LAB #0: INTRODUCTION TO PYTHON FOR DATA SCIENCE

CS 109A, STAT 121A, AC 209A: Data Science

Fall 2016

Harvard University

What is Jupyter/IPtyhon

What Python Looks Like

Variables and Types

Manipulating Python Data Types

Basic Control Structures

Task I: Tabular Data from File
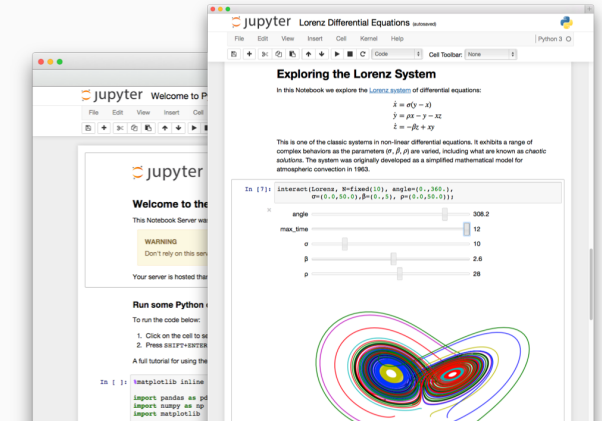
Task II: Visualize Data
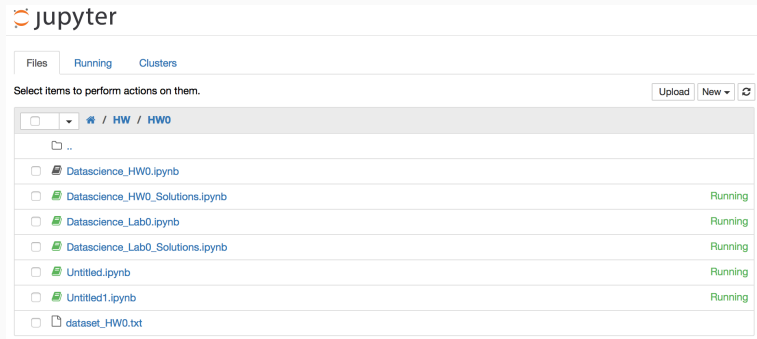
Task III: Web Data

Task IV: Data from Simulations

WHAT IS JUPYTER/IPTYHON

"The Jupyter Notebook is a web application that allows you to create interactive documents that contain live code, equations, visualizations and explanatory text."

When Jupyter app loads, you see a dashboard displaying files in the Jupyter home directory (you can reset this)
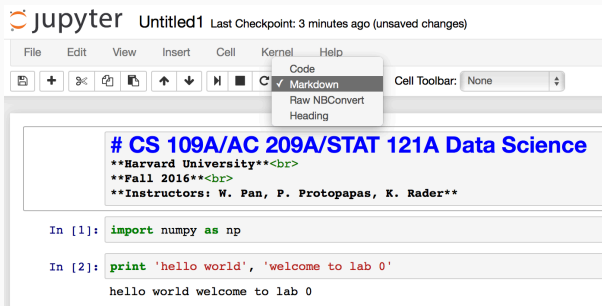
Each notebook consists of blocks of cells. Each cell can display rich text elements (Markdown) or code. Code is executed by an "computational engine" called the *kernel* (IPython). The output of the code is displayed directly below.

Each cell can be executed independently, but once a block of code is executed, it lives in the memory of the kernel.

```
In [1]: x = 2
```

Some expository text

```
In [2]: print x + 1
        3
```

WHAT PYTHON LOOKS LIKE

Code readability is key, Python syntax itself is close to plain english.

· <mark>Your variables should be given **descriptive identifiers**</mark>!

Identifiers for variable should be descriptive words separated by underscore (not spaces) and in all lower case

```
BAD                     GOOD
var6 = 25               age_of_mother = 25
AG3ofMoTh3R = 25        age_of_mother = 25
```

Code readability is key, Python syntax itself is close to plain english.

