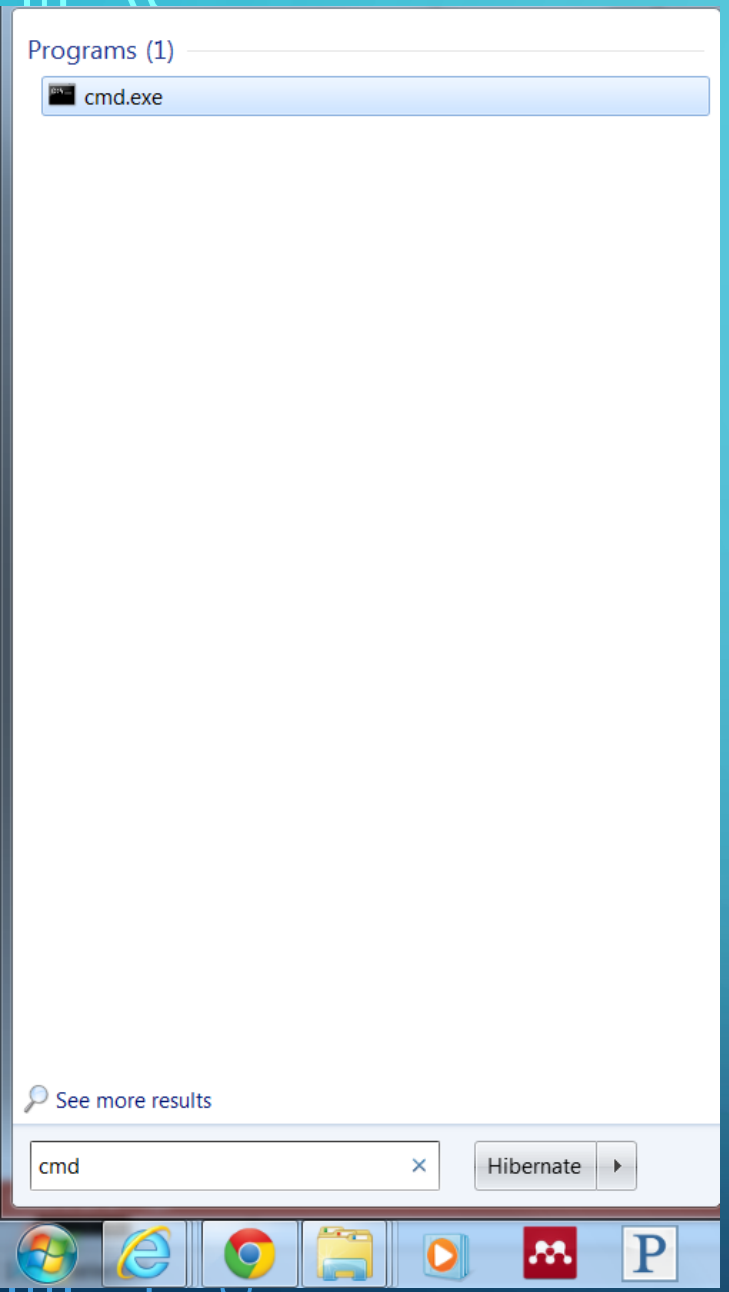
- 6th year graduate student in CIS
- Switched from MATLAB to Python about 3.5 years ago
- Have used python to analyze several large data sets
  - 2000+ participant interactive experiments
  - Yelp dataset
  - A private database of social media activity including text content and numerical metrics
- Graduating shortly
  - Insight Data Science Fellow before entering industry

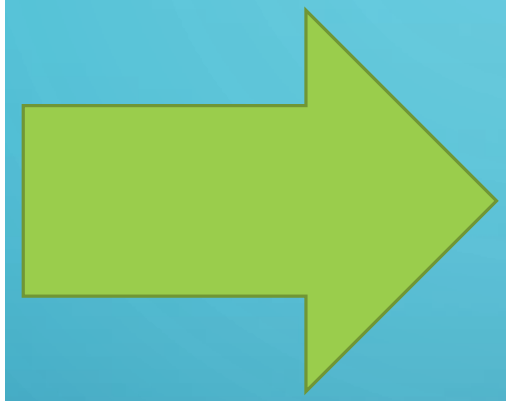
PROGRAMMING IN PYTHON

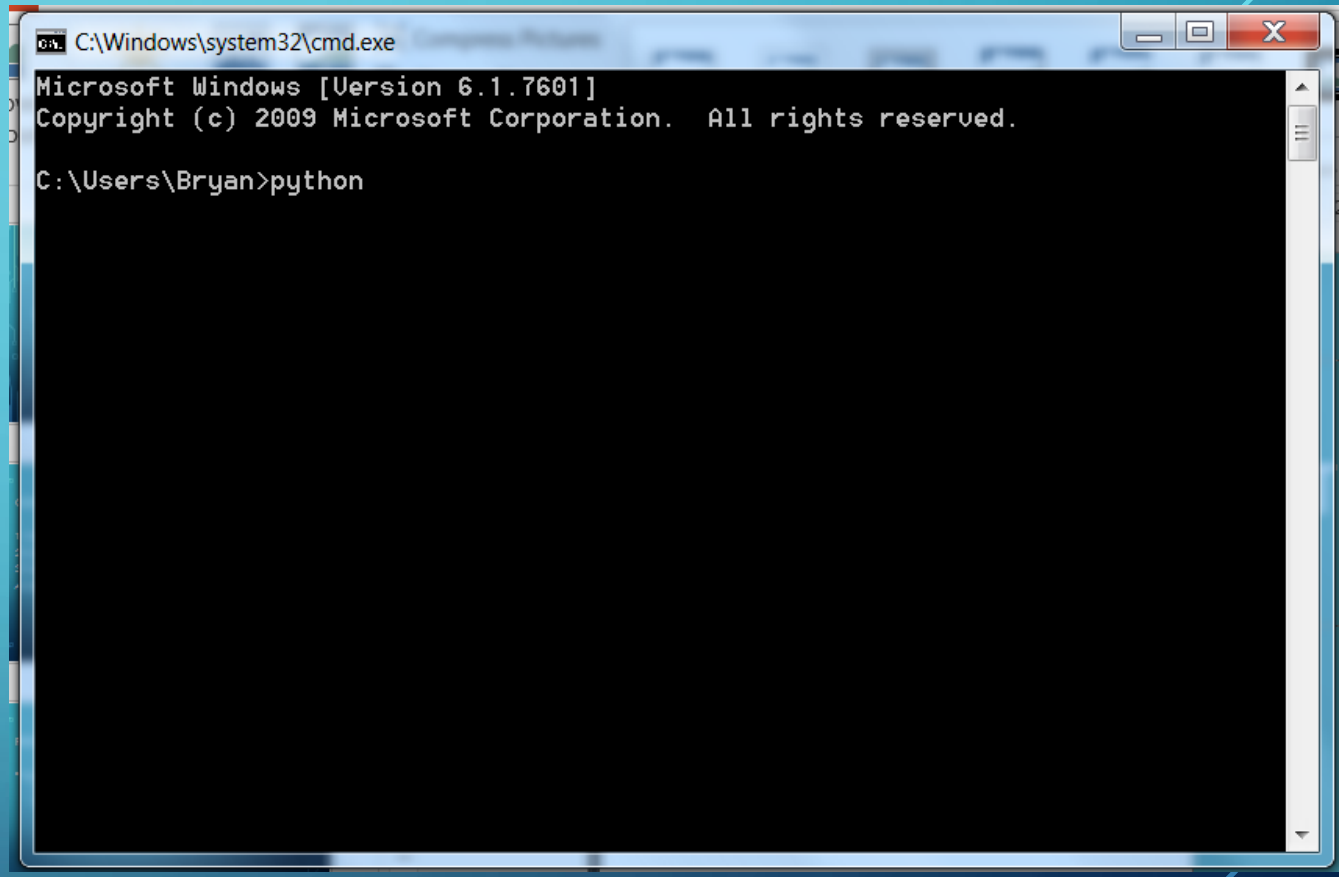
- Assumptions:
  - You have done **some** programming before
  - That's it.

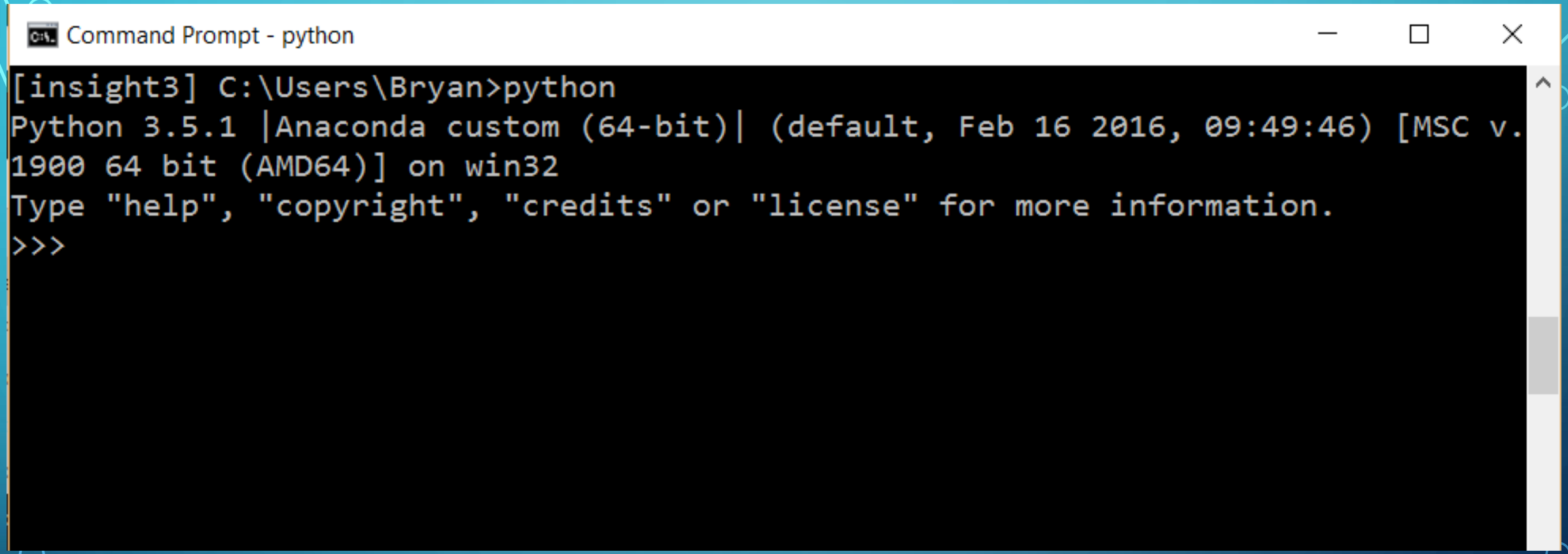
HOW TO GET PYTHON

- Download and install Anaconda
  - https://store.continuum.io/cshop/anaconda/
- It's free
- Includes almost all of the scientific packages you will never need
  - Don't need to figure out how to install complicated modules


Programs (1)
cmd.exe
See more results
cmd
Hibernate

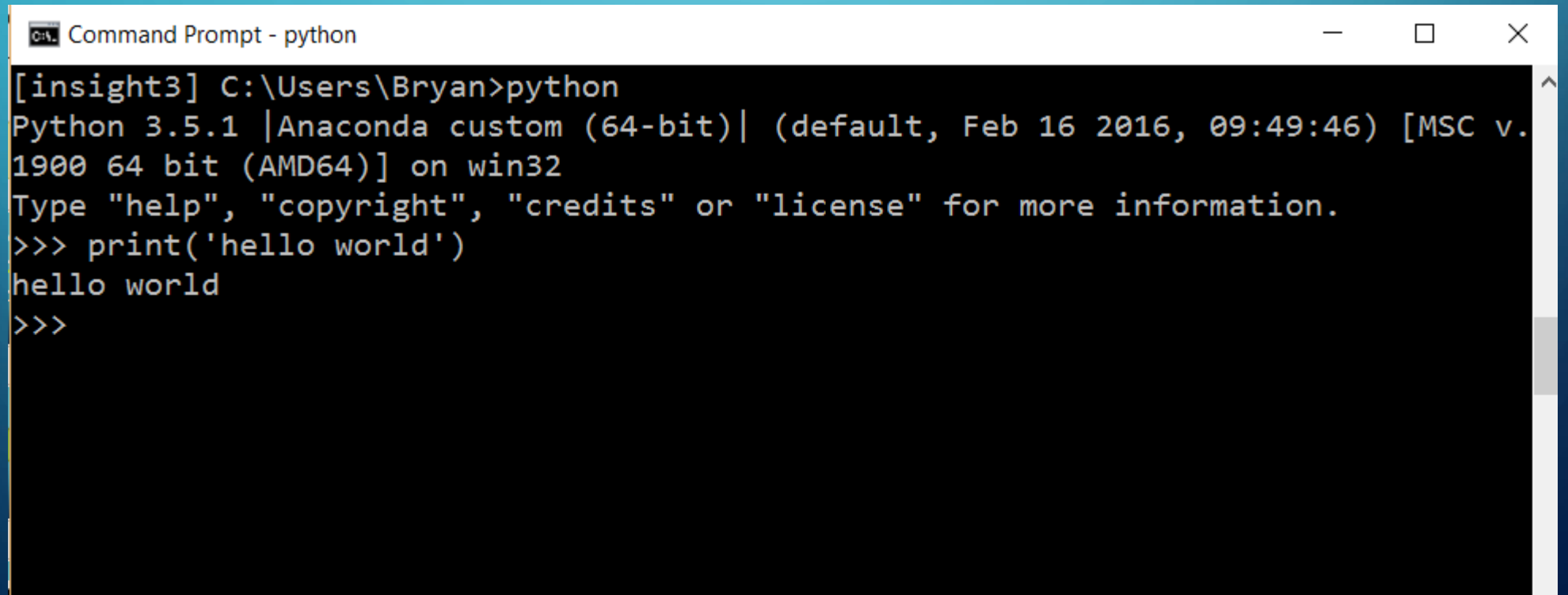



C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Bryan>python


Command Prompt - python
[insight3] C:\Users\Bryan>python
Python 3.5.1 |Anaconda custom (64-bit)| (default, Feb 16 2016, 09:49:46) [MSC v.
1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

