Module 1: Introduction to everything

Lab 1: Getting Started with Python

About Python

Python is a

- general purpose programming language
- interpreted, not compiled
- both **dynamically typed** *and* **strongly typed**
- supports multiple programming paradigms: object oriented, functional
- comes in 2 main versions in use today: 2.7 and 3.x

Why Python for Data Science?

Python is great for data science because:

- general purpose programming language (as opposed to R)
- faster idea to execution to deployment
- battle-tested
- mature ML libraries

Python's Interactive Console : The Interpreter

- The Python interpreter is a console that allows interactive development
- We are currently using the Jupyter notebook, which uses an advanced Python interpreter called IPython
- This gives us much more power and flexibility

Let's try it out !

```
print("Hello World!") #As usual with any language we start with with the print function
```

```
Hello World!
```

Objevtive

- **Python Basics**
  - **Strings** - Creating a String, variable assignments - String Indexing & Slicing - String Concatenation & Repetition - Basic Built-in String Methods
  - **Numbers** - Types of Numbers - Basic Arithmetic
- **Data Types & Data Structures**
  - Lists
  - Dictionaries
  - Sets & Booleans
- **Python Programming Constructs**
  - Loops & Iterative Statements - if,elif,else statements - for loops, while loops

Python Basics

Let's understand

- Basic data types
- Variables and Scoping
- Modules, Packages and the **`import`** statement
- Operators

Strings

Strings are used in Python to record text information, such as name. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

This idea of a sequence is an important one in Python and we will touch upon it later on in the future.

In this lecture we'll learn about the following:

```
1.) Creating Strings
2.) Printing Strings
3.) String Indexing and Slicing
4.) String Properties
5.) String Methods
6.) Print Formatting
```

Creating a String

To create a string in Python you need to use either single quotes or double quotes. For example:

```
# Single word
print('hello')

print() # Used to have a line space between two sentences. Try
deleting this line & seeing the difference.

# Entire phrase
print('This is also a string')
```

```
hello

This is also a string
```

Variables : Store your Value in me!

In the code below we begin to explore how we can use a variable to which a string can be assigned.

This can be extremely useful in many cases, where you can call the variable instead of typing the string everytime.

This not only makes our code clean but it also makes it less redundant. Example syntax to assign a value or expression to a variable,

```
variable_name = value or expression
```

Now let's get coding!!. With the below block of code showing how to assign a string to variable.

```
s = 'New York'

print(s)

print(type(s))

print(len(s))  # what's the string length
```

```
New York
<class 'str'>
8
```

String Indexing

We know strings are a sequence, which means Python can use indexes to call parts of the sequence. Let's learn how this works.

In Python, we use brackets [] after an object to call its index. We should also note that indexing starts at 0 for Python. Let's create a new object called s and the walk through a few examples of indexing.

```
# Assign s as a string
s = 'Hello World'

# Print the object
print(s)

print()

# Show first element (in this case a letter)
print(s[0])

print()

# Show the second element (also a letter)
print(s[1])
```

```
Hello World

H

e
```

String Concatenation and Repetition

String Concatenation is a process to combine two strings. It is done using the '+' operator.

String Repetition is a process of repeating a same string multiple times

The examples of the above concepts is as follows.

```
# concatenation (addition)

s1 = 'Hello'
s2 = "World"
print(s1 + " " + s2)
```

```
Hello World
```

```
# repetition (multiplication)

print("Hello_" * 3)
print("-" * 10)
```

```
Hello_Hello_Hello_
----------
```

String Slicing & Indexing

String Indexing is used to to select the letter at a particular index/position.

String Slicing is a process to select a subset of an entire string

The examples of the above stated are as follows

```
s = "Namaste World"

# print sub strings
print(s[1])     #This is indexing.
print(s[6:11])  #This is known as slicing.
print(s[-5:-1])

# test substring membership
print("Wor" in s)
```

```
a
e Wor
Worl
True
```

Note the above slicing. Here we're telling Python to grab everything from 6 up to 10 and from fifth last to second last. You'll notice this a lot in Python, where statements and are usually in the context of "up to, but not including".

Basic Built-in String methods

Objects in Python usually have built-in methods. These methods are functions inside the object (we will learn about these in much more depth later) that can perform actions or commands on the object itself.

We call methods with a period and then the method name. Methods are in the form:

object.method(parameters)

Where parameters are extra arguments we can pass into the method. Don't worry if the details don't make 100% sense right now.

Later on we will be creating our own objects and functions!

Here are some examples of built-in methods in strings:

```
s = "Hello World"

print(s.upper()) ## Convert all the element of the string to Upper
case..!!
print(s.lower()) ## Convert all the element of the string to Lower
case..!!
```

```
HELLO WORLD
hello world
```

Print Formatting

We can use the .format() method to add formatted objects to printed string statements.