· Your variables should be given **descriptive identifiers**!

· You should use **white space** to increase readability.

```
BAD                      GOOD
x=[2,3,4]                num_list = [2, 3, 4]
v=1/3*(pi*r**2*h)        v = 1/3 * (pi * r**2 * h)
```

Code readability is key, Python syntax itself is close to plain english.

· Your variables should be given **descriptive identifiers**!
· You should use **white space** to increase readability.
· You should liberally intersperse your code with **comments**!

BAD   GOOD

                                       ```
                                       #volume of cone
v = 1/3 * (pi * r**2 * h)    v = 1/3 * (pi * r**2 * h)
                                       ```

A line of text following by **#** is treated as a comment.

Code readability is key, Python syntax itself is close to plain english.

· Your variables should be given **descriptive identifiers**!
· You should use **white space** to increase readability.
· You should liberally intersperse your code with **comments**!
· **Proper indentation is non-negotiable**!

<u>BAD</u>
```
for i in range(5):
print i
```

<u>GOOD</u>
```
for i in range(5):
    print i
```

Code blocks are not indicated by delimiters (e.g. { }) only by indentation!

VARIABLES AND TYPES

The basic built-in Python data types we'll be using today are:

1. **integers, floating points**: 7, 7.0
2. **booleans**: True, False with logical operations, and, or, not
3. **strings**: 'hi', "7.0"
4. **lists**: sequence of data (of various types)

In Python, you do not need to *declare* the types of your variables. The type is inferred based on the valued assigned to the variable.

For example: The assignment

$$my\_var = 7$$

types **my_var** as an integer. Later, the assignment

$$my\_var =' hello'$$

will cause **my_var** to be typed as a string.

Function definition follows the def keyword. The first line, the heading, contains the function name and the list of parameters names (you need not specify the type of each parameter). Finally, if your function returns a value, you can do so with the return keyword.

```
def add(x, y):
   return x + y
```
You call a function by its name with values for each parameter:

```
answer = add(1, 2)
```

Calling a function belonging to an object or library:

```
returned_val = object.method(param_1, param_2, ...)
returned_val = library.function(param_1, param_2, ...)
```

MANIPULATING PYTHON DATA TYPES

Python has a variety of built-in **arithmetic operators** that allows you to combine numbers.

Operator	Description	Example
+	adds values on either side	1.2 + 2 = 3.2
-	subtracts the right value from the left	1.2 - 0.2 = 1.0
*	multiplies values on either side	1.2 * 2 = 2.4
/	divides the left value by the right	4 / 2 = 2.0
%	divides the left value by the right and returns the remainder	4 % 3 = 1
**	exponentiate the left value by the right	3**2 = 9
//	divides the left value by the right and removes the decimal part	3//2 = 1

Python also has a variety of built-in **comparison operators** for numbers.

Operator	Description	Example
==	checks if values on either side are equal	`1 == 2` is `False`
!=	checks if values on either side are unequal	`1 != 2` is `True`
>	checks if left value is greater	`1 > 2` is `False`
<	checks if left value is smaller	`1 < 2` is `True`
>=	checks if left value is greater or equal	`2 >= 2` is `True`
<=	checks if left value is smaller or equal	`1 <= 2` is `True`

String literals in Python are a set of characters enclosed by either single or double quotation marks.

For example: The following are two equivalent assignments

```
my_str = "Hello World!"
my_str = 'Hello World!'
```

Python has a variety of built-in operator for string manipulation.

Let's say `s = 'Hi!'`.

Operator	Description	Example
==	checks if strings on either side are equal	`s == 'hi!'` is `False`
!=	checks if strings on either side are unequal	`s != 'hi!'` is `True`
+	appends right string to end of left	`'Hi' + '!'` is `'Hi!'`
[n]	returns the *n*-th character	`s[0]` is `'H'`
[n:m]	returns the substring from *n* up to *m*	`s[0:1]` is `'H'`
[n:]	returns the substring from *n* on	`s[1:]` is `'i!'`
[:n]	returns the substring up to *n*	`s[:2]` is `'Hi'`

Note: Python enumerates starting from zero!