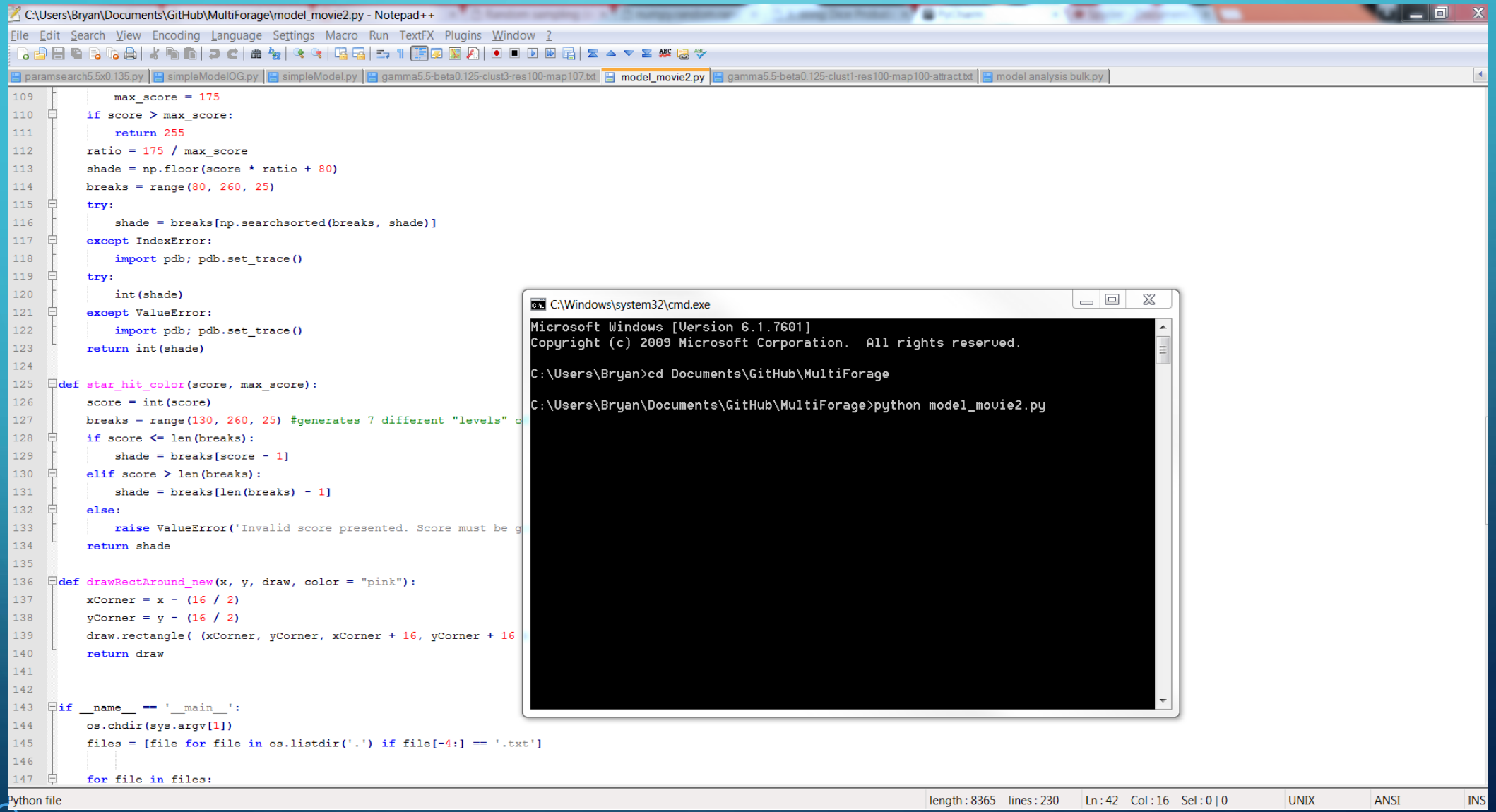
HELLO WORLD

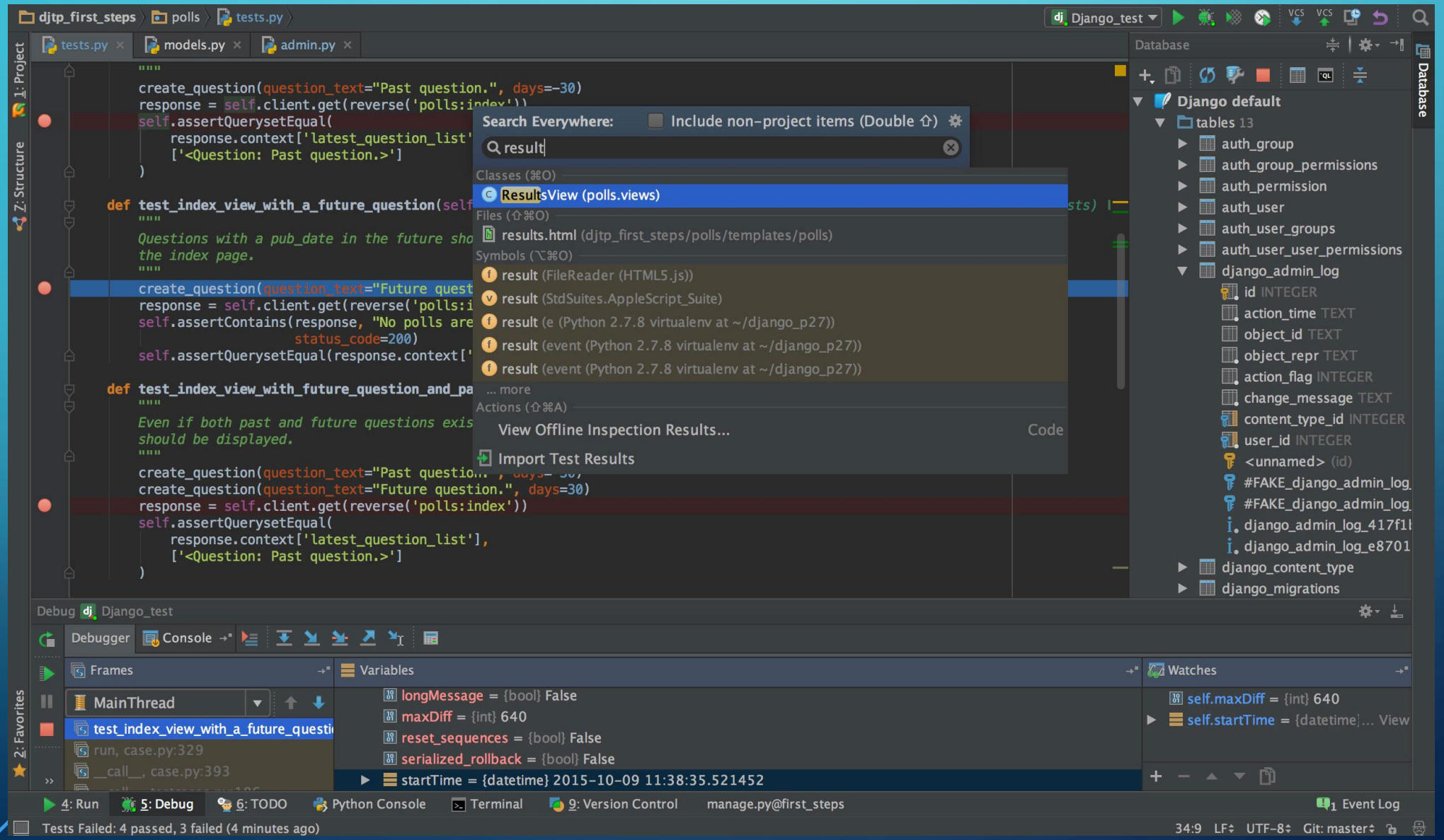
- You can print to the command line in python with the command `print`
  - Try `print( 'hello world' )`



WORK ENVIRONMENT

- Common ways to work in python
  - Text editor + command line
    - Good for running smaller scripts
  - Full IDEs
    - Pycharm is the most common: https://www.jetbrains.com/pycharm/
    - Spyder is also frequently used for smaller projects. Similar to matlab in style
  - IPython / Jupyter
    - Great for data analysis


C:\Users\Bryan\Documents\GitHub\MultiForage\model_movie2.py - Notepad++
File Edit Search View Encoding Language Settings Macro Run TextFX Plugins Window ?
paramsearch5.5x0.135.py
simpleModelOG.py
simpleModel.py
gamma5.5-beta0.125-clust3-res100-map107.txt
model_movie2.py
gamma5.5-beta0.125-clust1-res100-map100-attract.txt
model analysis bulk.py
109 max_score = 175
110 if score > max_score:
111 return 255
112 ratio = 175 / max_score
113 shade = np.floor(score * ratio + 80)
114 breaks = range(80, 260, 25)
115 try:
116 shade = breaks[np.searchsorted(breaks, shade)]
117 except IndexError:
118 import pdb; pdb.set_trace()
119 try:
120 int(shade)
121 except ValueError:
122 import pdb; pdb.set_trace()
123 return int(shade)
124
125 def star_hit_color(score, max_score):
126 score = int(score)
127 breaks = range(130, 260, 25) #generates 7 different "levels" o
128 if score <= len(breaks):
129 shade = breaks[score - 1]
130 elif score > len(breaks):
131 shade = breaks[len(breaks) - 1]
132 else:
133 raise ValueError('Invalid score presented. Score must be g
134 return shade
135
136 def drawRectAround_new(x, y, draw, color = "pink"):
137 xCorner = x - (16 / 2)
138 yCorner = y - (16 / 2)
139 draw.rectangle((xCorner, yCorner, xCorner + 16, yCorner + 16
140 return draw
141
142
143 if __name__ == '__main__':
144 os.chdir(sys.argv[1])
145 files = [file for file in os.listdir('.') if file[-4:] == '.txt']
146
147 for file in files:
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Bryan>cd Documents\GitHub\MultiForage
C:\Users\Bryan\Documents\GitHub\MultiForage>python model_movie2.py
Python file
length : 8365 lines : 230
Ln : 42 Col : 16 Sel : 0 | 0
UNIX
ANSI
INS


djtp_first_steps
polls
tests.py
Django_test
tests.py
models.py
admin.py
Search Everywhere:
Include non-project items (Double ⇧)
result
Classes (⌘O)
ResultsView (polls.views)
Files (⇧⌘O)
results.html (djtp_first_steps/polls/templates/polls)
Symbols (⌥⌘O)
result (FileReader (HTML5.js))
result (StdSuites.AppleScript_Suite)
result (e (Python 2.7.8 virtualenv at ~/django_p27))
result (event (Python 2.7.8 virtualenv at ~/django_p27))
result (event (Python 2.7.8 virtualenv at ~/django_p27))
... more
Actions (⇧⌘A)
View Offline Inspection Results...
Code
Import Test Results
Database
Django default
tables 13
auth_group
auth_group_permissions
auth_permission
auth_user
auth_user_groups
auth_user_user_permissions
django_admin_log
id INTEGER
action_time TEXT
object_id TEXT
object_repr TEXT
action_flag INTEGER
change_message TEXT
content_type_id INTEGER
user_id INTEGER
<unnamed> (id)
django_content_type
django_migrations
Debug
Debugger
Console
Frames
Variables
Watches
MainThread
test_index_view_with_a_future_questi
longMessage = {bool} False
maxDiff = {int} 640
reset_sequences = {bool} False
serialized_rollback = {bool} False
startTime = {datetime} 2015-10-09 11:38:35.521452
self.maxDiff = {int} 640
self.startTime = {datetime}... View
4: Run
5: Debug
6: TODO
Python Console
Terminal
9: Version Control
manage.py@first_steps
Event Log
Tests Failed: 4 passed, 3 failed (4 minutes ago)
34:9
LF
UTF-8
Git: master

IP[y]: Notebook analysis Last Checkpoint: Mar 09 16:10 (autosaved)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

In [56]:
```
def average_min_dist(df):
    '''Takes a dataframe and calculates the average min distance between the two agents'''

    p1 = df[df['agent'] == 1]
    p2 = df[df['agent'] == 2]
    # get distances and add a small random value
    dists = cdist(p1[['x', 'y']], p2[['x', 'y']], 'euclidean') + (np.random.rand(150, 150) / 1000)

    min_dists = []
    for i in range(150):
        indices = np.unravel_index(np.argmin(dists), (150,150))
        min_dists.append(dists[indices])
        dists[indices[0],:] = 1000
        dists[:,indices[1]] = 1000

    return np.mean(min_dists)
```

In [67]:
```
def get_values(filename):
    df = pd.read_csv(filename, delim_whitespace=True, names=['x', 'y', 'found_val', 'actual_val', 'agent'])
    min_dist = average_min_dist(df)
    score = df['actual_val'].sum()
    return min_dist, score
```

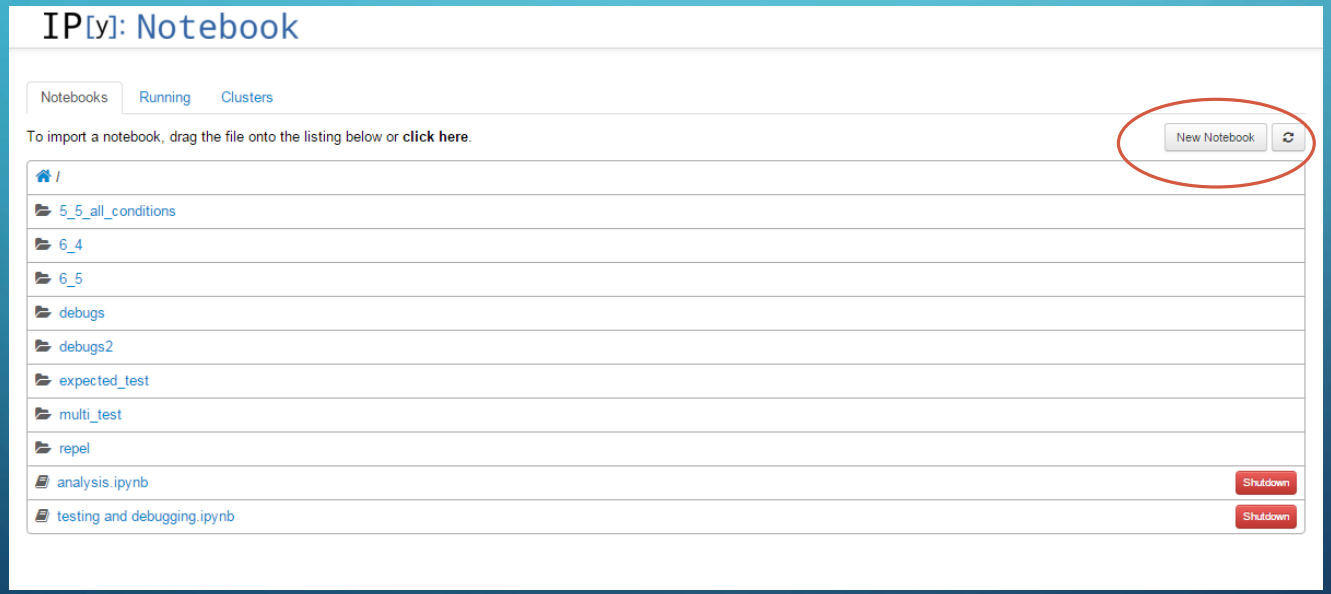
In [35]:
```
average_min_dist('gamma5.5-beta0.125-clust1-res100-map100-independent.txt')
```

Out[35]: 19.001172316142615

In [62]:
```
df_dict = {'map_num':[], 'clustering':[], 'num_stars':[], 'condition':[], 'min_dist':[], 'score':[]}
filenames = [f for f in os.listdir('.') if f[-4:] == '.txt']
for file_name in filenames:
    cond = re.search('(independent)|(repel)|(attract)|(follow)', file_name).group(0)
    clustering = re.search('(clust)([0-9])' ,file_name).groups()[1]
    num_stars = re.search('(res)([0-9]{3,4})' ,file_name).groups()[1]
    map_num = re.search('(map)([0-9]{1,3})', file_name).groups()[1]
    min_dist, score = get_values(file_name)

    df_dict['map_num'].append(int(map_num))
    df_dict['clustering'].append(clustering)
    df_dict['num_stars'].append(int(num_stars))
    df_dict['condition'].append(cond)
    df_dict['min_dist'].append(min_dist)
    df_dict['score'].append(int(score))

df = pd.DataFrame(df_dict)

```

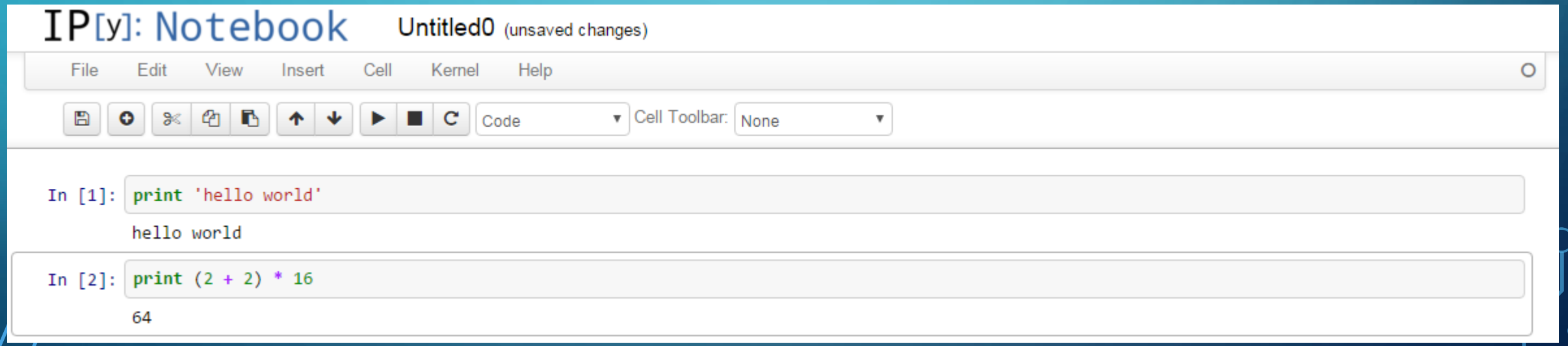
IPYTHON/JUPYTER

- Start a command prompt in the folder you want to work in
- Type "`jupyter notebook`"



IPYTHON/JUPYTER

- Start a new notebook and we're ready to go.



VARIABLES

- How to assign a variable:
  - x = 3
  - y = 7.8
  - favorite_cheese = ‘brie’
- Variables **should** start with a lower case letter. Case matters. Use under_scores instead of CamelCase in python

INDENTATION

- White space matters!
- Use 4 spaces for tabs in python
- Most editors will do this automatically
- Check out all of the other style guidelines in PEP8
  - http://legacy.python.org/dev/peps/pep-0008/

CONTROL FLOW IN PYTHON

- Conditionals
  - The mighty `if` statement
  - Checks the truth value of a statement

```
if 5 > 3:
    print( '5 is greater!' )
elif 5 < 3:
    print( '5 is lesser!' )
else:
    print( 'They're the same!' )
```

LOOPS

- They loop over stuff

```
for i in range(0,10):
    print( i )
```

LOOPS

- Loops in python loop over each item. Not just a series of numbers
  - akin to foreach in other languages
  - example:

```
<-
fruits = ['apple', 'banana', 'pear']
for item in fruits:
    print(item)
->
apple
banana
pear
```

LOOPS

- enumerate() is handy if you want to keep track of the loop iterator

```
<-
for i, item in enumerate(fruits):
    print(i, item)
->
(0, 'apple')
(1, 'banana')
(2, 'pear')
```

FUNCTIONS

- Functions help keep your code neat and readable.
- Any frequently reused bit of code should probably be turned into a function
- If you are copy and pasting code you should really, REALLY consider a function

FUNCTIONS

- Functions take some number of arguments as input
- perform an operation
- Return an output

FUNCTIONS

```
def multiply(input1, input2):
'''Three quotes can be used to create docstrings.
  These are a special kind of comment that are
generally used at the start of a function to document
how it works '''
    out = input1 * input 2
    return out
```

FUNCTIONS

```
<- multiply(2, 5)
-> 10


<- for i in range(3):
<-     print( multiply(i, 2) )
->
   0
   2
   4
```

DATA TYPES

- Variable typing in python is implicit
  - You do not have to declare the type of the variable
- This can lead to issues when a variable ends up being a type you don't expect.
  - If you find yourself with a weird bug you can't figure out, makes sure your variable is the type you expect
  - It is important to know your types!

