DATA2010: Intro to Python and Anaconda

Lecture Objectives

- Get familiar with the Anaconda and the IDE's within.
- Learn Basic syntax of Python and re-connect Python and R
- Better understading of Numpy and Pandas

Python: the basics

Python is a general purpose programming language that supports rapid development of scripts and applications.

Python's main advantages:

- Open Source software, supported by Python Software Foundation
- Available on all major platforms (ie. Windows, Linux and MacOS)
- It is a general-purpose programming language, designed for readability
- Supports multiple programming paradigms ('functional', 'object oriented')
- Very large community with a rich ecosystem of third-party packages

Interpreter

Python is an interpreted language* which can be used in two ways:

- "Interactive" Mode: It functions like an "advanced calculator", executing one command at a time:
- "Scripting" Mode: Executing a series of "commands" saved in text file, usually with a `.py` extension after the name of your file:

Using interactive Python in Jupyter-style notebooks

A convenient and powerful way to use interactive-mode Python is via a Jupyter Notebook, or similar browser-based interface.

This particularly lends itself to data analysis since the notebook records a history of commands and shows output and graphs immediately in the browser.

There are several ways you can run a Jupyter(-style) notebook - locally installed on your computer or hosted as a service on the web. Today we will use a Jupyter notebook service provided by Google: https://colab.research.google.com (Colaboratory).

Jupyter-style notebooks: a quick tour

Go to https://colab.research.google.com and login with your Google account.

Select ***NEW NOTEBOOK*** → ***NEW PYTHON 3 NOTEBOOK*** - a new notebook will be created.

Type some Python code in the top cell, eg:

In [1]:

```python
print("Hello Jupyter!")
```

```
Hello Jupyter!
```

Shift-Enter to run the contents of the cell

You can add new cells.

Insert → Insert Code Cell

NOTE: When the text on the left hand of the cell is: `In [*]` (with an asterisk rather than a number), the cell is still running. It's usually best to wait until one cell has finished running before running the next.

Let's begin writing some code in our notebook.

Prerequisite

Check the system path and version of jupyter notebook and python

In [2]:

```python
import sys
print(sys.executable)
print(sys.version)
print(sys.version_info)
```

```
E:\Anaconda\envs\py3.6\python.exe
3.6.13 |Anaconda, Inc.| (default, Mar 16 2021, 11:37:27) [MSC v.1916 64 bit (AMD64)]
sys.version_info(major=3, minor=6, micro=13, releaselevel='final', serial=0)
```

In [3]:

```python
from platform import python_version
print(python_version())
```

```
3.6.13
```

Or most simple way to check

In [4]:

```
!python -V
```

```
Python 3.6.13 :: Anaconda, Inc.
```

> The exclamation/bang(!) simply means to run a conda install command.

Comments in Python

In [5]:

```python
# This is a single line comment
print("Hello, World!")
```

```
Hello, World!
```

In [6]:

```python
"""
```

```
This is a multiline
comment.

"""

print("Hello, World!")
```

```
Hello, World!
```

Importing Libraries or Modules

In [7]:

```
import numpy as np
import pandas as pd
```

> The Word "import" allocates the specific library installed in the environment

In [8]:

```
# Import the pyplot model class from matplotlib
from matplotlib import pyplot as plt
```

In [9]:

```
# We run this to suppress various deprecation warnings and keeps our notebook cleaner
import warnings
warnings.filterwarnings('ignore')
```

Basic Data Structures

Numbers

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.

Examples	Number Type
1, 2, 100, -100	Integer
1.2,-0.5,2e2,3E2	Floating Point Numbers

Basic Arithmetic

In [10]:

```
# Addition
2 + 1
```

Out[10]:

```
3
```

In [11]:

```
# Subtraction
2 - 1
```

Out[11]:

```
1
```

In [12]:

```
# Multiplication
2 * 2
```

Out[12]:

```
4
```

In [13]:

```
# Division
3 / 2
```

Out[13]:

```
1.5
```

In [14]:

```
# Floor Division
7 // 4
```

Out[14]:

```
1
```

In [15]:

```
# Modulo
7 % 4
```

Out[15]:

```
3
```

In [16]:

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

Out[16]:

```
8
```

In [17]:

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

Out[17]:

```
156
```

Variable Assignment

In [18]:

```
# Let's create an object called "x" and assign it the number 5
x = 20
```

In [19]:

```
# Adding the objects
x + x
```

Out[19]:

```
40
```

In [20]:

```
# Reassignment
x = 20
x = x + x
```

Rules for naming variables

1. Names can not start with a number.

2. There can be no spaces in the name, use _ instead.
3. Can't use any of these symbols :'",<>/?|()!@#$%^&*~-+
4. It's considered best practice (PEP8) that names are lowercase.
5. Avoid using words that have special meaning in Python like "list" and "str"