Lists in Python are collections of items of possibly **different types**. Lists are created and displayed with items separated by commas and enclosed by square brackets. The empty list is denoted by [ ].

For example: The following list contains both numerical and string data types.

```
>>> ['hi', 70, 2.1, ':(', '<3']
```

Python has a variety of built-in operator for list manipulation (they look just like the string operators).

Let's set lst = ['hi', 7, 'c'].

Operator	Description	Example
+	appends right list to end of left	['H'] + [2] is ['H', 2]
[n]	returns the *n*-th item	lst[0] is 'hi'
[n:m]	returns items from *n* up to *m*	lst[0:1] is ['hi']
[n:]	returns items from *n* on	lst[1:] is [7, 'c']
[:n]	returns items up to *n*	lst[:2] is ['hi', 7]

BASIC CONTROL STRUCTURES

SELECTION: 'IF', 'ELIF', 'ELSE'

In Python, selection is implemented using the `if, elif, else` constructions.

For example: Our holistic 0-5 homework grading scheme might translate into:

```
if grade == 5:
    print "Everything was outstanding!"
elif grade == 4:
    print "Everything was good with no major mistakes"
elif grade == 3 or grade == 2:
    print "Good with some major mistakes"
elif grade == 1:
    print "Hmm...there seem to be lots of missing solutions"
elif grade == 0:
    print "Oops! You forgot to submit this one!"
else:
    print "That's not a valid grade!"
```

We can directly **iterate over the items in a list** (from 0-th index to end).

```
In [9]: my_list = [1, 2, 3, 4]

        for val in my_list:
            print val

        1
        2
        3
        4
```

We can **iterate over a range of numbers**.

```
In [10]: my_list = [1, 2, 3, 4]

         for index in range(4):
             print my_list[index]

         1
         2
         3
         4
```

Variations on `range()`:

· `range(10)` produces a list-like of numbers 0 thru 9:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$$

· `range(5, 10)` will produce a list-like of numbers starting at 5 and thru 9

$$5, 6, 7, 8, 9$$

· `range(0, 10, 2)` will produce a list-like of numbers between 0 and 9 counting by 2:

$$0, 2, 4, 6, 8$$

TASK I: TABULAR DATA FROM FILE

A comma-separated values (CSV) file stores tabular data in plain text. Each line of the file is a single record. Each record consists of one or more values, numeric or text, separated, typically, by commas.

```
1,0.455,0.365,0.095,0.514,0.2245,0.101,0.15,13
1,0.35,0.265,0.09,0.2255,0.0995,0.0485,0.07,5
2,0.53,0.42,0.135,0.677,0.2565,0.1415,0.21,7
1,0.44,0.365,0.125,0.516,0.2155,0.114,0.155,8
0,0.33,0.255,0.08,0.205,0.0895,0.0395,0.055,5
0,0.425,0.3,0.095,0.3515,0.141,0.0775,0.12,6
2,0.53,0.415,0.15,0.7775,0.237,0.1415,0.33,18
2,0.545,0.425,0.125,0.768,0.294,0.1495,0.26,14
1,0.475,0.37,0.125,0.5095,0.2165,0.1125,0.165,7
2,0.55,0.44,0.15,0.8945,0.3145,0.151,0.32,17
2,0.525,0.38,0.14,0.6065,0.194,0.1475,0.21,12
1,0.43,0.35,0.11,0.406,0.1675,0.081,0.135,8
1,0.49,0.38,0.135,0.5415,0.2175,0.095,0.19,9
2,0.535,0.405,0.145,0.6845,0.2725,0.171,0.205,8
2,0.47,0.355,0.1,0.4755,0.1675,0.0805,0.185,8
1,0.5,0.4,0.13,0.6645,0.258,0.133,0.24,10
```
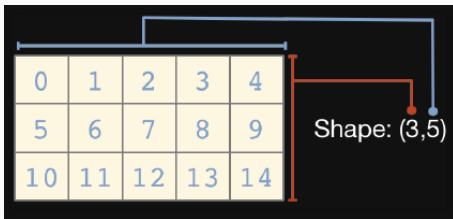
numpy is a useful package for scientific computation. numpy is typically imported as np.