IMPORTANT DATA STRUCTURES

In python variables will automatically be typed based on their contents

- Integers
  - Numbers that do NOT have a decimal
    - 0
    - 1
    - 5
    - 152

IMPORTANT DATA STRUCTURES

- Floating Point Numbers or Floats
  - Numbers that do have a decimal
    - 5.5
    - 0.12342
    - 3.1459
    - 1.0

IMPORTANT DATA STRUCTURES

- Strings
  - Contains text. Should be surrounded by either single or double quotes
  - string1 = ‘hello’
  - string2 = “goodbye”
- Data read in from a text file will be made up of strings

SOME USEFUL STRING FUNCTIONS

>>>x = 'hello WORLD'

- lower() : makes all letters lowercase

  x.lower() -> 'hello world'

- upper() : makes all letters lowercase

  x.upper() -> 'HELLO WORLD'

- zfill(amount_of_padding) : pads a string made up of digits with the given number of zeros

  '15'.zfill(4) -> '0015'

- split() : will split a string into a list separated by white space

  x.split() -> ['hello', 'world']

- split(separator) : will split a string into a list using the given separator

  x.split('R') -> ['hello WO', 'LD']

CONVERTING BETWEEN TYPES

- You will often need to convert between different types.
  - E.g. if you read numbers from a text file you may want to convert them from strings to ints
- Functions that will convert **to** their respective types
  - int(x)
  - float(x)
  - str(x)
- `int('2') -> 2`
- `float(2) -> 2.0`
- `str(2) -> '2'`

IMPORTANT DATA STRUCTURES

- Lists contain ordered groups of things. They are declared with [and]
  - List1 = [1, 2, 5, 20]
  - List2 = [‘b’, 6, ‘brie’]
- You can access items in lists based on their location

```
>>>List1[2]
5
>>>List2[0]
'b'
```

LISTS

- Creating empty lists
  - List1 = []
  - List1 = list()
- Adding items to lists
  - append() will add an object to the end of the list
  - List1.append('monkey')

LISTS

- Example:

```
>>>list1 = [ ]
>>>print list1
[]

>>>list1.append(5)
>>>print list1
[5]

>>>list1.append(56)
>>>print list1
[5, 56]

>>>print list1[0]
5
```

IMPORTANT DATA STRUCTURES

- Tuples
  - Tuples are sequences of values separated by commas
  - They are also generally surrounded by ( )
  - tuple1 = (1, 3)
  - Tuple2 = (‘bob’, ‘jane’, ‘joe’)
  - Tuples are immutable
    - Individual values in a tuple cannot be changed
  - Generally used for returning multiple values at the same time

IMPORTANT DATA STRUCTURES

- Dictionaries or “dicts”
  - Unordered containers made up of key/value pairs
  - You access a value by using the key

```
>>>name_count_dict = {‘joe’:3, ‘bob’:2, ‘mary’:1, ‘sue’:2}
>>>print name_count_dict[‘bob’]
2
>>>name_count_dict[‘sue’] = name_count_dict[‘sue’] + 1
>>>print name_count_dict
{'bob': 2, 'joe': 3, 'mary': 1, 'sue': 3}
```

IN KEYWORD

- The in keyword can frequently be useful when dealing with containers (e.g. lists, tuples, dictionaries
- Use in to check if a value exists inside the container

```
<- list1 = [ 2, 6, 12 ]
<- 6 in list1
-> True
<- 4 in list1
-> False
```

```
for i in range(6):
    if i in list1:
        print('Woo!')
    else:
        print('Boo')
```

- Note: For dictionaries in checks if the value is a key

FILE INPUT AND OUTPUT

- Reading Files
  - There are 2 main ways of reading in files
  - Read and act on the file a line at a time
    - This is generally the preferred method
  - Read in the entire contents of the file at once
    - You should generally avoid this method unless you have a specific reason
    - For example you might do this if you want to interact with multiple separate lines at the same time

FILE INPUT AND OUTPUT

- Reading a file one line at a time
  - First you need to open a “file handle” for the file you which to read.
  - Then you loop over the file object. Python automatically loops a line at a time on a file object

```
# open a file handle to 'filename.txt' in read only mode, with f as the variable for the handle
with open('filename.txt', 'r') as f:
    for line in f: # this line will start looping over each line of the file
        print( line ) # here you would start acting on the current line
```

FILE INPUT AND OUTPUT

- Reading in an entire file
  - Once you have a file handle you can read the file
  - x = f.read() will read the entire file in as one big string into x
  - x = f.readlines() will read all the lines in the file into a list

```
with open('filename.txt') as f:
    list_of_lines = f.readlines()
print( list_of_lines )
```

EXAMPLE

- Read in the third word from each line in a file named ‘gruyere.txt’

```
third_words = [] #create an empty list that we will put the words into
with open('gruyere.txt') as f:
    for line in f:
        split_line = line.split() # split the line into pieces based on white space
                if len(split_line) >= 3: # check to make sure there are at least 3 words.
                        third_words.append( split_line[2] )
```

EXERCISE 1

roster.txt contains 3 columns: first name, last name, score

Each column is separated by a tab (‘\t’)

- Goal: create a dictionary that has the first letter of each person’s first name as the keys, and the sum of scores from each person whose name starts with that letter as the value
- E.g. score_dict[‘a’] should return the sum of scores from every person whose first name starts with the letter a

- Hint: You can access a specific character in a string by indexing it the same way you would a list. string1[2] would get you the third character in string1

HINT

```
score_dict = {}
with open('roster.txt') as f:
    for line in f:
        split_line = line.split()
        letter = split_line[0][0]
```

ANSWER

'A': 1,

'B': 1,

'C': 8,

'D': 1,

'E': 3,

'F': 1,

'I': 1,

'J': 4,

'K': 3,

'L': 5,

'M': 1,

'P': 2,

'R': 3,

'S': 2,

'T': 1,

'V': 1

MY SOLUTION

```
score_dict = {}
with open('roster.txt') as f:
    for line in f:
        split_line = line.split()
        letter = split_line[0][0]
        if letter in score_dict:
            score_dict[letter] += float(split_line[2])
        else:
            score_dict[letter] = float(split_line[2])
```

MY SOLUTION (AS A FUNCTION

```
def letter_counter(fname):
    ''' Get the number of times first names start with each letter.
        This would be more useful if used frequently '''
    score_dict = {}
    with open(fname) as f:
        for line in f:
            split_line = line.split()
            letter = split_line[0][0]
            if letter in score_dict:
                score_dict[letter] += 1
            else:
                score_dict[letter] = 1
    return score_dict


letter_counter('roster.txt')
```

LET'S GET SOME SCIENCE UP IN HERE

- There are a couple important libraries that enable scientific computing in python
  - Numpy
    - Adds fast numerical arrays to python
    - Lets you work in vectors
  - Scipy
    - Gives access to a number of common scientific function
    - Includes basic stats, interpolation, some other useful stuff
  - Matplotlib
    - Enables graphical plotting

WHAT IS A LIBRARY AND WHAT DOES IT DO?

- Libraries are external packages of code that do stuff.
- To access a library you need to `import` it. (Assuming you've installed it)

- Import statements should generally all go at the top of your code.
  - Example ways of importing:
    - `import numpy`
    - `import numpy as np`
    - `from numpy import mean`

LET'S GET TO KNOW NUMPY

NUMPY

- Numpy gives access to the ndarray
  - A multi dimensional array
  - enables math operations on blocks of data
  - If you've used matlab this should be familiar
- How to import numpy:

```
import numpy as np
```

NUMPY

```
data = np.array([0.9526, -0.246, -0.8856],
                                [0.5639, 0.2379, 0.9104] )
<- data * 10
-> array([[9.526, -2.46, -8.856], [5.639, 2.379, 9.104]])

<- data.shape()
-> (2, 3)
```

- Each array should be a single type

<- data.dtype

-> dtype('float64')

MAKING ARRAYS

- Make arrays by using np.array()
  - np.array([1,2,5,7])

- Multidimensional arrays are created by using nested structures
  - E.g. a list of lists will create a 2d array
  - data2 = [[1,2,3,4], [5,6,7,9]]
  - arr2 = np.array(data2)
  - arr2.shape -> (2,4)

MAKING ARRAYS

- <- np.zeros(5)
- -> array([0., 0., 0., 0., 0.])

- <- np.ones((2,4))
- -> array([[1., 1., 1., 1.], [1., 1., 1., 1.]])

- <- np.arange(5)
- -> array([0, 1, 2, 3, 4])

CONVERTING ARRAY TYPES

```
<- arr = np.array([1,2,3,4,5])
<- arr.dtype
-> dtype('int64')

<- float_arr = arr.astype(np.float64)
-> array([1., 2., 3., 4., 5.])

<- float_arr.dtype
-> dtype('float64')
```

INDEXING AND SLICING

- Note: Lists can be indexed in the same way as 1d arrays

- Indexing:
  - For 1d arrays indexing is very simple

```
<- arr = np.array([5,6,7,8,9])
<- arr[2]
-> 7
<- arr[0]
-> 5
```

ARRAY INDEXING

- 2d Arrays can be indexed by separating each axis by a comma (,)

```
<- arr2d = np.array([[0,1,2], [3,4,5]])
<- arr2d[0,1]
-> 1

<-arr2d[1,2]
-> 5
```

Axis 1

	0	1	2
0	0,0	0,1	0,2
1	1,0	1,1	1,2

Axis 0

ARRAY SLICING

- Slices allow you to select portions of a list or array instead of single values
- slice format
  - arr[a:b:c]
    - a is start point, b is end point, c is step size

```
<- arr = np.array([0,1,2,3,5,6])

<- arr[1:4]
-> array([1,2,3])

<- arr[1:4:2]
-> array([1,3])
```

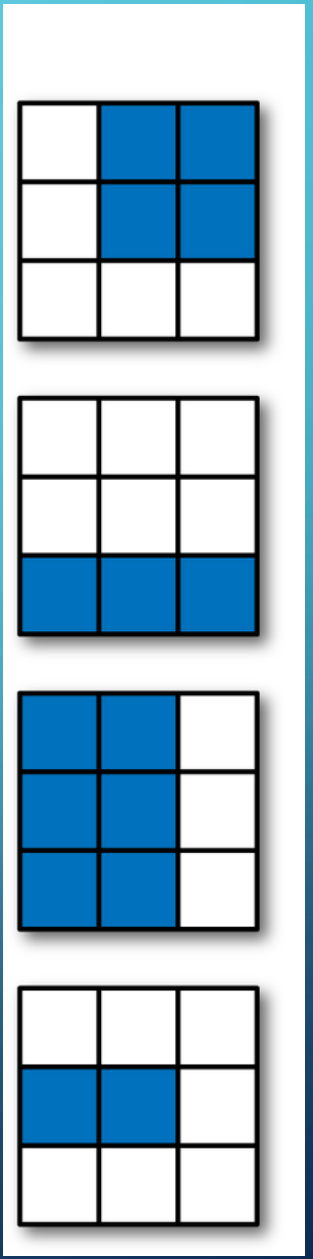
SLICING

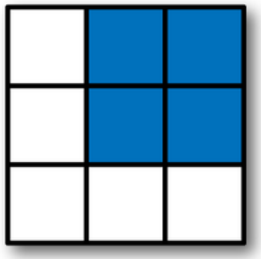
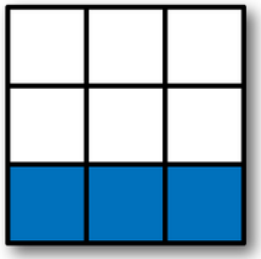
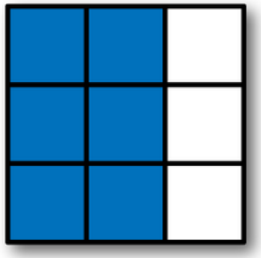
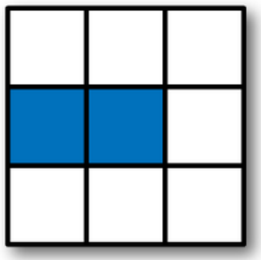
- Examples:
- [:5]
  - Beginning to the fifth value
- [3:]
  - From the 3rd value to the end
- [3:-2]
  - from the 3rd value, to the second to last
- [3:6:-1]
  - From the 3rd value to the 6th value backwards

EXERCISE 2

- What index should be used to generate each blue slice?
- What shape will each slice be?

- Feel free to create an array and experiment



	Expression	Shape
	`arr[:2, 1:]`	`(2, 2)`
	`arr[2]`	`(3,)`
	`arr[2, :]`	`(3,)`
	`arr[2:, :]`	`(1, 3)`
	`arr[:, :2]`	`(3, 2)`
	`arr[1, :2]`	`(2,)`
	`arr[1:2, :2]`	`(1, 2)`

SOME USEFUL NUMPY FUNCTIONS

- Statistics:
  - sum
  - mean
  - std, var
  - min, max
  - cumsum

EXAMPLE

```
<- arr = np.array([2,2,3,4,5])
<- np.mean(arr)
-> 3.200000

<- arr.mean()
-> 3.200000
```

OTHER USEFUL FUNCTIONS

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating point, or complex values. Use fabs as a faster alternative for non-complex-valued data
sqrt	Compute the square root of each element. Equivalent to arr ** 0.5
square	Compute the square of each element. Equivalent to arr ** 2
exp	Compute the exponent e^x of each element
log, log10, log2, log1p	Natural logarithm (base e), log base 10, log base 2, and log(1 + x), respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
ceil	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
floor	Compute the floor of each element, i.e. the largest integer less than or equal to each element
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as separate array
isnan	Return boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf	Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
cos, cosh, sin, sinh, tan, tanh	Regular and hyperbolic trigonometric functions
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverse trigonometric functions

PANDAS

- A fast and efficient **DataFrame** object for data manipulation with integrated indexing;
- Tools for **reading and writing data** between in-memory data structures and different formats: CSV and text files, Microsoft Excel, SQL databases, and the fast HDF5 format;
- Intelligent **data alignment** and integrated handling of **missing data**: gain automatic label-based alignment in computations and easily manipulate messy data into an orderly form;
- Flexible **reshaping** and pivoting of data sets;
- Intelligent label-based **slicing, fancy indexing**, and **subsetting** of large data sets;
- Columns can be inserted and deleted from data structures for **size mutability**;
- Aggregating or transforming data with a powerful **group by** engine allowing split-apply-combine operations on data sets;
- High performance **merging and joining** of data sets;
- **Hierarchical axis indexing** provides an intuitive way of working with high-dimensional data in a lower-dimensional data structure;
- **Time series**-functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging. Even create domain-specific time offsets and join time series without losing data;
- Highly **optimized for performance,**

PANDAS

- Pandas (from Panel Data) is the tool that makes data analysis in python great.

- To use it:
  - `import pandas as pd`

PANDAS SERIES

- There are two very important data structures in pandas
- the first is pd.Series()
  - A series is a 1d array with an associated array of indexes
- `<-  pd.Series([4,7,-3,6])`
- `->`

```
0    4
1    7
2   -3
3    6
dtype: int64
```

```
<-   dat = pd.Series([4,7,-3,6])
<- dat[dat > 4]
->
   1    7
   3    6
       dtype: int64
```

```
<- dat * 2
->
0       8
1      14
2      -6
3      12
dtype: int64
```

INDEXES CAN BE ANYTHING

- Useful when working with time series data
  - Set the timestamp as the index
- Useful to label specific points

```
<- state_scores = pd.Series([5, 1, 3, 7], index=['CA', 'MA', 'IL', 'FL'])
->
    CA    5
    MA    1
    IL    3
    FL    7
    dtype: int64
```

ACCESS ITEMS BY INDEX

```
<- state_scores['FL']
-> 7


<- state_scores[ ['CA', 'FL'] ]
->
   CA    5
   FL    7
   dtype: int64
```

DATAFRAME

- The dataframe is a spreadsheet-esque data structure
  - Essentially a collection of columns
  - Each column can be a different data type (eg int, float, string, etc)
  - Similar to R data.frame
  - The magic glue that makes data analysis in Python easy (relatively)

CREATING A DATAFRAME

- The easiest way to make a dataframe is from a dictionary of equal length lists

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}

frame = pd.DataFrame(data)
```

CREATING A DATAFRAME

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

CREATING A DATAFRAME EXAMPLE

- Often you will need to do some level of preprocessing your data.
- If you want generate your dataframe as you process using a dictionary of lists

CREATING A DATAFRAME EXAMPLE

- Example: For some reason I want to make a dataframe that contains the first, fifth, and last character on each line in a file

```
# The keys will become column names
df_dict = { 'first':[], 'fifth':[], 'last':[] }
with open(file_name, 'r') as f:
    for line in f:
        df_dict['first'].append(line[0])
        df_dict['fifth'].append(line[4])
        df_dict['last'].append(line[-1])
```

DATAFRAME

- DataFrames have tons and tons of features. I am only going to highlight a few of them that I use most frequently
  - Check out the pandas documentation: http://pandas.pydata.org/pandas-docs/stable/

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

ACCESSING COLUMNS

```
<- frame['year']

->

   0    2000
   1    2001
   2    2002
   3    2001
   4    2002
```

ACCESSING COLUMNS

<- frame[['year', 'state']]

->

	year	state
0	2000	Ohio
1	2001	Ohio
2	2002	Ohio
3	2001	Nevada
4	2002	Nevada

ACCESSING ROWS

- .ix[ ] will access rows by their index

```
<- frame.ix[2]
->
pop          3.6
state       Ohio
year        2002
```

```
<- frame.ix[2:4]
->
   pop    state  year
2  3.6     Ohio  2002
3  2.4   Nevada  2001
4  2.9   Nevada  2002
```

ACCESSING ROWS

- .iloc[ ] will access rows by their position. This is different from .ix[ ] which accesses by the index which does NOT have to be the same as the position
- Example:

```
<- temp_frame = frame.drop(3) # drops the row labeled 3
<- frame.iloc[3]
->
    pop          2.4
    state     Nevada
    year        2001
```

```
<- temp_frame.iloc[3]
->
    pop          2.9
    state     Nevada
    year        2002
```

ACCESSING ROWS & COLUMNS

- .loc[ ] will allow you to access both row labels (indices) and column labels
- Example:

```
<- frame.loc[2, 'year']
-> 2002
```

ACCESSING ROWS & COLUMNS

- There are loads of fancy things you to index and slice your dataframe
- See this page for all sorts of cool tricks:
  - http://pandas.pydata.org/pandas-docs/stable/indexing.html

ADD A NEW COLUMN

```
<- frame['debt'] = [2.1, 3.3, 4.1, 1.4, 2.6]
->
   pop    state  year  debt
0  1.5     Ohio  2000   2.1
1  1.7     Ohio  2001   3.3
2  3.6     Ohio  2002   4.1
3  2.4   Nevada  2001   1.4
4  2.9   Nevada  2002   2.6
```

ADD A NEW COLUMN