The easiest way to show this is through an example:

```
name = "Einstein"
age = 22
married = True

print("My name is %s, my age is %s, and it is %s that I am married" %
(name, age, married))

print("My name is {}, my age is {}, and it is {} that I am
married".format(name, age, married))
```

```
My name is Einstein, my age is 22, and it is True that I am married
My name is Einstein, my age is 22, and it is True that I am married
```

Numbers

Having worked with string we will turn our attention to numbers

We'll learn about the following topics:

```
1.) Types of Numbers in Python
2.) Basic Arithmetic
3.) Object Assignment in Python
```

Types of numbers

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

Throughout this course we will be mainly working with integers or simple float number types.

Here is a table of the two main types we will spend most of our time working with some examples:

Now let's start with some basic arithmetic.

Basic Arithmetic

```
# Addition
print(2+1)

# Subtraction
print(2-1)

# Multiplication
print(2*2)

# Division
print(3/2)
```

```
3
1
4
1.5
```

Arithmetic continued

```
# Powers
2**3
```

```
8
```

```
# Order of Operations followed in Python
2 + 10 * 10 + 3
```

```
105
```

```
# Can use parenthesis to specify orders
(2+10) * (10+3)
```

```
156
```

Variable Assignments

Now that we've seen how to use numbers in Python as a calculator let's see how we can assign names and create variables.

We use a single equals sign to assign labels to variables. Let's see a few examples of how we can do this.

```
# Let's create an object called "a" and assign it the number 5
a = 5
```

Now if I call *a* in my Python script, Python will treat it as the number 5.

```
# Adding the objects
a+a
```

```
10
```

What happens on reassignment? Will Python let us write it over?

```
# Reassignment
a = 10
```

```
# Check
a
```

```
10
```

Data Types & Data Structures

- Everything in Python is an "object", including integers/floats
- Most common and important types (classes)
  - "Single value": None, int, float, bool, str, complex
  - "Multiple values": list, tuple, set, dict
- Single/Multiple isn't a real distinction, this is for explanation
- There are many others, but these are most frequently used

Identifying Data Types

```
a = 42
b = 32.30

print(type(a))#gets type of a
print(type(b))#gets type of b
```

```
<class 'int'>
<class 'float'>
```

Single Value Types

- int: Integers
- float: Floating point numbers
- bool: Boolean values (True, False)
- complex: Complex numbers
- str: String

Lists

Lists can be thought of the most general version of a *sequence* in Python.

Unlike strings, they are mutable, meaning the elements inside a list can be changed!

In this section we will learn about:

```
1.) Creating lists
2.) Indexing and Slicing Lists
3.) Basic List Methods
4.) Nesting Lists
5.) Introduction to List Comprehensions
```

Lists are constructed with brackets [] and commas separating every element in the list.

Let's go ahead and see how we can construct lists!

```
# Assign a list to an variable named my_list
my_list = [1,2,3]
```

We just created a list of integers, but lists can actually hold different object types. For example:

```
my_list = ['A string',23,100.232,'o']
```

Just like strings, the len() function will tell you how many items are in the sequence of the list.

```
len(my_list)
```

```
4
```

Adding New Elements to a list

We use two special commands to add new elements to a list. Let's make a new list to remind ourselves of how this works:

```
my_list = ['one','two','three',4,5]
```

```
# append a value to the end of the list
l = [1, 2.3, ['a', 'b'], 'New York']
l.append(3.1)
print(l)
```

```
[1, 2.3, ['a', 'b'], 'New York', 3.1]
```

```
# extend a list with another list.
l = [1, 2, 3]
l.extend([4, 5, 6])
print(l)
```

```
[1, 2, 3, 4, 5, 6]
```

Slicing

Slicing is used to access individual elements or a rage of elements in a list.

Python supports "slicing" indexable sequences. The syntax for slicing lists is:

- `list_object[start:end:step]` or
- `list_object[start:end]`

start and end are indices (start inclusive, end exclusive). All slicing values are optional.

```
lst = list(range(10)) # create a list containing 10 numbers starting
from 0
print(lst)

print("elements from index 4 to 7:", lst[4:7])
print("alternate elements, starting at index 0:", lst[0::2])
# prints elements from index 0 till last index with a step of 2

print("every third element, starting at index 1:", lst[1::3])
# prints elements from index 1 till last index with a step of 3
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
elements from index 4 to 7: [4, 5, 6]
```

```
alternate elements, starting at index 0: [0, 2, 4, 6, 8]
every third element, starting at index 1: [1, 4, 7]
```

- **`.append`**: add element to end of list
- **`.insert`**: insert element at given index
- **`.extend`**: extend one list with another list

Dictionaries

Now we're going to switch gears and learn about *mappings* called *dictionaries* in Python. If you're familiar with other languages you can think of these Dictionaries as hash tables.