Dynamic Typing

```
In [21]:
num = 2
num
```

```
Out[21]: 2
```

```
In [22]:
num = "This is a number"
num
```

```
Out[22]: 'This is a number'
```

Determining type of variable

Python provides a build in function called `type` to check the type of variable.

```
In [23]:
x = 5
type(x)
```

```
Out[23]: int
```

```
In [24]:
y = 5.5
type(y)
```

```
Out[24]: float
```

```
In [25]:
t = (4, 5)
type(t)
```

```
Out[25]: tuple
```

Some common data types are:

- int (for integer)
- float
- str (for string)
- list
- tuple
- dict (for dictionary)
- set
- bool (for Boolean True/False)

Strings

Strings in Python are actually a sequence, used to record textual info.

In [26]:

```
# String using single quotes
'I am a string'
```

Out[26]:

```
'I am a string'
```

In [27]:

```
# String using double quotes
"I am also a string"
```

Out[27]:

```
'I am also a string'
```

In [28]:

```
# Be careful with quotes!
'I'm using single quotes, but this will create an error'
```

```
  File "<ipython-input-28-d4e01b799727>", line 2
    'I'm using single quotes, but this will create an error'
       ^
SyntaxError: invalid syntax
```

In [29]:

```
# You have to use the escape(\) binder
'I\'m using single quotes, but this won\'t create an error'
```

Out[29]:

```
"I'm using single quotes, but this won't create an error"
```

Python provides a `print` method to print a string.

In [30]:

```
print('Hello, Welcome to the workshop')
```

```
Hello, Welcome to the workshop
```

In [31]:

```
print('This is an example how to use print method.\nI will be printed in next line because o
```

```
This is an example how to use print method.
I will be printed in next line because of \n
```

Python provides a `len` method to calculate the length of a string.

In [32]:

```
len('DATA2010 - Fall2021')
```

Out[32]:

```
19
```

Indexing and Slicing in Strings

In [33]:

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

In [34]:

```
s
```

Out[34]:

```
'Hello World'
```

In [35]:

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

Out[35]:

```
'H'
```

We use `:` to perform Slicing

In [36]:

```
# Grab everything past the first term all the way to the length of s which is len(s) However
s[1:]
```

Out[36]:

```
'ello World'
```

In [37]:

```
# Grab everything UP TO the 3rd index
s[:3]
```

Out[37]:

```
'Hel'
```

In [38]:

```
# What if we done provide any index in slicing?

s[:]
```

Out[38]:

```
'Hello World'
```

Python also supports Negative Indexing

In [39]:

```
#Last letter (one index behind 0 so it loops back around)
s[-1]
```

Out[39]:

```
'd'
```

Grab everything but the last letter

In [40]:

```
s[:-1]
```

Out[40]:

```
'Hello Worl'
```

Step size in Slicing. Grab everything, but go in step sizes of 2

In [41]:

```
s[::2]
```

Out[41]:

```
'HloWrd'
```

Reverse a string

In [42]:

```
s[::-1]
```

Out[42]:

```
'dlroW olleH'
```

String Properties

String are Immutable !

Immutability may be used to ensure that an object remains constant throughout your program. The values of mutable objects can be changed at any time and place, whether you expect it or not.

In [43]:

```
#Concate Strings

s = s + ' concatenate me!!'
```

example

In [44]:

```
s
```

Out[44]:

```
'Hello World concatenate me!!'
```

In [45]:

```
# String Multiplication


label = 'p'
label * 5
```

Out[45]:

```
'ppppp'
```

String Methods

Upper Case

Python has a build in method `upper` to get uppercase of a string.

In [46]:

```
# Upper Case a string
s.upper()
```

Out[46]:

```
'HELLO WORLD CONCATENATE ME!!'
```

> Note these methods don't change the original string!

In [47]:

```
s
```

Out[47]:

```
'Hello World concatenate me!!'
```

In [48]:

```
s.lower()
```

Out[48]:

```
'hello world concatenate me!!'
```

Split

Extract words from string

In [49]:

```
# Split a string by blank space (this is the default)
s.split()
```

Out[49]:

```
['Hello', 'World', 'concatenate', 'me!!']
```

In [50]:

```
# Split by a specific element (doesn't include the element that was split on)
s.split('W')
```

Out[50]:

```
['Hello ', 'orld concatenate me!!']
```

String Formatting

String formatting lets you inject items into a string rather than trying to chain items together.