```
<- frame['debt_per_person'] = frame['debt'] / frame['pop']
->
```

	pop	state	year	debt	debt_per_person
0	1.5	Ohio	2000	2.1	1.400000
1	1.7	Ohio	2001	3.3	1.941176
2	3.6	Ohio	2002	4.1	1.138889
3	2.4	Nevada	2001	1.4	0.583333
4	2.9	Nevada	2002	2.6	0.896552

THE SECRET TO GOOD DATA ANALYSIS CODE:

- **Think in terms of vectors**
  - Many, many problems can be solved with vector operations
  - Ask yourself: "Am I applying some kind of function to each row or column in my data?"
    - If the answer is yes you should NOT be using loops
    - The answer will almost always be yes.

THINK IN TERMS OF VECTORS

- The apply() function is invaluable for applying a function across each row or column in your data

THINK IN TERMS OF VECTORS

- Task: Find the first two letters of each state and prepend it to it's population to create an id code

```
codes = [ ]
for row in frame.iterrows():
    code = '{}{}'.format(row[1]['state'][:2],
row[1]['pop']
    codes.append(code)
```

THAT IS BAD. DO NOT DO IT.

- Think in vectors. Not loops.
- Instead design a function that will do what you want to a single row

THINK IN VECTORS

```
def create_code(row):
    code = '{}{}'.format(row['state'][:2], row['pop'])
    return code
```

- You can test your function by calling it on a specific row

```
<- create_code(frame.ix[2])
-> 'Oh3.6'
```

APPLY()

- You can call apply from either a dataframe or a series
- apply() takes several important arguments:
  - func – the function you want to apply
  - axis –
    - 0: apply function to each column
    - 1: apply function to each row
  - args –
    - an iterable containing any additional arguments to passed to the function

APPLY()

```
def create_code(row):
    code = '{}{}'.format(row['state'][:2], row['pop'])
    return code


<- frame.apply(create_code, axis=1)
->
0    Oh1.5
1    Oh1.7
2    Oh3.6
3    Ne2.4
4    Ne2.9
```

THINK IN VECTORS

- Always ask yourself: How can I solve this problem without a giant loop?
  - Most of the time there's a way, and it will be both faster, and easier to read

USING APPLY()

- apply() can also be used with existing functions from other places

```
frame['pop'].apply(np.sum)
```

  - Note: This is equivalent to frame['pop'].sum()

- You can also drop your result right into your dataframe
  - `frame['codes'] = frame.apply(create_codes, axis=1)`

READING AND WRITING FILES

- Pandas has very useful functions for reading and writing preformatted files of various kinds
- http://pandas.pydata.org/pandas-docs/stable/io.html

read_csv

read_excel

read_hdf

read_sql

read_json

read_html

read_stata

read_sas

read_clipboard

read_pickle

READING FILES

- pd.read_csv() is easy to use, but also takes a LOT of arguments that give a great deal of flexibility in file formats as well
  - Shift + Tab in the jupyter notebook is great for a quick documentation check
- For a standard csv file:

```
df = pd.read_csv(file_name)
```

QUICK EXERCISE

- Open roster.csv as a DataFrame
- Calculate the mean and standard deviation of the Grade column

MY SOLUTION

```
df = pd.read_csv('roster.csv')

print( df['Grade'].mean(), df['Grade'].std() )
```

READING FILES

- Now lets try opening roster.txt from earlier instead
  - tab delimited instead of comma
  - no header row
- read_csv can be used with most .txt files

```
df = pd.read_csv('roster.txt', header=None, names=['First', 'Last', 'Grade'], delim_whitespace=True)
```

WRITING FILES

to_csv

to_excel

to_hdf

to_sql

to_json

to_html

to_stata

to_clipboard

to_pickle

WRITING FILES

- Oops! Forgot to add the extra credit to the grades.
- Let's add 1 to every grade, then save over roster.csv

```
df['Grades'] += 1
df.to_csv('roster.csv')
```

- Note: Writing methods are called from the dataframe you are trying to save.

READING AND WRITING FILES

- All of the various file formats can be useful based on your needs.
- Personally I use csv files frequently
  - Easy to read, cross compatible with nearly everything
- I also frequently use pickle
  - pickle is a python specific binary format
  - Pros: When you load your dataframe it will be exactly how you left it. Can handle complex nested data structures (e.g. arrays nested inside cells)
  - Cons: Low compatibility. Not guaranteed to function if your data structures change (e.g. a new version of numpy changes an important part of how arrays work under the hood.
  - If you use pickle, make sure you have your data accessible in another format as well.

FILTERING DATA

- DataFrames can be filtered using "masks" of boolean series
- Example: We want to look at only the students with grades greater than 70.0
- `df['Grade'] > 70` will create a Series which indicates which rows that statement is true for, and which ones its false

0	True
1	True
2	True
3	True
4	False
5	True
6	True
7	True
8	False
9	True
10	True
11	True
12	False
13	True
14	True
15	True
16	False
17	True
18	True
19	True
20	True
21	True
22	False
23	False
24	False
25	True
26	True
27	True
28	True
29	False
30	False
31	True
32	False
33	True
34	True
35	False
36	True
37	True

FILTERING DATA

```
good_students_df = df[ df['Grade'] > 70 ]
```

- Creates a view of the dataframe containing only those entries where grade > 70

- Note: If you edit this view, the changes will effect the original dataframe as well
- Use .copy() if you need to avoid this
  - `good_students_df = df[ df['Grade'] > 70 ].copy()`

MULTIPLE FILTERS

- You can apply multiple filters simultaneously
  - There is a special syntax
  - Each conditional must be in parentheses
  - Use `&` or `|` instead of the typical `and` or `or`
- `Example:`

```
avg_stdnts_df = df[ (df['Grade'] > 70) & (df['grade'] < 80) ]
```

READING FILES

- Let's open some real data!
- First we need to move into the PBRdata folder.
  - The built in os module contains utilities including file system navigation

```
import os
os.chdir('PBRdata')
```

READING FILES

- Let's open the file in excel first and check for any weirdness
- Take a look at Control_left.csv

READING FILES

1	TRIAL BY TRIAL DATA																
2	subject	trial	stimfile	condition	code	resp_1	resp_2	response	distractor	error	init time	RT	MD_1	MD_2	AUC_1	AUC_2	M
3	1	8	SPRUCE	2	b1_con_le	WORD	NONWOR	2	2	1	109	1653	1.3388		2.9365		
4	1	12	WRENCH	1	b1_con_le	WORD	NONWOR	1	2	0	203	1638		0.2959		0.0667	
5	1	15	NAIL	1	b1_con_le	WORD	NONWOR	1	2	0	109	1420		0.8443		1.4629	
6	1	19	MONTH	1	b1_con_le	WORD	NONWOR	1	2	0	31	1482		0.2231		0.239	
7	1	22	OKRA	2	b1_con_le	WORD	NONWOR	2	2	1	46	1388	0.314		0.3126		
8	1	24	OBOE	2	b1_con_le	WORD	NONWOR	2	2	1	343	2106	0.3706		0.3115		
9	1	29	DECADE	1	b1_con_le	WORD	NONWOR	1	2	0	78	1794		0.2643		0.315	
10	1	30	YARD	1	b1_con_le	WORD	NONWOR	1	2	0	109	1435		0.443		0.4174	

READING FILES

7	109	TIGER	1	b2_con_le	WORD	NONWOR	1	2	0	421	1154		0.1714		0.1647	560	4
7	112	JACKET	1	b2_con_le	WORD	NONWOR	1	2	0	234	1482		1.0882		2.5071	792	10
7	113	LINEN	1	b2_con_le	WORD	NONWOR	1	2	0	312	1607		0.7874		1.4595	907	11
7	114	SILK	1	b2_con_le	WORD	NONWOR	1	2	0	359	1248		0.2664		0.2961	581	5
7	115	RED	1	b2_con_le	WORD	NONWOR	1	2	0	125	1139		0.2175		0.2535	564	4
7	118	HORSE	1	b2_con_le	WORD	NONWOR	1	2	0	500	1248		0.1384		0.0758	593	7
7	120	MACE	1	b2_con_le	WORD	NONWOR	1	2	0	187	1123		0.1634		0.1853	667	9
7	125	ARTICLE	1	b2_con_le	WORD	NONWOR	1	2	0	265	1669		0.0969		0.0462	843	9
7	127	RULER	1	b2_con_le	WORD	NONWOR	1	2	0	561	1310		-0.1038		-0.0661	856	6
7	128	TIGER	1	b2_con_le	WORD	NONWOR	1	2	0	140	1279		0.1731		0.2151	785	9
7	130	ARTICLE	1	b2_con_le	WORD	NONWOR	1	2	0	297	1280		0.2951		0.2589	608	6

MEAN SUBJECT-BY-SUBJECT DATA

subject	condition	incorrect	init time	RT	MD	AUC	MD time	x-flip	y-flip	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
1	1	0	198.2037	1346.222	0.2304	0.3212	610.4259	6.9444	5.4444	0	0	-0.0001	-0.0001	-0.0002	-0.0003	-0.0007	-0.001
107	1	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---
113	1	0	120.4386	1950.053	0.3142	0.3666	994.7018	9.7018	6.4211	0	0	-0.0002	-0.0005	-0.0007	-0.001	-0.001	-0.0009
14	1	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---
25	1	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

READING FILES

- Header is on the second row, not the first

- There's some extra stuff at the end of the file, after the main table

```
df = pd.read_csv('Control_left.csv', skiprows=1, nrows=720)
```

.HEAD() AND .TAIL()

- For large tables .head() and .tail() can be useful for peaking at a dataframe
- .head(x)
  - Shows the first x rows of the dataframe, default *5*
- tail(x)
  - Shows the last x rows of the dataframe, default *5*

GROUPING AND AGGREGATING

- Group By allows you to
  - **split** your data into groups
  - **apply** a function to each group independently
  - **combine** the results into a new dataframe

- http://pandas.pydata.org/pandas-docs/stable/groupby.html

GROUPING AND AGGREGATING

- Usually you will want to group by a certain column
  - There are a number of more advanced ways to use groupby(), check the docs for details

- We'll group by the stimfile column

```
grouped = df.groupby('stimfile')
```

GROUPING AND AGGREGATING

- Groups can be iterated through
  - I mostly find this useful for plotting or debugging

```
for name, group in grouped:
    print( name, group['RT'].mean() )
```

- You can also select specific groups

```
grouped.get_group(2)
```

GROUPING AND AGGREGATING

- You can aggregate and combine your group using the .agg() function
  - .agg() takes a function and applies it to each of your groups
- `grouped.agg(np.mean)`
  - returns a dataframe with the mean for each column
- Many aggregation functions are built into GroupBy objects
  - `grouped.mean()` is equivalent

GROUPING AND AGGREGATING

- As a matter of style it often convenient to skip assigning the GroupBy object to a variable if you will not be using it.
- Example:
    - `grouped = df.groupby('stimfile')`
    - `mean_df = grouped.mean()`
  - vs
    - `mean_df = df.groupby('stimfile').mean()`

QUICK EXERCISE

- What is the mean reaction time (RT) when someone makes an error compared to when they don't?

ANSWER

	error
0	1475.541787
1	1942.730769

MY SOLUTION

```
df.groupby('error')['RT'].mean()
```

QUICK EXAMPLE: TTEST

- We've decided to do a quick significance test on that result
- SciPy has some basic stats functions including t tests

```
from scipy.stats import ttest_ind # ttest for 2 independant samples


grouped = df.groupby('error')
ttest_ind(grouped['RT'].get_group(0), grouped['RT'].get_group(1))
```

or

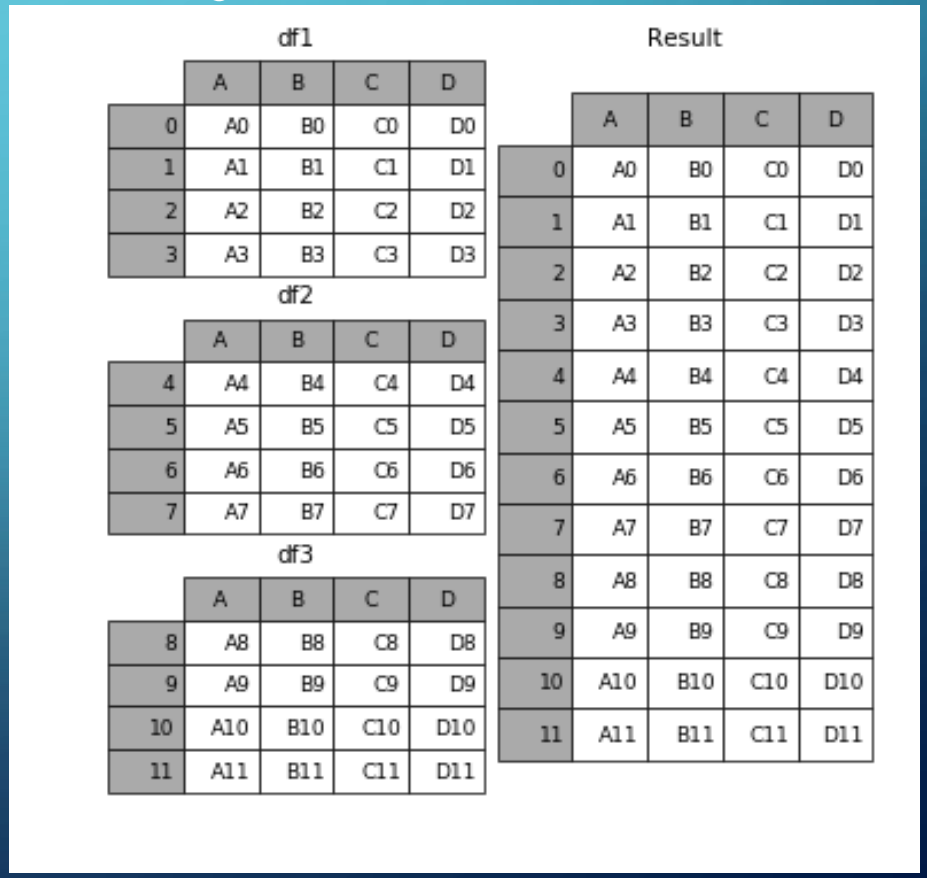
```
ttest_ind( df[ df['error'] == 0], df[ df['error'] == 1] )
```

MERGING DATAFRAMES

- There a wide range of ways to merge together different dataframes
  - I will only cover two of the most common here
  - http://pandas.pydata.org/pandas-docs/stable/merging.html

APPENDING DATAFRAMES

- concat() can be used to concatenate dataframes together
- `result = pd.concat( [df1, df2, df3] )`



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

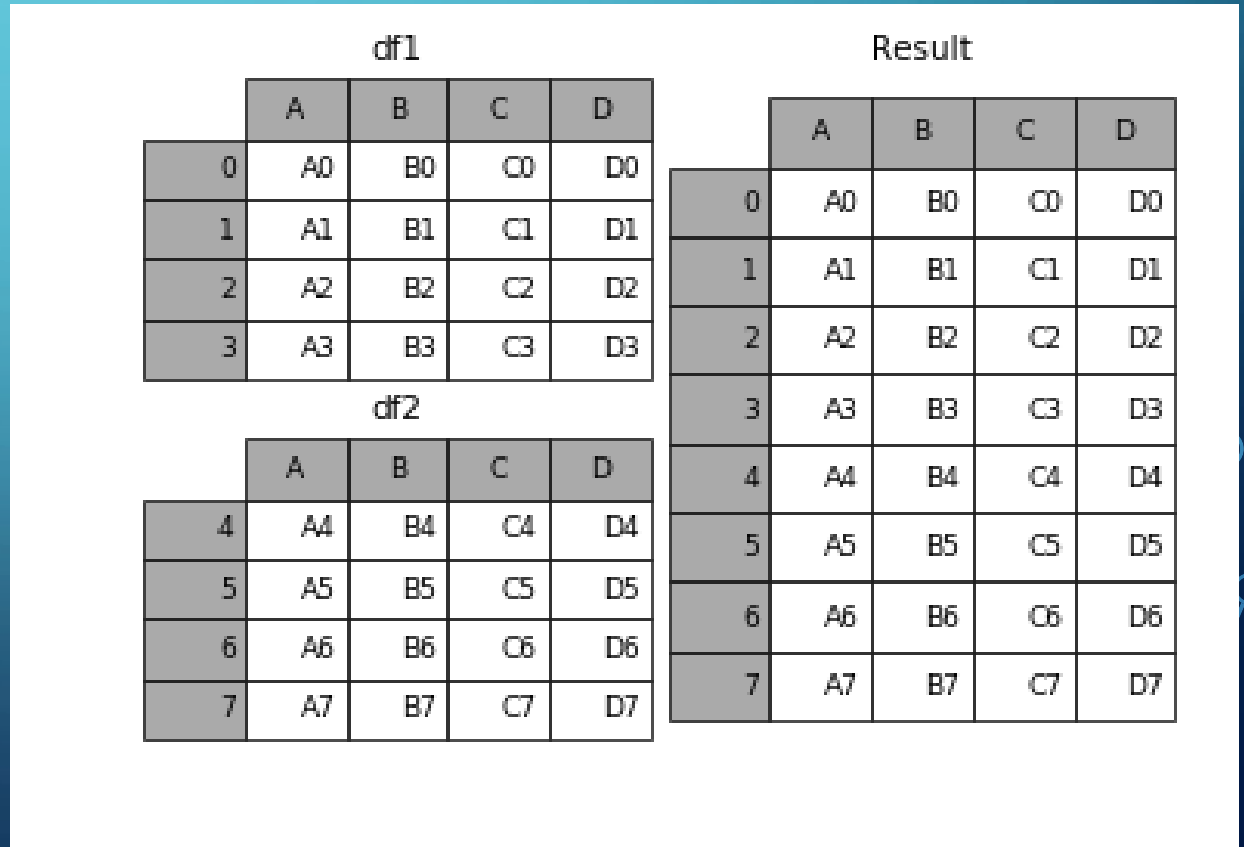
	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

APPENDING DATAFRAMES

- df.append( ) can be used to append one dataframe to the bottom of another
- `result = df1.append(df2)`



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

EXAMPLE

- So far we've only been working with one of the mouse tracking conditions. We have 9 more files of data.
- Let's place all 9 files into the same dataframe
  - We'll use a new column to mark which trial is from what condition

EXAMPLE JUPYTER NOTEBOOK

MERGING COLUMNS

- Pandas is capable of doing very complex database/SQL style joins. Again I will not be covering them here. If you need them, check the docs
- This example demonstrates how to merge to dataframes if they have the same index but contain different columns

MERGING COLUMNS