This section will serve as a brief introduction to dictionaries and consist of:

```
1.) Constructing a Dictionary
2.) Accessing objects from a dictionary
3.) Nesting Dictionaries
4.) Basic Dictionary Methods
```

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}
```

```
# Call values by their key
my_dict['key2']
```

```
'value2'
```

We can effect the values of a key as well. For instance:

```
my_dict['key1']=123
my_dict
```

```
{'key1': 123, 'key2': 'value2'}
```

```
# Subtract 123 from the value
my_dict['key1'] = my_dict['key1'] - 123
```

```
#Check
my_dict['key1']
```

```
0
```

A quick note, Python has a built-in method of doing a self subtraction or addition (or multiplication or division). We could have also used += or -= for the above statement. For example:

```
# Set the object equal to itself minus 123
my_dict['key1'] -= 123
my_dict['key1']
```

```
-123
```

Now its your turn to get hands-on with Dictionary, create a empty dicts. Create a new key calle animal and assign a value 'Dog' to it..

```
# Create a new dictionary
d = {}
# Create a new key through assignment
d['animal'] = 'Dog'
```

Set and Booleans

There are two other object types in Python that we should quickly cover. Sets and Booleans.

Sets

Sets are an unordered collection of *unique* elements. We can construct them by using the set() function. Let's go ahead and make a set to see how it works

```
x = set()

# We add to sets with the add() method
x.add(1)

#Show
x
```

```
{1}
```

Note the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

We know that a set has only unique entries. So what happens when we try to add something that is already in a set?

```
# Add a different element
x.add(2)

#Show
x
```

```
{1, 2}
```

```
# Try to add the same element
x.add(1)

#Show
x
```

```
{1, 2}
```

Notice how it won't place another 1 there. That's because a set is only concerned with unique elements! We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

```
# Create a list with repeats
l = [1,1,2,2,3,4,5,6,1,1]

# Cast as set to get unique values
set(l)
```

```
{1, 2, 3, 4, 5, 6}
```

Python Programming Constructs

We'll be talking about

- Looping
- Conditional Statements

Loops and Iterative Statements

If,elif,else Statements

if Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:

"Hey if this case happens, perform some action"

We can then expand the idea further with elif and else statements, which allow us to tell the computer:

"Hey if this case happens, perform some action. Else if another case happens, perform some other action. Else-- none of the above cases happened, perform this action"

Let's go ahead and look at the syntax format for if statements to get a better idea of this:

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action 3
```

```
a = 5
b = 4

if a > b:
    # we are inside the if block
    print("a is greater than b")
elif b > a:
    # we are inside the elif block
    print("b is greater than a")
else:
    # we are inside the else block
    print("a and b are equal")

# Note: Python doesn't have a switch statement
```

```
a is greater than b
```

Indentation

It is important to keep a good understanding of how indentation works in Python to maintain the structure and order of your code. We will touch on this topic again when we start building out functions!

For Loops

A **for** loop acts as an iterator in Python, it goes through items that are in a *sequence* or any other iterable item.

Objects that we've learned about that we can iterate over include strings,lists,tuples, and even built in iterables for dictionaries, such as the keys or values.

We've already seen the **for** statement a little bit in past lectures but now lets formalize our understanding.

Here's the general format for a **for** loop in Python:

```
for item in object:
    statements to do stuff
```

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code.

This item name can then be referenced inside you loop, for example if you wanted to use if statements to perform checks.

Let's go ahead and work through several example of **for** loops using a variety of data object types.

```
#Simple program to find the even numbers in a list

# Initialised the list
list_1 = [2,4,5,6,8,7,9,10]

# Selects one element in list_1
for number in list_1:
    if number % 2 == 0:
    #Checks if it is even. IF even,only then, goes to next step else
performs above step and continues iteration

        print(number,end=' ')

        #prints no if even. end=' ' prints the nos on the same line
with a space in between.
```

```
2 4 6 8 10
```

```
lst1 = [4, 7, 13, 11, 3, 11, 15]
lst2 = []

for index, e in enumerate(lst1):
    if e == 10:
        break
    if e < 10:
        continue
    lst2.append((index, e*e))
else:
    print("out of loop without using break statement")
```

```
lst2

out of loop without using break statement

[(2, 169), (3, 121), (5, 121), (6, 225)]
```

While loops

The **while** statement in Python is one of most general ways to perform iteration. A **while** statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

```
while test:
    code statement
else:
    final code statements
```

Let's look at a few simple while loops in action.

```
x = 0

while x < 10:
    print ('x is currently: ',x,end=' ')
    #end=' ' to put print below statement on the same line after thsi
statement
    print (' x is still less than 10, adding 1 to x')
    x+=1
```

```
x is currently:  0  x is still less than 10, adding 1 to x
x is currently:  1  x is still less than 10, adding 1 to x
x is currently:  2  x is still less than 10, adding 1 to x
x is currently:  3  x is still less than 10, adding 1 to x
x is currently:  4  x is still less than 10, adding 1 to x
x is currently:  5  x is still less than 10, adding 1 to x
x is currently:  6  x is still less than 10, adding 1 to x
x is currently:  7  x is still less than 10, adding 1 to x
x is currently:  8  x is still less than 10, adding 1 to x
x is currently:  9  x is still less than 10, adding 1 to x
```

Thank You !!!