In [51]:

```
player = 'Thomas'
points = 33
```

In [52]:

```
'Last night, '+player+' scored '+str(points)+' points.'  # concatenation
```

Out[52]:

```
'Last night, Thomas scored 33 points.'
```

In [53]:

```
f'Last night, {player} scored {points} points.'          # string formatting
```

Out[53]:

```
'Last night, Thomas scored 33 points.'
```

> There are **three ways** to perform string formatting.

1. The oldest method involves placeholders using the modulo % character.
2. An improved technique uses the .format() string method.
3. The newest method, introduced with Python 3.6, uses formatted string literals, called f-strings.

First Method* using %

In [54]:

```
print("I'm going to inject %s here." %'something')
```

```
I'm going to inject something here.
```

In [55]:

```
x, y = 'some', 'more'
print("I'm going to inject %s text here, and %s text here."%(x,y))
```

```
I'm going to inject some text here, and more text here.
```

Second Method using .format()

In [56]:

```
print('This is a string with an {}'.format('insert'))
```

```
This is a string with an insert
```

In [57]:

```
print('The {2} {1} {0}'.format('fox','brown','quick'))
```

```
The quick brown fox
```

Third Method using f-strings

In [58]:

```
name = 'Asif'
```

```
print(f"He said his name is {name}.")
```

```
He said his name is Asif.
```

List

List is an ordered sequence of elements.

In [59]:

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

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

In [60]:

```
my_list[1] = 5
my_list
```

Out[60]:

```
[1, 5, 3]
```

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

In [61]:

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

Out[61]:

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

Just like strings, list also has `len` to find length of string.

In [62]:

```
my_list = ['This', 'is', 'a', 'DATA2010', 'Lab']
len(my_list)
```

Out[62]:

```
5
```

Indexing and Slicing in List

Indexing and slicing work just like in strings.

In [63]:

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

Out[63]:

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

In [64]:

```
# Grab element at index 0
my_list[0]
```

Out[64]:

```
'one'
```

In [65]:

```
# Grab index 1 and everything past it
my_list[1:]
```

Out[65]:

```
['two', 'three', 4, 5]
```

We can also use + to concatenate lists, just like we did for strings.

In [66]:

```
my_list + ['new item']
```

Out[66]:

```
['one', 'two', 'three', 4, 5, 'new item']
```

> **Note**: *This doesn't actually change the original list!*

We can also use the * for a duplication method similar to strings:

In [67]:

```
# Make the list double
my_list * 2
```

Out[67]:

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

List Methods

append

> Add an item to the end of the list

In [68]:

```
# Append
my_list.append('append me!')
my_list
```

Out[68]:

```
['one', 'two', 'three', 4, 5, 'append me!']
```

pop

> Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list.

In [69]:

```
# Pop off the 0 indexed item
my_list.pop(0)
```

Out[69]:

```
'one'
```

In [70]:

```
# Assign the popped element, remember default popped index is -1
popped_item = my_list.pop()
popped_item
```

Out[70]:

```
'append me!'
```

reverse

> Reverse the elements of the list in place.

In [71]:

```
new_list = ['a','e','x','b','c']
```

In [72]:

```
# Use reverse to reverse order (this is permanent!)
```

```
new_list.reverse()
new_list
```

Out[72]:

```
['c', 'b', 'x', 'e', 'a']
```

sort

> Sort the items of the list in place

In [73]:

```
# Use sort to sort the list (in this case alphabetical order, but for numbers it will go asce
new_list.sort()

new_list
```

Out[73]:

```
['a', 'b', 'c', 'e', 'x']
```

Nested List

In [74]:

```
# Let's make three lists
lst_1=[1,2,3]
lst_2=[4,5,6]
lst_3=[7,8,9]
```

In [75]:

```
# Make a list of lists to form a matrix
matrix = [lst_1,lst_2,lst_3]


matrix
```

Out[75]:

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

In [76]:

```
# Grab first item in matrix object
matrix[0]
```

Out[76]:

```
[1, 2, 3]
```

List Comprehension

In [77]:

```
nums = [1, 2, 3, 4]

squares = [ n * n for n in nums ]


squares
```

Out[77]:

```
[1, 4, 9, 16]
```

Dictonary

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds **key:value** pair.

Dictonaries are like Hash Tables used in other languages.

In [78]:

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

In [79]:

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

Out[79]:

```
'value2'
```

Its important to note that dictionaries are **very flexible** in the data types they can hold.