```
result = pd.merge(left, right, left_index=True, right_index=True, how='outer')
```

left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

Result

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

EXAMPLE

- I prefer to place my data in nested arrays instead of having 200 columns of x and y data
  - Makes my dataframe less busy, and makes accessing and working with the data simpler
  - Note: doing this makes reading and writing CSV files more difficult

EXAMPLE NOTEBOOK

OTHER PANDAS FEATURES

- Cleaning data
  - removing missing values
  - filling in missing values
  - removing duplicates
- Time Series specific manipulation
  - full support for indexing by date or datetime

PLOTTING WITH MATPLOTLIB

- Matplotlib is the primary python plotting library
  - It is extremely powerful, but can be a little weird and tricky at first

- Import matplotlib

```
import matplotlib.pyplot as plt
```

- Activate plotting in the jupyter notebook

```
%matplotlib inline
```

MATPLOTLIB

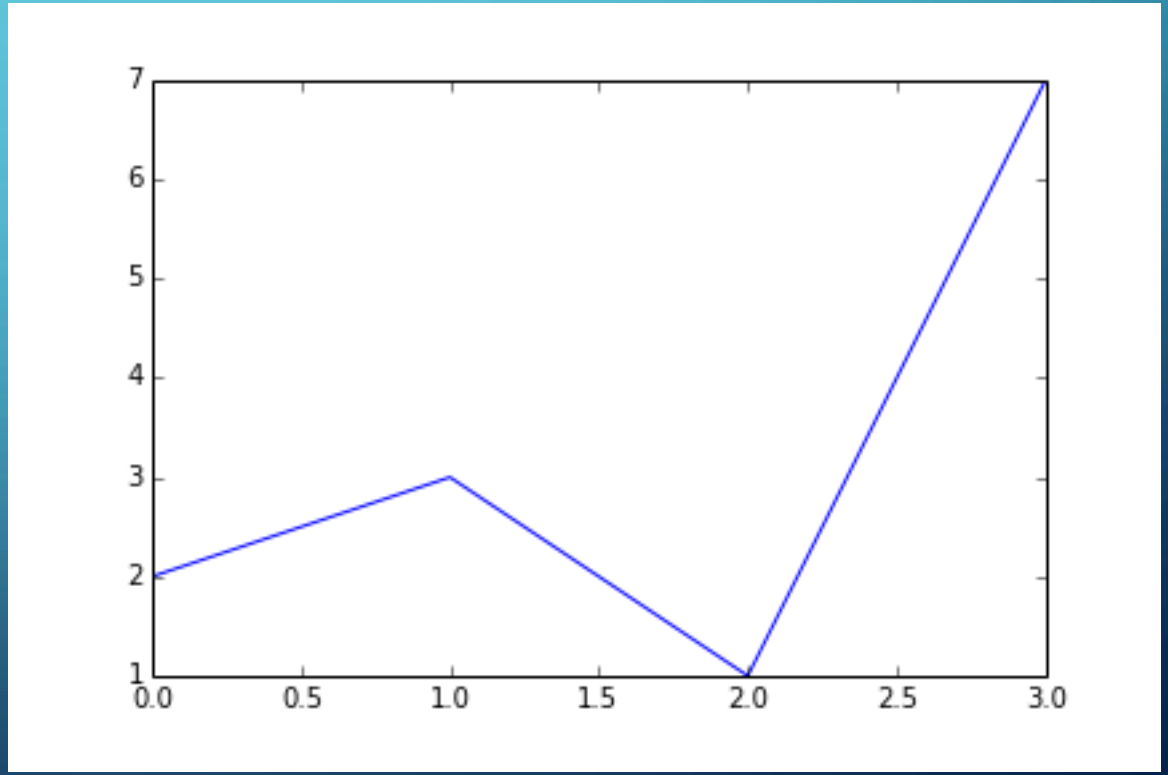
- The pyplot documentation is invaluable and covers many more types of plots than I will discuss here
- http://matplotlib.org/api/pyplot_api.html

PLT.PLOT()

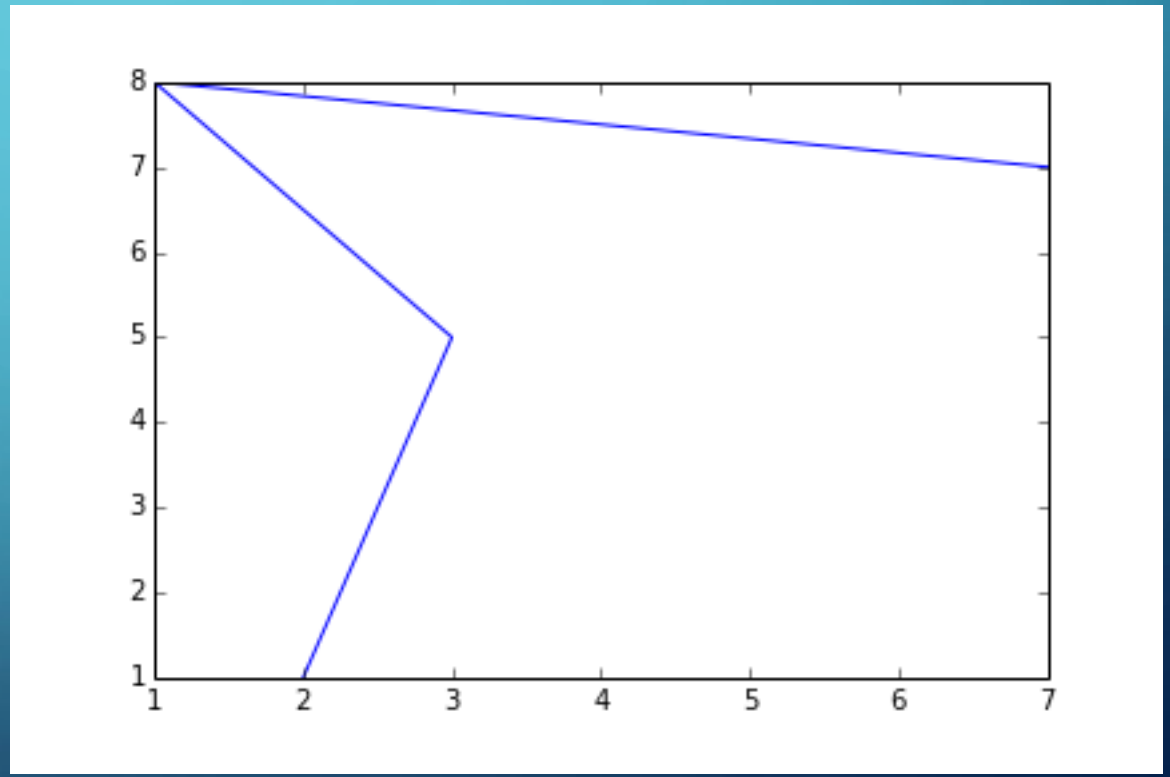
- plt.plot() is matplotlib's main plotting function
  - It is very flexible
  - It is very similar to matlab's plot()

PLT.PLOT()

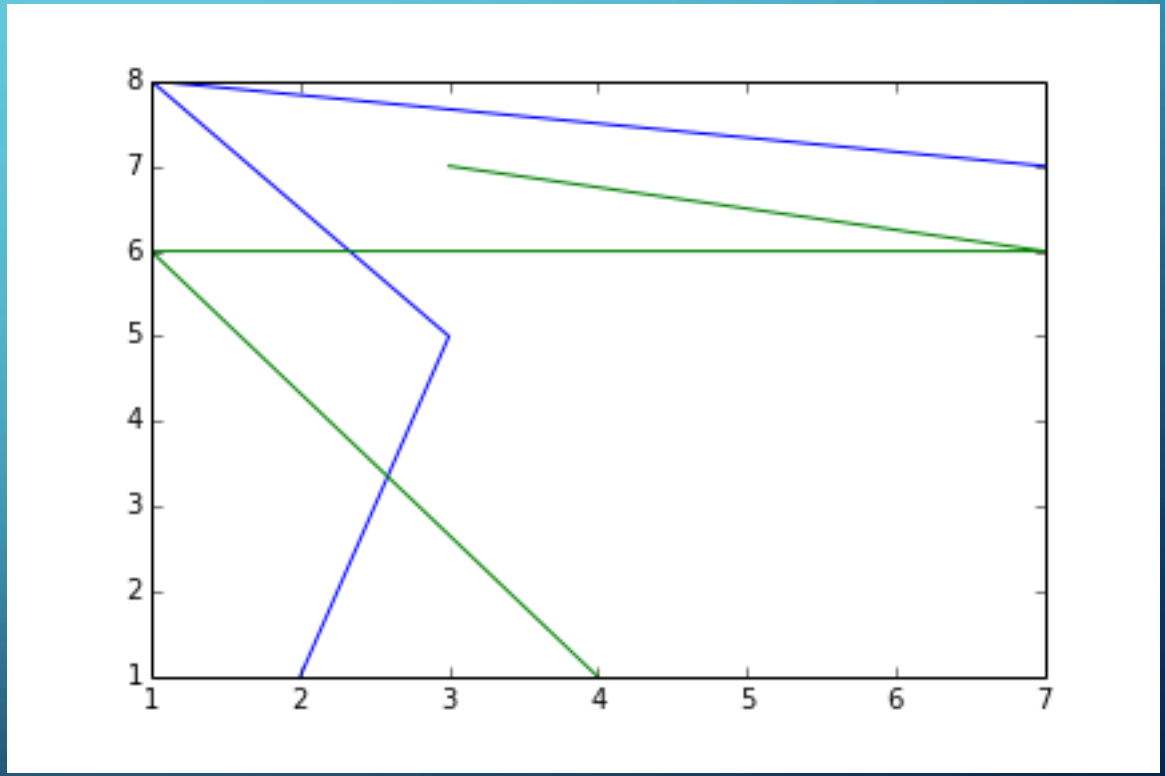
```
x = [2, 3, 1, 7]

plt.plot(x)
```



```
x = [ 2, 3, 1, 7 ]
y = [ 1, 5, 8, 7 ]
plt.plot( x, y )
```

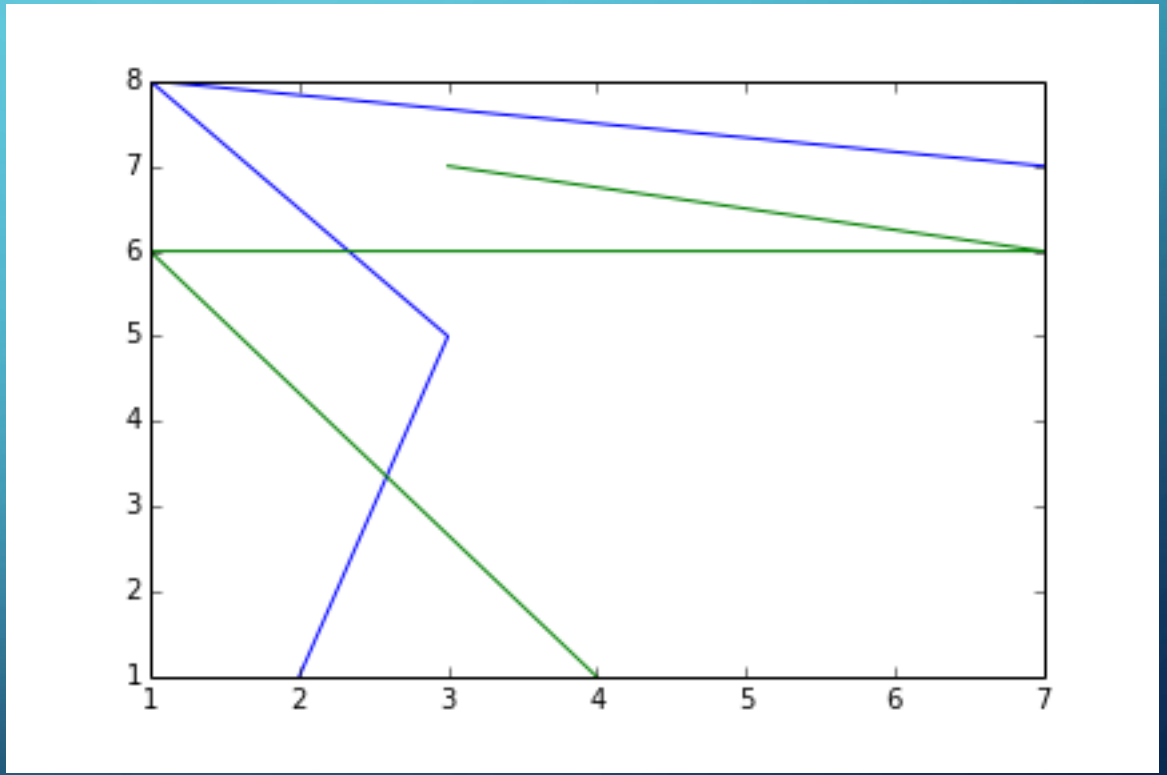


```
x2 =  [3, 7, 1, 4]
y2 = [7, 6, 6, 1]
plt.plot(x, y, x2, y2)
```

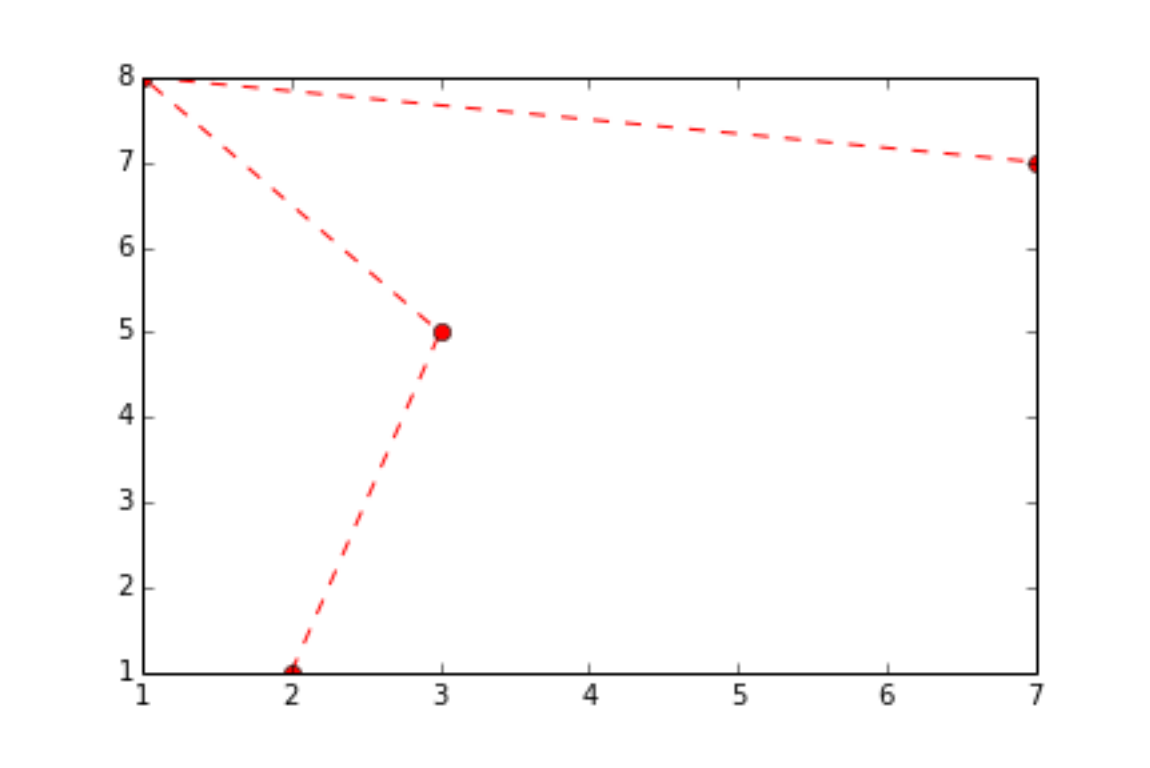