Notably, numpy provides an multi-dimensional array object which optimizes storing and manipulating data.

numpy also provides a number of way to load csv data into arrays (through loadtxt or genfromtxt).

Each array has a shape, recorded as a tuple $(n, m, \ldots)$.

Creating 1D arrays:

```
In [19]: my_array = np.array([1, 2, 3, 4])
         print my_array.shape

         (4,)
```

Note: indexing with 1D arrays work just like with lists and strings!

Computations with 1D arrays:

```
In [21]: print my_array.mean()
         print np.mean(my_array)

         2.5
         2.5
```

You can compute the mean by either call the mean function belonging to the array object or by applying **numpy**'s mean function to the array.

Creating 2D arrays: we can make 2-D arrays out of a list of rows, each row is a list of values.

```
In [24]: my_array = np.array([[1, 2], [3, 4]])
         print my_array

         [[1 2]
          [3 4]]

In [25]: print my_array.shape

         (2, 2)
```
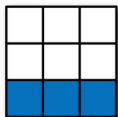
Indexing 2D arrays: The element at the *n*-th row and the *m*-th column is indexed as [n, m]. Just like lists, you can also get multiple array values at a time:
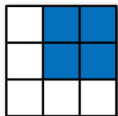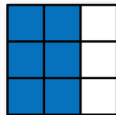

arr[:2, 1:]

arr[:, :2]

arr[2]
arr[2, :]

Finally, you can get a "list" rows: arr[[0, 1]] or arr[[0, 1], :]

Filtering 2D arrays: Even more sophisticated, you can get values from an array that satisfy a bunch of criteria!

```
In [24]: my_array = np.array([[1, 2], [3, 4]])
         print my_array

         [[1 2]
          [3 4]]

In [44]: my_array = np.array([[1, 2], [3, 4]])
         print my_array[:, 0] == 1

         [ True False]

In [34]: print my_array[my_array[:, 0] == 1]

         [[1 2]]

In [37]: print my_array[(my_array[:, 0] == 1) | (my_array[:, 0] == 3)]

         [[1 2]
          [3 4]]
```

Question: how do you get the values that are greater than one? What is the shape of this array of filtered values?

Let's process some tabular data!

Download files for lab 0 at: https://github.com/cs109Alabs/lab_files

Do problem 1, parts a and b

TASK II: VISUALIZE DATA

matplotlib

matplotlib is a plotting library, the pyplot module contains a set of functions especially useful for generating a wide range of simple plots. pyplot is typically imported as plt.

Plotting function	Input	Result
plt.plot(x, y)	*x*-coords and *y*-coords	curve defined by the set of x, y coords
plt.scatter(x, y)	*x*-coords and *y*-coords	scatter plot defined by the set of x, y coords
plt.hist(vals)	an array or list of values	histogram of the list of values
plt.title(plot_title)	a string	adds title
plt.show()	none	displays all figures

To generate a group of 3 plots in a 3×1 grid, say. We want to explicitly create a figure and add subplots to particular positions of the grid.

Function	Input	Result
`plt.figure()`	(optional) figure size	returns a new figure
`figure.add_subplot(n, m, k)`	row, column, subplot number	returns an axes for the *k*-th subplot in the *nxm*-grid column
`figure.add_subplot(n, m, k, projection='3d')`	row, column, subplot number, projection type	returns an axes for the *k*-th subplot in the *nxm*-grid column

You can do all your favorite plotting on the axes of each subplot.