In [80]:

```
my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}
```

In [81]:

```
# Let's call items from the dictionary
my_dict['key3']
```

Out[81]:

```
['item0', 'item1', 'item2']
```

In [82]:

```
# Can call an index on that value
my_dict['key3'][0]
```

Out[82]:

```
'item0'
```

> To see the power and flexibility of Python lets see nested dictonaries

In [83]:

```
# Dictionary nested inside a dictionary nested inside a dictionary
d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

Let's see how we can grab that value:

In [84]:

```
# Keep calling the keys
d['key1']['nestkey']['subnestkey']
```

Out[84]:

```
'value'
```

Dictonary Methods

In [85]:

```
# Create a typical dictionary
d = {'key1':1,'key2':2,'key3':3}
```

In [86]:

```
# Method to return a list of all keys
d.keys()
```

Out[86]:

```
dict_keys(['key1', 'key2', 'key3'])
```

In [87]:

```
# Method to grab all values
```

```
d.values()
```

Out[87]:

```
dict_values([1, 2, 3])
```

In [88]:

```
# Method to return tuples of all items
d.items()
```

Out[88]:

```
dict_items([('key1', 1), ('key2', 2), ('key3', 3)])
```

Tuples

In Python tuples are very similar to lists, however, unlike lists they are immutable meaning they can not be changed.

The construction of a tuples use () with elements separated by commas.

In [89]:

```
# Create a tuple
t = (1,2,3)

t
```

Out[89]:

```
(1, 2, 3)
```

In [90]:

```
# Check len just like a list

len(t)
```

Out[90]:

```
3
```

> *Can also **mix object types***

In [91]:

```
t = ('one',2)

# Show
t
```

Out[91]:

```
('one', 2)
```

*Use **indexing** just like we did in lists*

In [92]:

```
t[0]
```

Out[92]:

```
'one'
```

***Slicing** just like a list*

In [93]:

```
t[-1]
```

Out[93]:

```
2
```

Tuple Methods

Tuple has two built-in methods.

index

In [94]:

```
# Use .index to enter a value and return the index
t.index('one')
```

Out[94]:

```
0
```

count

In [95]:

```
# Use .count to count the number of times a value appears
t.count('one')
```

Out[95]:

```
1
```

Tuples are Immutable!

In [96]:

```
t[0]= 'change'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-96-1257c0aa9edd> in <module>
----> 1 t[0]= 'change'

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

> Because of this immutability, tuples can't grow. Once a tuple is made we can not add to it.

In [97]:

```
t.append('nope')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-97-b75f5b09ac19> in <module>
----> 1 t.append('nope')

AttributeError: 'tuple' object has no attribute 'append'
```

Use of Tuple

If in your program you are passing around an object and need to make sure it does not get changed, then a tuple becomes your solution. It provides a convenient source of data integrity.

Sets

Sets are an unordered collection of unique elements.

In [98]:

```
x = set()

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

```
#Show
x
```

Out[98]:

```
{1}
```

In [99]:

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

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


#Show
x
```

Out[99]:

```
{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:

In [100…

```
# Create a list with repeats
list1 = [1,1,2,2,3,4,5,6,1,1]


# Cast as set to get unique values
set(list1)
```

Out[100…

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

Booleans

In [101…

```
# Boolean values are primitives (Note: the capitalization)
True  # => True
False  # => False
```

Out[101…

```
False
```

In [102…

```
# negate with not
not True   # => False
not False  # => True
```

Out[102…

```
True
```

In [103…

```
# Boolean Operators
True and False  # => False
False or True   # => Truev
```

Out[103…

```
True
```

> Note "and" and "or" are case-sensitive

Operators

Operators are special symbols in python that carry out arthimetic and logical operations.

Operator Types

1. Arithmetic Operators
2. Comparison Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Special Operators

Arithmetic Operators

In [104…

```
x = 15
y = 6

# Addition Operator
print('x + y = ', x + y)

# Subtraction Operator
print('x - y = ', x - y)

# Multiplication Operator
print('x * y = ', x * y)

# Division Operator
print('x / y = ', x / y) # True Division
print('x // y =', x//y) # Class Division

# Exponential Operator
print('x ** y = ', x ** y)
```

```
x + y =  21
x - y =  9
x * y =  90
x / y =  2.5
x // y = 2
x ** y =  11390625
```

Comparison Operators

In [105…

```
x = 12
y = 10

# Greater Than Operator
print('x > y = ',x > y)

# Greater Than or Equal To
print('x >= y', x >= y)

# Lesser Than Operator
print('x < y = ',x < y)

# Lesser Than or Equal To
print('x <= y', x <= y)
```

```
# Equal To Operator
print('x == y ',x == y)

# Not Equal To
print('x != y', x != y)
```

```
x > y =  True
x >= y True
x < y =  False
x <= y False
x == y  False
x != y True
```

Logical Operators