```
plt.plot(x, y)
plt.plot(x2, y2)
```

Same result



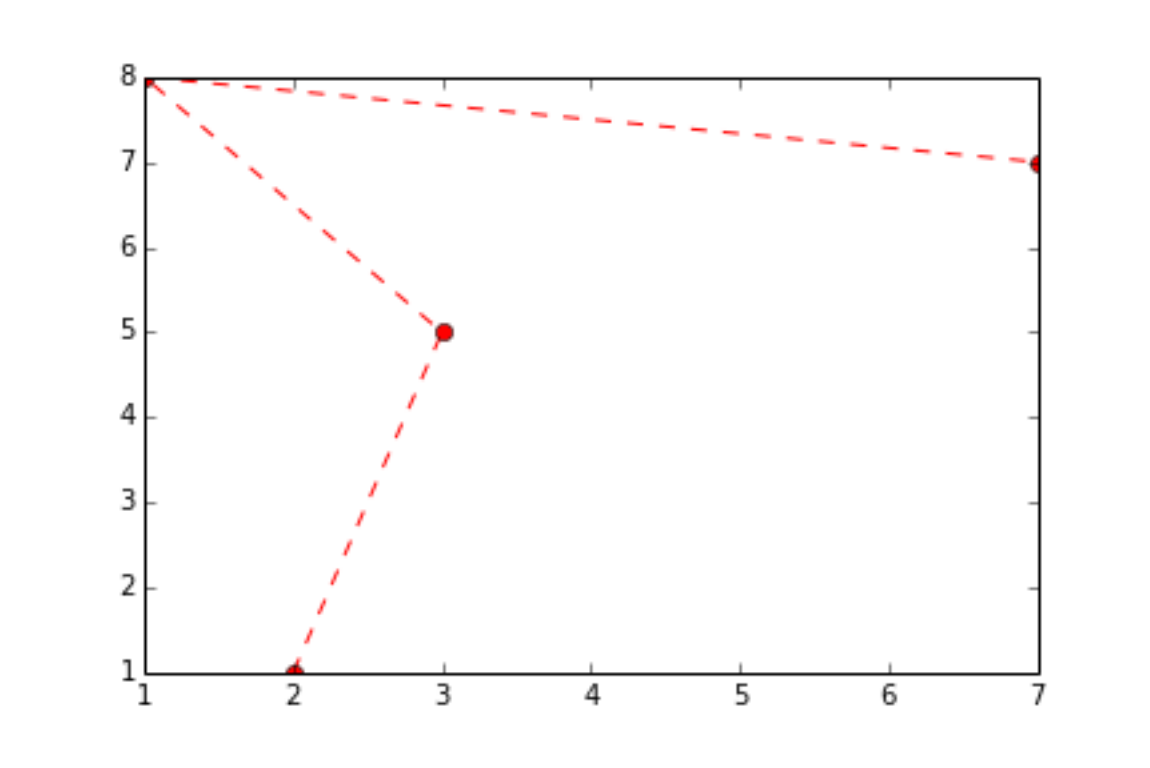
```
plt.plot(x,y, color='red', linestyle='dashed', marker='o')
```



PLT.PLOT()

- Many style features can be expressed using a shorthand

```
plt.plot(x, y, 'r--o')
```



character	description
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style
'.'	point marker
','	pixel marker
'o'	circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker
'>'	triangle_right marker
'1'	tri_down marker
'2'	tri_up marker
'3'	tri_left marker
'4'	tri_right marker

's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
'|'	vline marker
'_'	hline marker

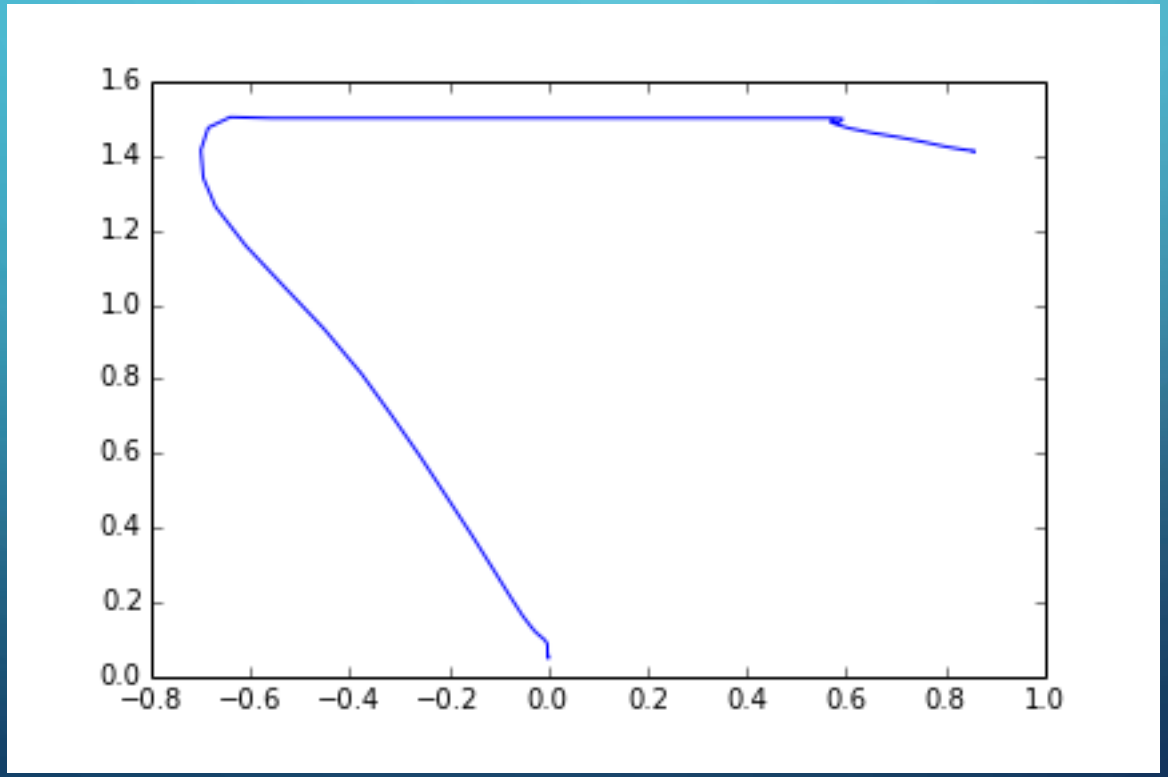
character	color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

QUICK EXERCISE

- Plot the X and Y mouse data from the first row of the dataframe

MY SOLUTION

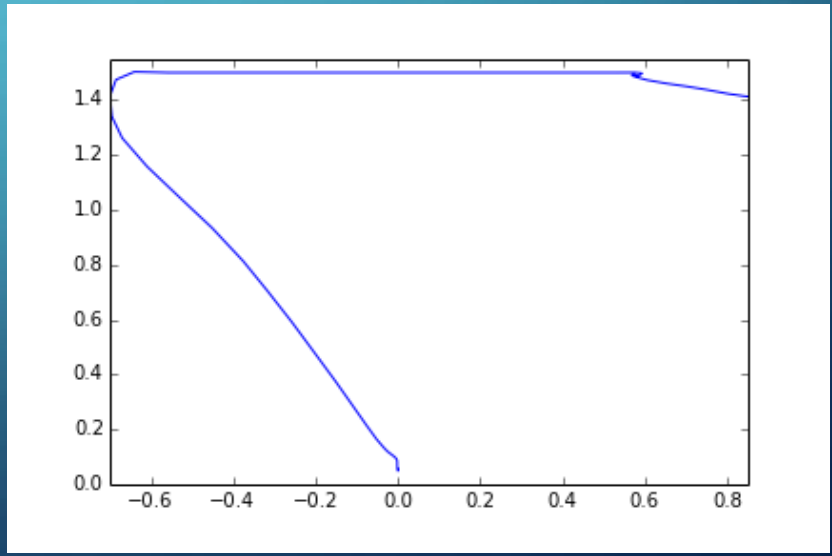
```
plt.plot( df.ix[0]['ts_x'],   df.ix[0]['ts_y'] )
```



XLIM & YLIM

- plt.xlim() and plt.ylim() can be used to both retrieve the current axis limits, as well as set new ones

```
plt.plot(df.ix[0]['ts_x'], df.ix[0]['ts_y'])
plt.xlim( (-0.7, 0.85) )
plt.ylim( (0, 1.55) )
```



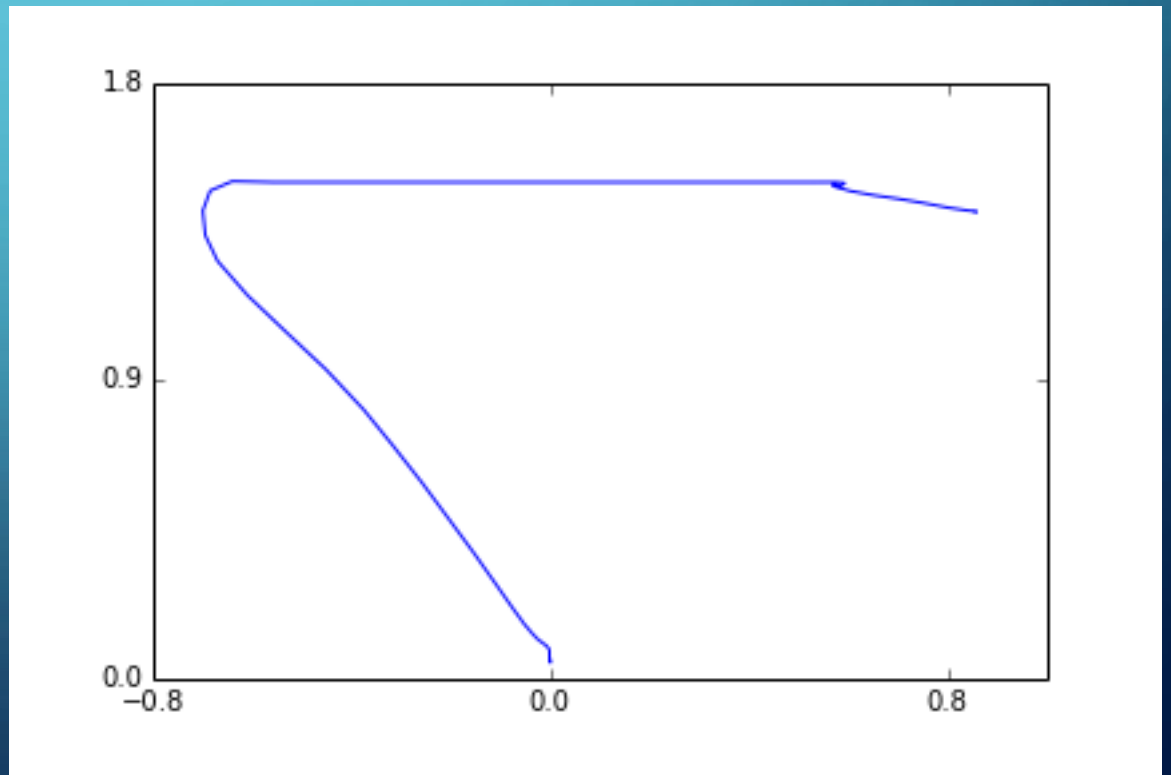
SET TICKS

- plt.xticks() and plt.yticks() will return current tick locations
- They will also allow you to change the tick locations and the tick labels

SET TICKS

- If you pass just one argument you will just change the locations

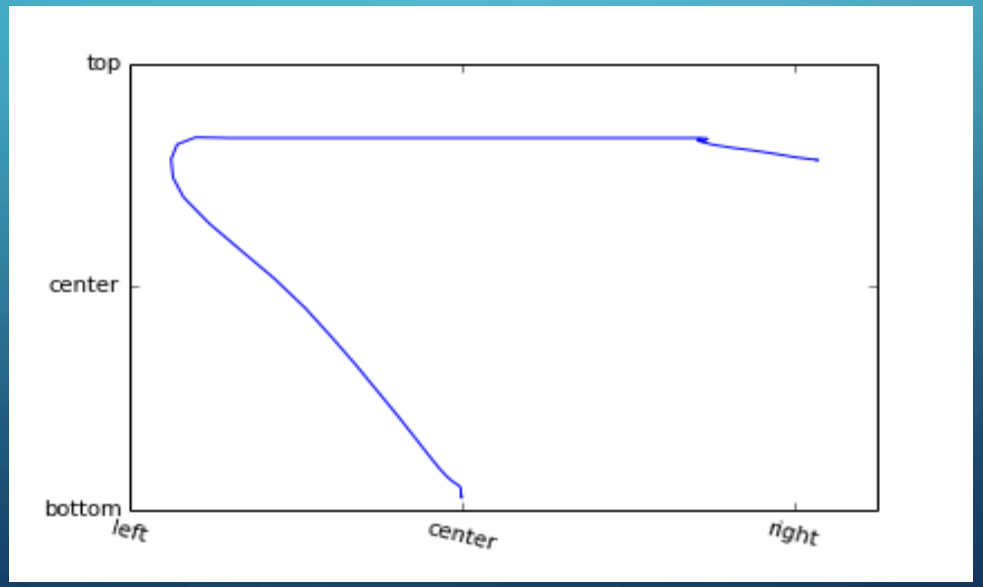
```
plt.plot(df.ix[0]['ts_x'], df.ix[0]['ts_y'])
plt.xticks([-.8, 0, .8])
plt.yticks([0, 0.9, 1.8])
```



SET TICKS

- Passing a second argument of equal length will change the tick labels

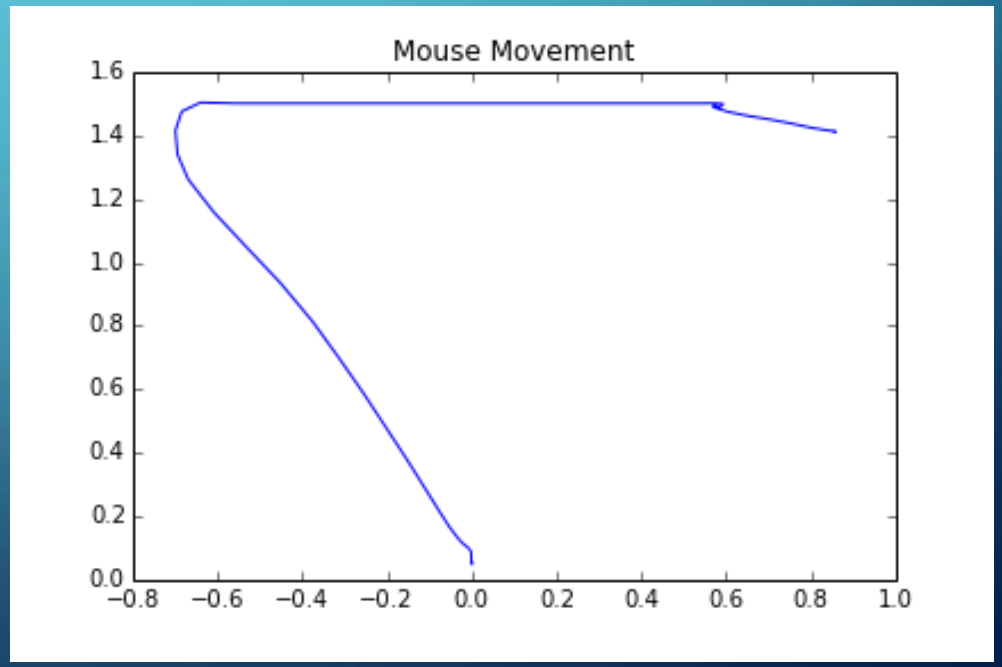
```
plt.plot(df.ix[0]['ts_x'], df.ix[0]['ts_y'])
plt.xticks([-.8, 0, .8], ['left', 'center', 'right'], rotation= -15 )
plt.yticks([0, 0.9, 1.8], ['bottom', 'center', 'top'] )
```



TITLE

- plt.title() will let you set a title for your figure

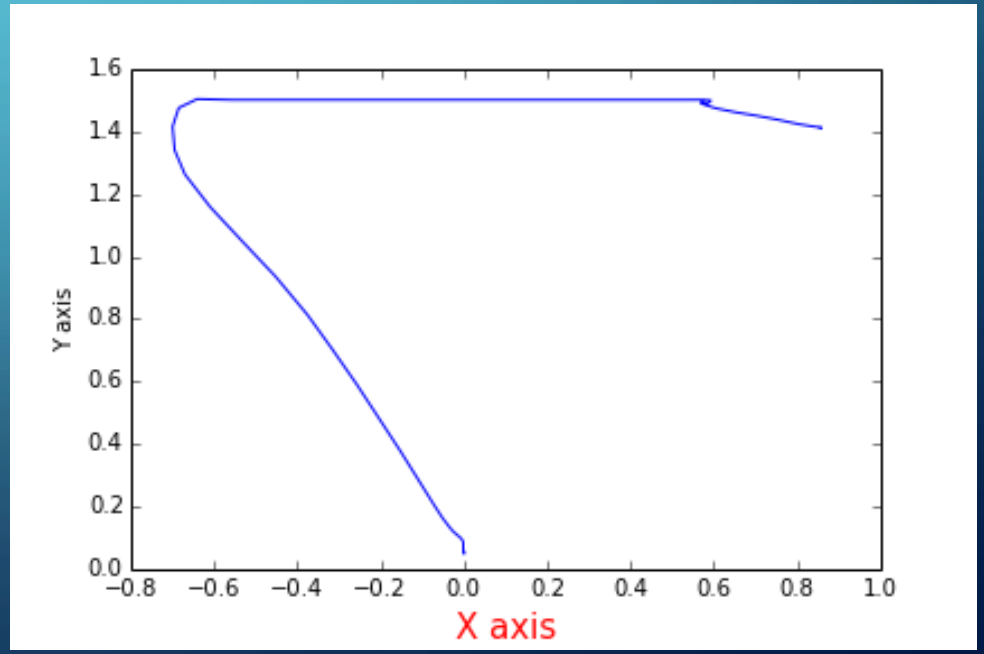
```
plt.plot(df.ix[0]['ts_x'], df.ix[0]['ts_y'])
plt.title('Mouse Movement')
```



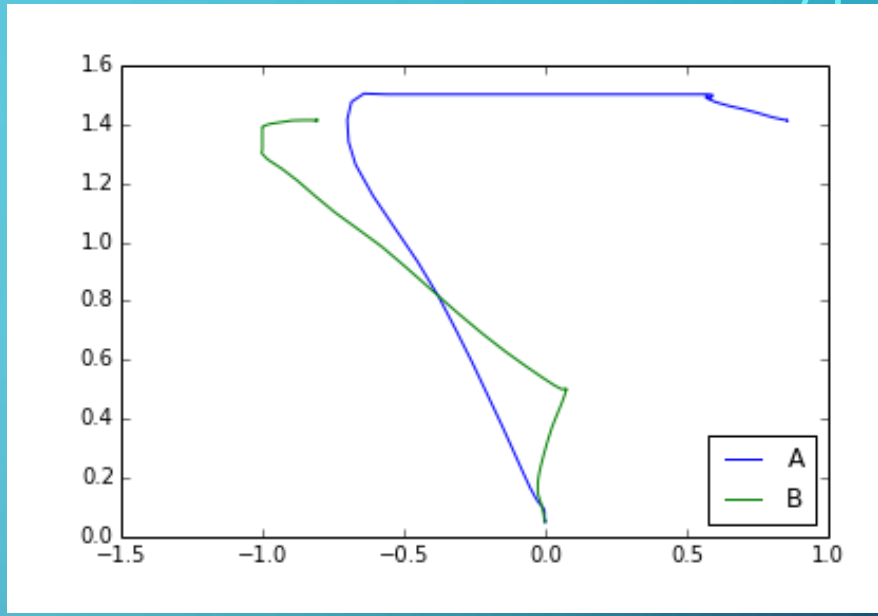
LABELS

- plt.xlabel() and plt.ylabel() set axis labels