To generate a group of 3 plots in a 3×1 grid, say. We want to explicitly create a figure and add subplots to particular positions of the grid.
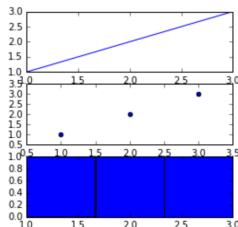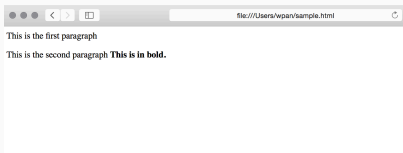
Let's do some simple data-visualization!

Do problem 1, parts c and d

TASK III: WEB DATA

An importance source of data is web data,
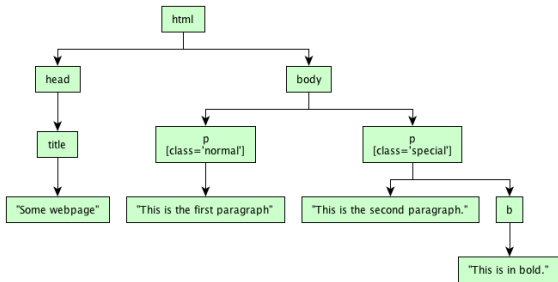where simple displays you see on screen are specified complex syntax.

BeautifulSoup is a Python package that will represent the
information in an html file in a simple to access data structure,
along with many useful functions to manipulate this structure.

```
page = urllib.urlopen("some_page.html").read()
soup = BeautifulSoup(page, "lxml")
```


The function soup.prettify() will turn the tree into a nicely formatted string (like the HTML file we wrote).

The function soup.get_text() will turn all the displayed text on the page as a string.

beautifulsoup

Code	Result	Example
`soup.tag`	returns the first instance of tag	`soup.b`
`parent.child_tag`	access the tag named 'child_tag' from it's parent tag	`soup.html.head`
`tag.contents`	returns all content (text and descendent tags)	`soup.html.head.contents`
`tag.string`	returns any strings in the tag, not belonging to child tags	`soup.html.head.string`
`tag.children`	returns a "list" of all child tags	`soup.html.head.children`
`soup.find_all(tag_name)`	returns a "list" of all tags named "tag_name"	`soup.find_all('b')`

Let's do some simple web-scrapping!

Do problem 2, parts a, b and c

In Python, strings are **immutable**!

```
In [1]: my_list = [0, 1, 'hi', 3, 4, 5]
        my_list[2] = 1
        print my_list

        [0, 1, 1, 3, 4, 5]
```

```
In [2]: my_string = 'hello'
        my_string[0] = 'H'

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        <ipython-input-2-e25a76cbc7ac> in <module>()
              1 my_string = 'hello'
        ----> 2 my_string[0] = 'H'

        TypeError: 'str' object does not support item assignment
```

```
In [3]: my_string = 'H' + my_string[1:]
        print my_string

        Hello
```

The Python string object has many useful functions.

`string = 'Hi world', substr = 'world', newstr='World', sep=' '.`

Function	Returned Value	Example
`len(string)`	length of string*	11
`string.replace(substr, newstr)`	string with substr replaced with newstr	`'Hello World'`
`string.split(sep)`	a list of substrings of string separated by sep	`['Hi', 'world']`
`string.find(substr)`	the first position where substr occurs in string	3
`' '.join(string, newstr)`	two strings concatenated separated by a space	`'Hello World World'`

* Note: `len()` also takes lists as input.

The Python list object also has many useful functions. For example, the `.append()` function allows us to build lists from scratch.

```
In [11]:  my_list = []

          for num in range(10):
              my_list.append(num)

          print my_list

          [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Let's do some text analysis on The Metamorphosis!

Do problem 2, parts d and e.

TASK IV: DATA FROM SIMULATIONS

numpy has a library/module called random that is great for sampling (generating) random numbers. Read the documentation for random!

Let's simulate a simple queue!

Do problem 3, part a.