```
and - True if both operands are true
or - True if one of the operand is true
not - True if operand is false
```

Bitwise Operators

```
& -> Bitwise AND
| -> Bitwise OR
~ -> Bitwise NOT
^ -> Bitwise XOR
>> -> Bitwise Right Shift
<< -> Bitwise Left Shift
```

In [106…

```
x = 10
y = 4

# Bitwise AND
print('x & y = ', x & y)
# Bitwise OR
print('x | y = ', x | y)
# Bitwise NOT
print('~ x = ', ~ x)
# Bitwise XOR
print('x ^ y = ', x ^ y)
# Right shift
print('x >> y = ', x >> y)
# Left Shift
print('x << y = ', x << y)
```

```
x & y =  0
x | y =  14
~ x =  -11
x ^ y =  14
x >> y =  0
x << y =  160
```

Assignment Operators

In [107…

```
a = 5
print(a)
a += 5 # a = a + 5
print(a)
a -= 5 # a = a - 5
print(a)
a *= 5 # a = a * 5
print(a)
```

```
a /= 5 # a = a / 5
print(a)
a //= 2 # a = a // 2
print(a)
a %= 1 # a = a % 1
print(a)
a = 10
a **= 2 # a = a ** 2
print(a)
```

```
5
10
5
25
5.0
2.0
0.0
100
```

Identity Operators

```
is - True if the operands are identical
is not - True if the operands are not identical
```

In [108…

```
x1 = 2
y1 = 2
x2 = 'Hello'
y2 = "Hello"
x3 = [1,2,3]
y3 = (1,2,3)

print('x1 is y1 = ', x1 is y1)
print('x1 is y2 = ', x1 is y2)
print('x3 is not y3 = ', x3 is not y3)
```

```
x1 is y1 =  True
x1 is y2 =  False
x3 is not y3 =  True
```

Membership Operators

```
in - True if value / variable is found in sequence
not in - True if value / variable is not found in the sequence
```

In [109…

```
x = 'Hello World'
y = {1:'a',2:'b'}

print("'H' in x ", 'H' in x)
print('hello not in x ','hello' not in x)
print('1 in y = ',1 in y)
print('a in y = ',a in y)
```

```
'H' in x  True
hello not in x  True
1 in y =  True
a in y =  False
```

Functions

- Function is a group of related statements that perform a specific task.

- Function help break large programs into smaller and modular chunks
- Function makes the code more organised and easy to manage
- Function avoids repetition and there by promotes code reusability

Two types of functions

1. Built-In Functions
2. User Defined Functions

Function Syntax

```
def function_name(arguments):
    '''This is the docstring for the function'''
    # note the indentation, anything inside the function must be indented
    # function code goes here
    ...
    return

# calling the function
function_name(arguments)
```

In [110…

```
# Simple Function
def greet():
    '''Simple Greet Function'''
    print('Hello World')

greet()
```

```
Hello World
```

In [111…

```
# Function with arguments
def greet(name):
    '''Simple Greet Function with arguments'''
    print('Hello ', name)

greet('John')
```

```
Hello  John
```

In [112…

```
# Function with return statement
def add_numbers(num1,num2):
    return num1 + num2

print(add_numbers(2,3.0))
```

```
5.0
```

In [113…

```
# Since arguments are not strongly typed you can even pass a string
print(add_numbers('Hello','World'))
```

```
HelloWorld
```

Scope and Lifetime of Variables

Variables in python has local scope which means parameters and variables defined inside the function is not visible from outside.

Lifetime of a variable is how long the variable exists in the memory. Lifetime of variables defined inside the function exists as long as the function executes. They are destroyed once the function is returned.

In [114…

```
def myfunc():
    x = 5
    print('Value inside the function ',x)

x = 10
myfunc()
print('Value outside the function',x)
```

```
Value inside the function  5
Value outside the function 10
```

In [115…

```
def myfunc():
    #x = 5
    print('Value inside the function ',x)

x = 10
myfunc()
print('Value outside the function',x)
```

```
Value inside the function  10
Value outside the function 10
```

Global Variable

Variables declared inside the function are not available outside. The following example will generate an error.

In [116…

```
def myfunc():
    y = 5
    print('Value inside the function ',y)

myfunc()
print('Value outside the function',y)
```

```
Value inside the function  5
Value outside the function {1: 'a', 2: 'b'}
```

In [117…

```
def myfunc():
    global z
    z = 5
    print('Value inside the function ',z)

#z = 10
myfunc()
print('Value outside the function',z)
```

```
Value inside the function  5
Value outside the function 5
```

Now let's play with Pandas and Numpy