```
plt.plot(df.ix[0]['ts_x'], df.ix[0]['ts_y'])
plt.xlabel('X axis', fontsize=15, color='red')
plt.ylabel('Y axis')
```



LEGENDS



- To create a legend label your plots as you draw them
- Then use plt.legend()
  - plt.legend() can take a loc argument to set the legend's location

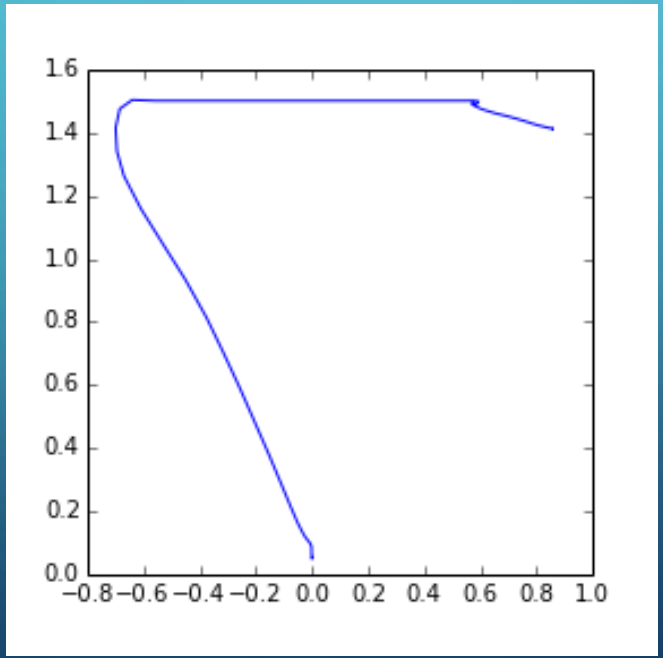
```
plt.plot(df.ix[0]['ts_x'], df.ix[0]['ts_y'], label='A')
plt.plot(df.ix[1]['ts_x'], df.ix[1]['ts_y'], label='B')
plt.legend(loc=4)
```

ADJUSTING YOUR FIGURE SIZE

- plt.figure() can be used to modify the dimensions of your figure
- plt.figure() takes 3 main arguments
  - num: if you set this to a number you can then edit this figure later by calling figure again with the same number. It is often best to pass `None`
  - figsize: takes a size in inches in the form of a tuple
  - dpi: the image resolution
- figsize * dpi = figure dimensions

SETTING FIGURE SIZE

```
plt.figure(num=None, figsize=(4,4), dpi=100)
plt.plot(df.ix[0]['ts_x'], df.ix[0]['ts_y'])
```



SAVING YOUR FIGURE

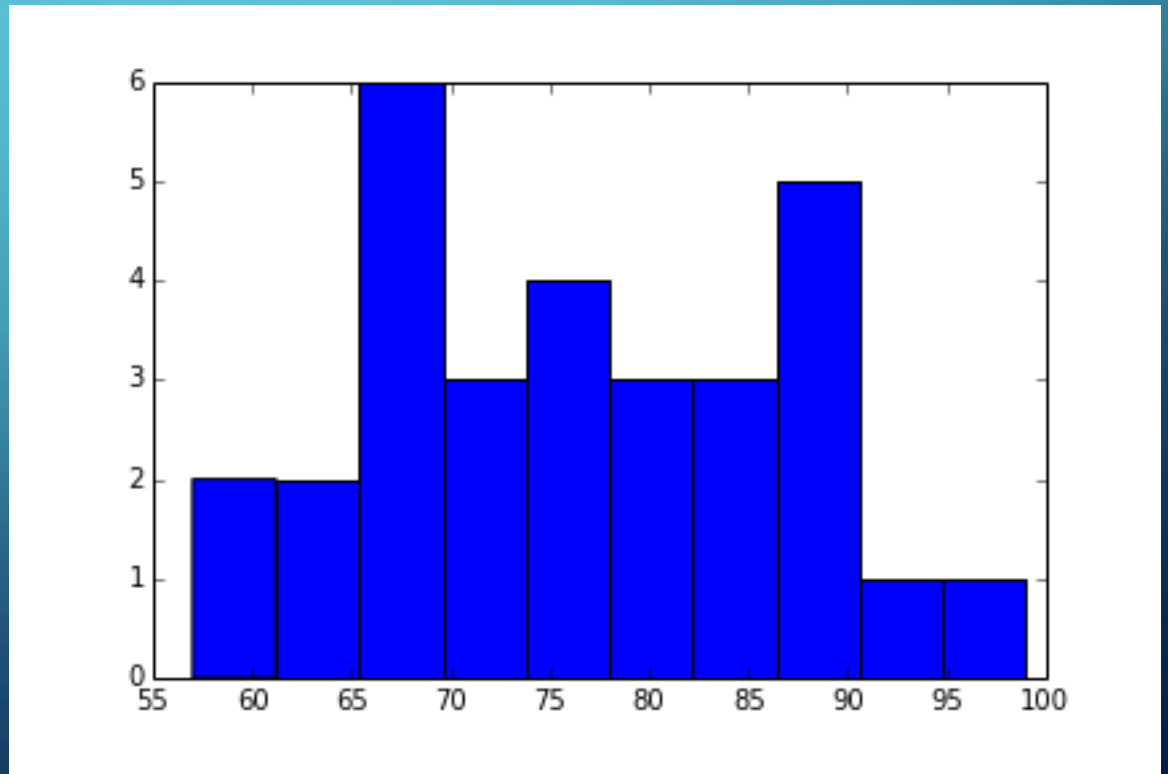
- plt.savefig() will save the current figure. Takes a filename as an argument

```
plt.savefig('awesome_figure.png')
```

HISTOGRAMS

```
nums = [57, 69, 90, 74, 77, 84, 66, 69, 78, 79, 62, 60, 86, 88,
78, 72, 67, 93, 89, 77, 90, 66, 74, 73, 85, 88, 71, 64, 99, 69]
```

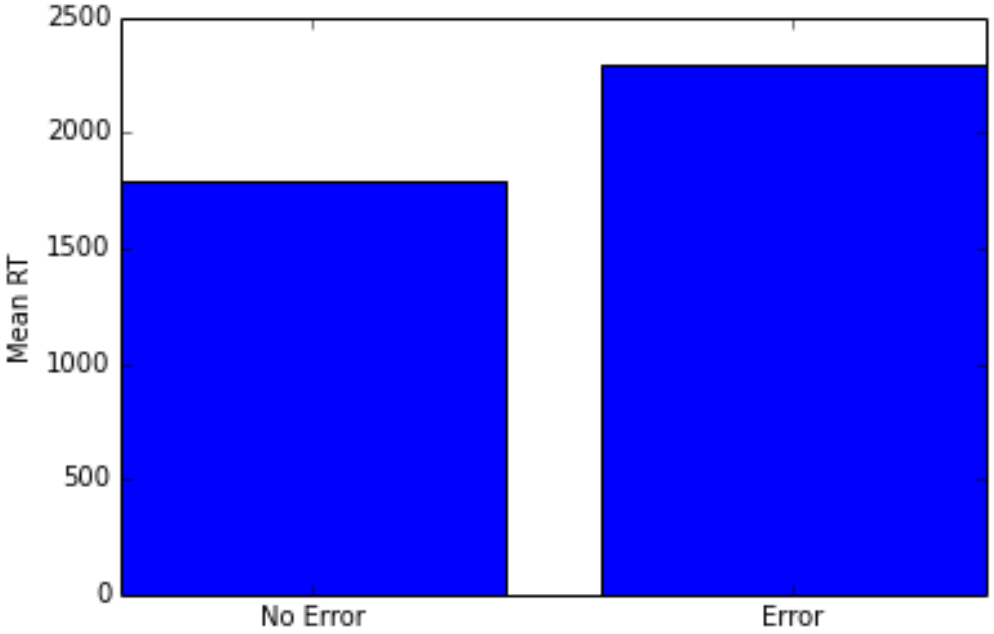
```
plt.hist(nums, bins=10)
```

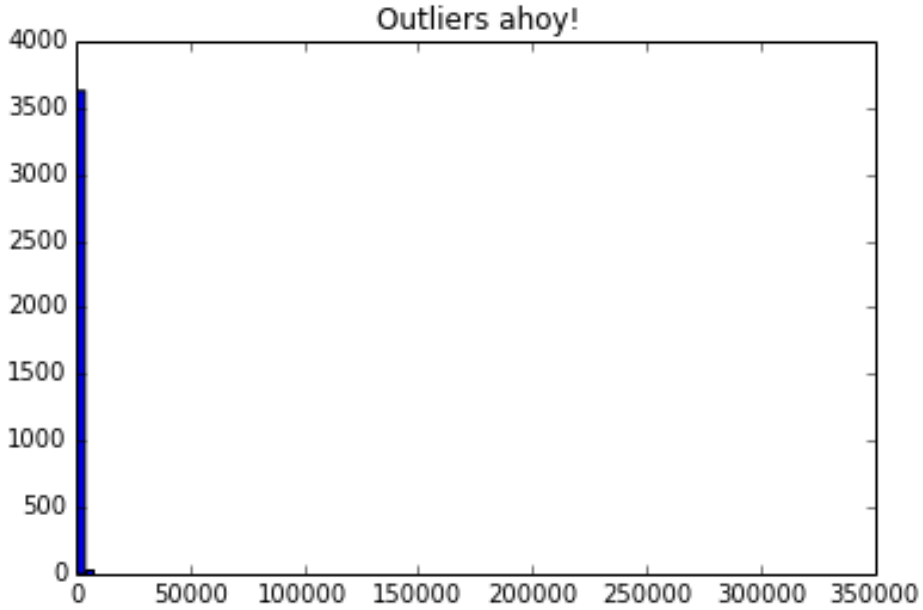


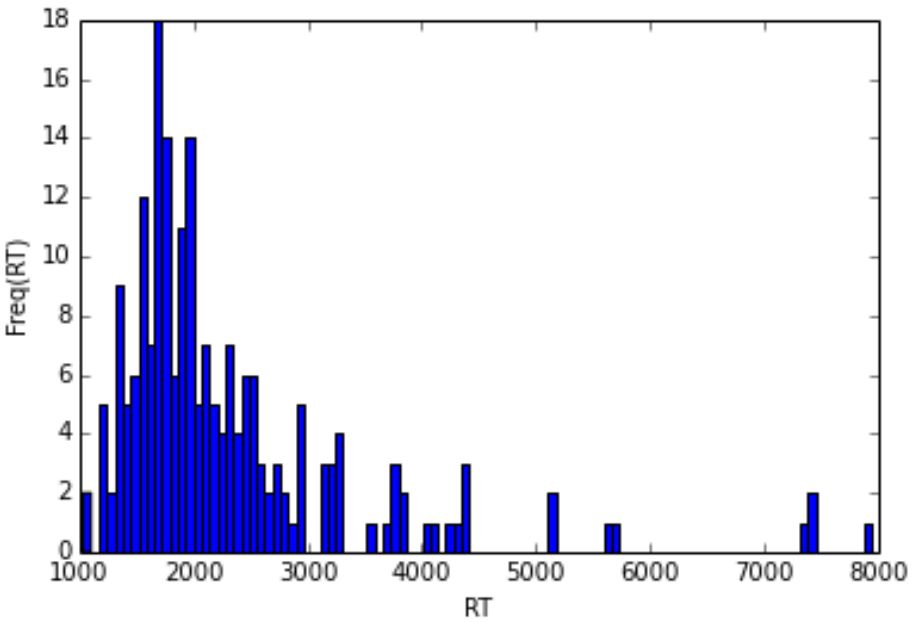
EXERCISE

- Let's take a look at how errors correlate with reaction time
- Plot a bar graph of the mean RTs when there is an error, and when there isn't
  - I haven't covered bar graphs, but you can figure it out
  - http://matplotlib.org/api/pyplot_api.html
- Plot 2 histograms of the reaction times
  - One for each error condition
  - Note: You might need to convert Series objects into Numpy Arrays

```
grouped = df.groupby('error')['RT']

plt.figure()
plt.bar([0, 1], grouped.mean())
plt.xticks([0.4, 1.4], ['No Error', 'Error'])
plt.ylabel('Mean RT')

plt.figure()
plt.hist(np.array(grouped.get_group(0)), bins=100 )
plt.title('Outliers ahoy!')

plt.figure()
plt.hist(np.array(grouped.get_group(1)), bins=100)
plt.xlabel('RT')
plt.ylabel('Freq(RT)')
```


2500
2000
1500
1000
500
0
Mean RT
No Error
Error

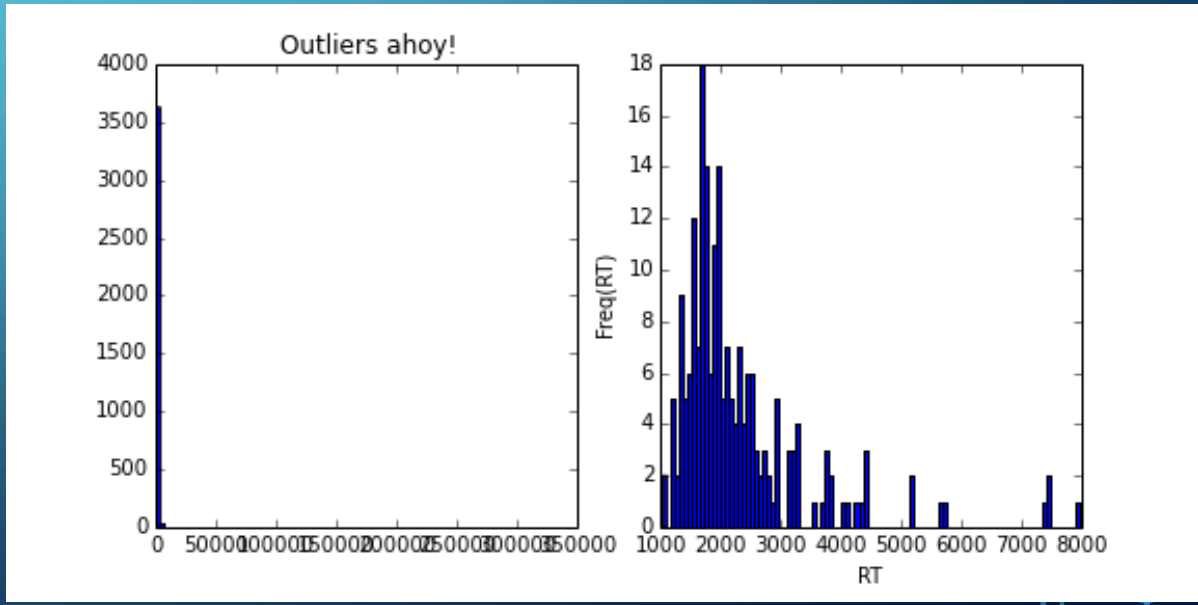

Outliers ahoy!
4000
3500
3000
2500
2000
1500
1000
500
0
0
50000
100000
150000
200000
250000
300000
350000


18
16
14
12
10
8
6
4
2
0
Freq(RT)
1000
2000
3000
4000
5000
6000
7000
8000
RT

SUBPLOTS

- plt.subplot() allows you to have multiple plots on the same figure
  - Takes the following arguments: nrows, ncols, plot_number
- Let's put both our histograms on the same figure

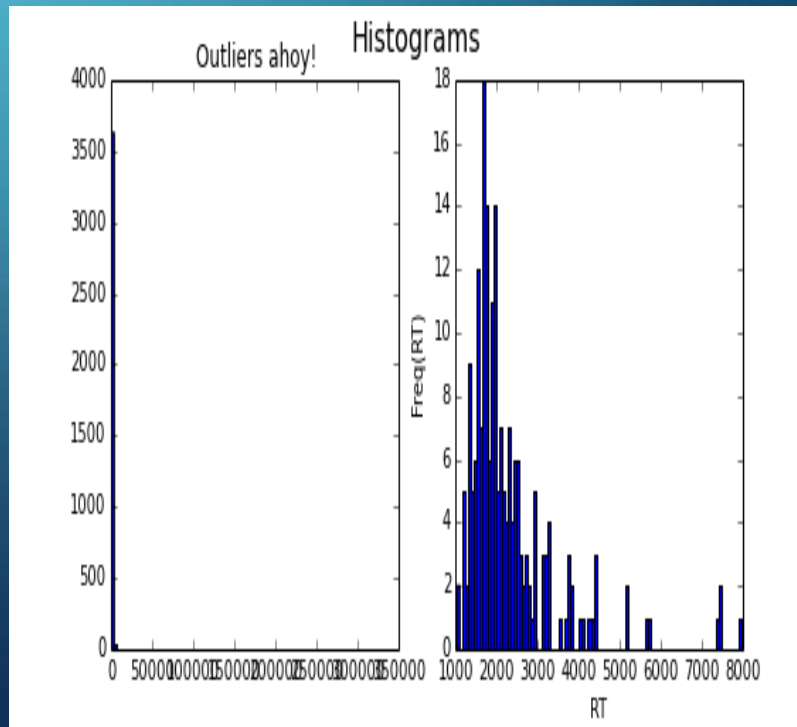
```
plt.figure( None, (8, 4) )
plt.subplot(1, 2, 1) # 1 row, 2 columns
plt.hist(np.array(grouped.get_group(0)), bins=100 )
plt.title('Outliers ahoy!')

plt.subplot(1, 2, 2) # second plot
plt.hist(np.array(grouped.get_group(1)), bins=100)
plt.xlabel('RT')
plt.ylabel('Freq(RT)')
```



SUP TITLE

- plt.suptitle() lets you set a title for the whole figure, even when you have subplots

```
plt.figure( None, (8, 4) )
plt.subplot(1, 2, 1) # 1 row, 2 columns
plt.hist(np.array(grouped.get_group(0)), bins=100 )
plt.title('Outliers ahoy!')

plt.subplot(1, 2, 2) # second plot
plt.hist(np.array(grouped.get_group(1)), bins=100)
plt.xlabel('RT')
plt.ylabel('Freq(RT)')
plt.suptitle('Histograms', fontsize=16)
```



MAKING PLOTS PRETTY

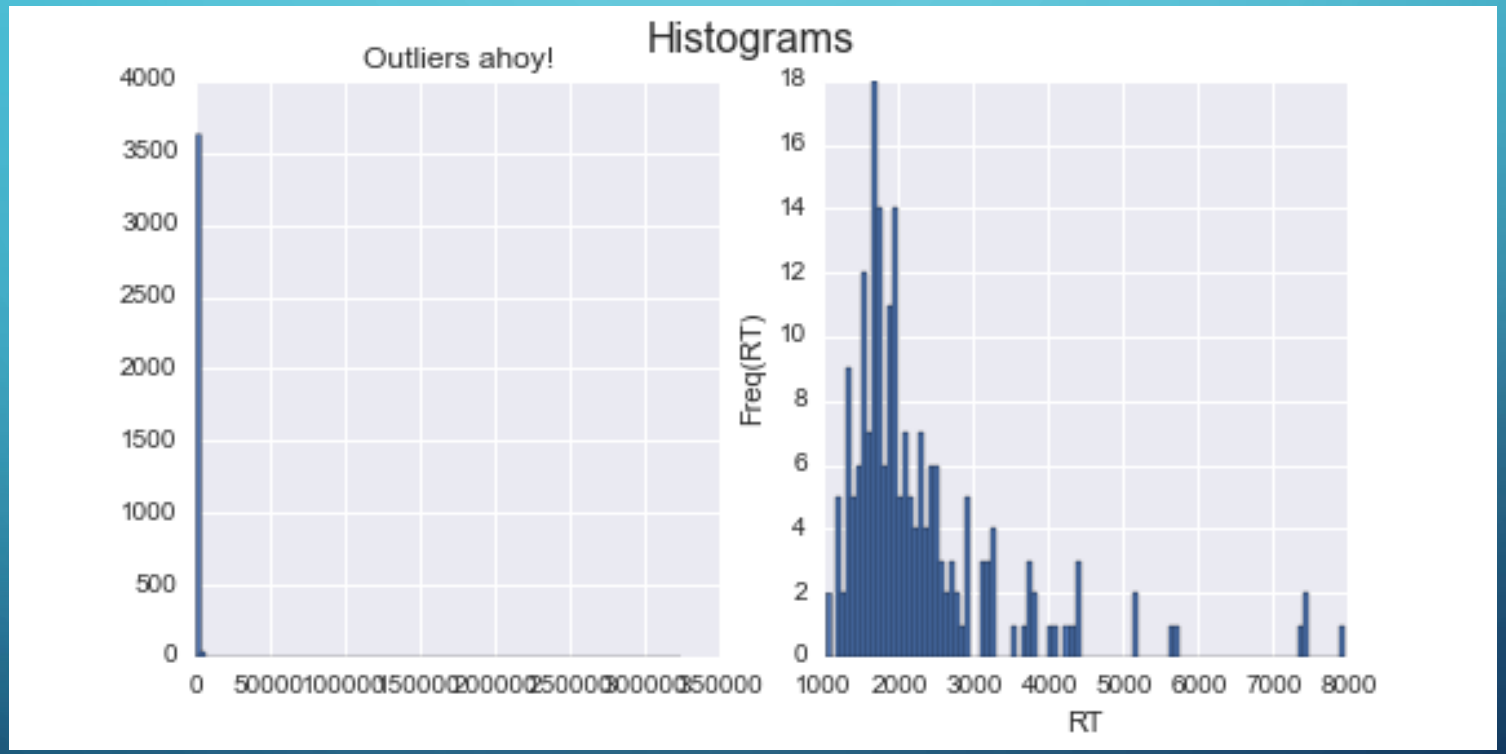
- Seaborn is an excellent library that makes matplotlib plots look nice
- Ages ago I told you a little bit about libraries, but not how to install them
- Generally you will be able to install a library using one of two command line utilities
  - conda
  - pip

INSTALLING LIBRARIES

- Both are accessed from the command line and use similar syntax
  - `conda install seaborn` will work for seaborn
  - packages available for conda can be found at:
    - https://docs.continuum.io/anaconda/pkg-docs
    - or just google "conda packages"
- If a package isn't on conda try pip
  - `pip install <package name>`
  - pip has many more packages, but can have compatibility issues with more complicated packages
  - https://pypi.python.org/pypi/pip

BACK TO MAKING PLOTS PRETTY

- Try importing seaborn, then creating the same plot


Histograms
Outliers ahoy!
4000
3500
3000
2500
2000
1500
1000
500
0
0
50000
100000
150000
200000
250000
300000
350000
18
16
14
12
10
8
6
4
2
0
Freq(RT)
1000
2000
3000
4000
5000
6000
7000
8000
RT

SEABORN

- Seaborn includes a number of different styles
- Change styles with `seaborn.set_style(style_name)`
  - 'darkgrid' - the default seaborn style
  - 'dark' - same, but without the gridlines
  - 'whitegrid' - white background with gridlines
  - 'white' - white without gridlines
  - 'ticks' - white with ticks along the outside

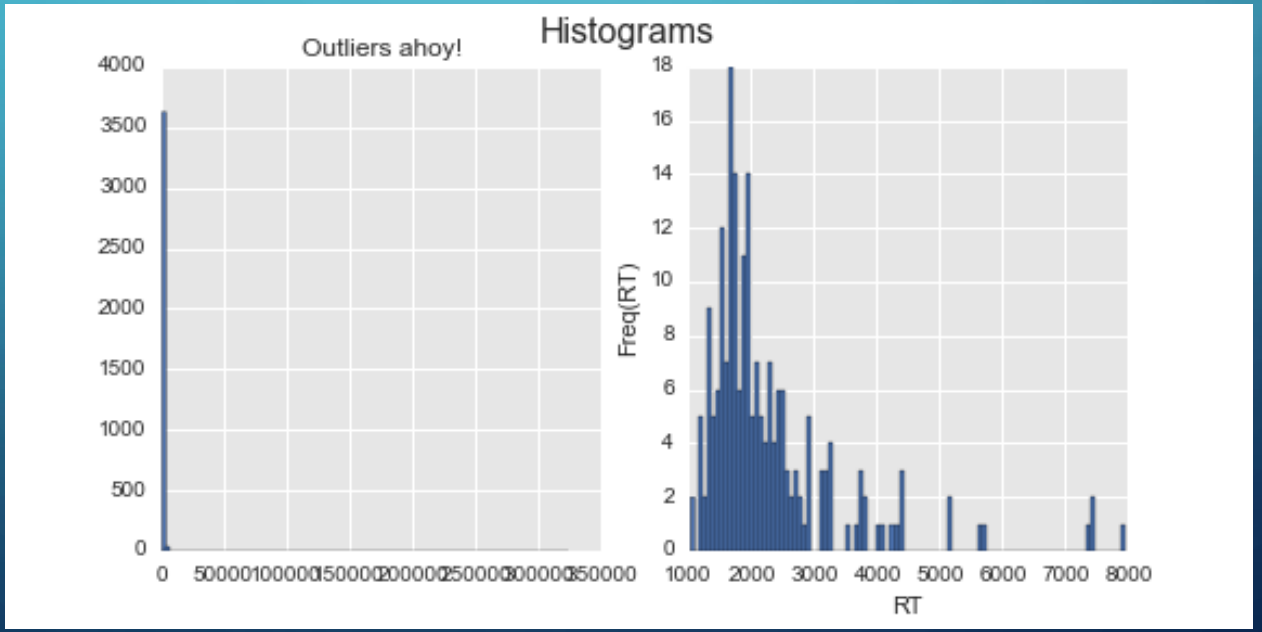
SEABORN

- You can easily view the settings seaborn uses to modify them yourself
- Call `seaborn.axes_style(style_name)` to view a dictionary of settings

```
{'axes.axisbelow': True,
 'axes.edgecolor': '.15',
 'axes.facecolor': 'white',
 'axes.grid': False,
 'axes.labelcolor': '.15',
 'axes.linewidth': 1.25,
 'font.family': 'Arial',
 'grid.color': '.8',
 'grid.linestyle': '-',
 'image.cmap': 'Greys',
 'legend.frameon': False,
 'legend.numpoints': 1,
 'legend.scatterpoints': 1,
 'lines.solid_capstyle': 'round',
 'text.color': '.15',
 'xtick.color': '.15',
 'xtick.direction': 'out',
 'xtick.major.size': 0,
 'xtick.minor.size': 0,
 'ytick.color': '.15',
 'ytick.direction': 'out',
 'ytick.major.size': 0,
 'ytick.minor.size': 0}
```

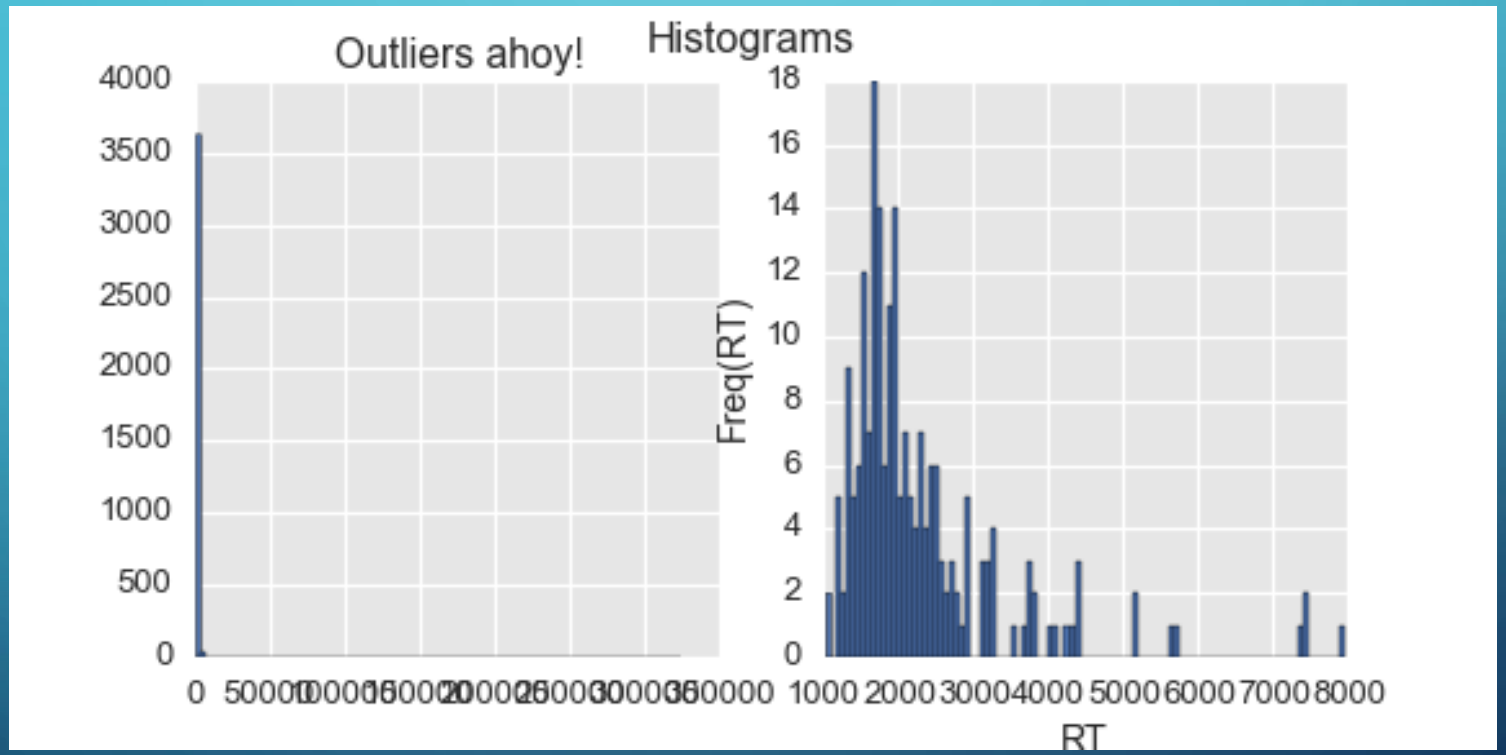
SEABORN

- Calling set_style() with a dictionary containing arguments you want to edit will customize the style
- `seaborn.set_style( 'darkgrid', {'axes.facecolor': '.9'} )`



SEABORN

- Seaborn contexts are also helpful
  - Contexts modify things like font size and line width
  - In order of increasing size: 'paper', 'notebook', 'talk', 'poster'
  - `seaborn.set_context(context_name)`
- I like big plots so I frequently use 'talk'


Outliers ahoy!
4000
3500
3000
2500
2000
1500
1000
500
0
0 50000 100000 150000 200000 250000 300000 350000
Histograms
18
16
14
12
10
8
6
4
2
0
Freq(RT)
1000 2000 3000 4000 5000 6000 7000 8000
RT

FINAL EXERCISE

- Drop the outliers from the data, then generate a figure with 10 plots
  - Each plot should be a histogram of the reaction times for each condition
  - Make the plots as pretty as possible

  - Hint: You should try to use a for loop to generate the subplots

OTHER PYTHON THINGS

- Plotting
  - ggplot: gives ggplot syntax to matplotlib (For the R fans out there)
  - bokeh: Generate interactive plots
  - rpy2: Call R from inside python. Still kinda buggy

OTHER PYTHON THINGS

- Stats and Machine Learning
  - SciPy: contains a lot of useful stuff including integration, optimization, interpolation, signal processing, linear algebra, and some basic statistics
  - scikit-learn: a large number of preprocessing tools and machine learning algorithms including dimensionality reduction, classification, regression, and clustering (no neural nets)
  - TensorFlow: Neural network framework. Made by Google
  - statsmodels: fancier statistics in python

PYTHON FOR SCIENCE

- Biopython: Bioinformatics library
- Astropy: Astronomy research library
- Psychopy: Package for running behavioral experiments

NATURAL LANGUAGE PROCESSING

- NLTK: library for classification, tokenizing, stemming, tagging, and parsing of natural language. Designed with computational linguistics in mind.
- Fuzzywuzzy: Fuzzy string matching using Levenshtein distance. Can be really handy. Made by seatgeek.com
- re: built in regex library. Don't underestimate it.

WEB SCRAPING

- Requests: downloads a web page for parsing
- Beautiful Soup 4: Really great web scraping library. Makes things easy.
- lxml: What BS4 uses under the hood. I know some people prefer to work with it directly
- Scrapy: Let's you build web crawlers

WEB HOSTING

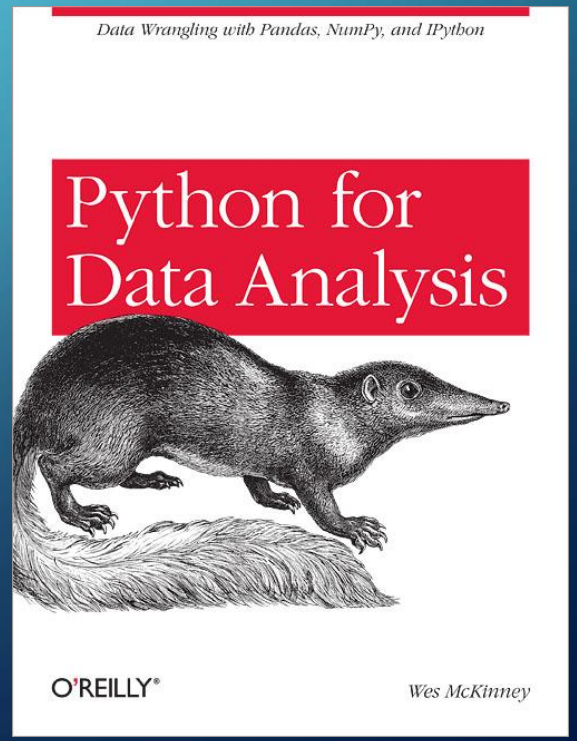
- Django: Full featured suite of tools to take care of every part of running a web page
- Flask: More minimalist set of tools for running a web page. Nice for smaller projects

OTHER STUFF

- SQLalchemy: Really good library for interacting with python
- Numba: Enables JIT compiling of specific parts of your code. Can make your code magically faster. Seriously.
- pillow: image manipulation library
- Twisted: Networking engine. Let's you write custom network applications
- Pygame: 2d game library, can be useful for certain kinds of interactive applications, including behavioral experiments
- pyQT: GUI toolkit for python. Make programs you can CLICK on!
- pdb: A built in python debugger.

USEFUL RESOURCE

- Python for Data Analysis by Wes McKinney
  - Great resource. Covers much of the things I covered here in more depth
  - We have free access to it
  - http://proquest.safaribooksonline.com/



END