Python Course 5/2018

Part (I)

By AmrZakaria Zaky

Index

10- Debugging (Reading error messages and learning how to debug)

11- working with Numpy

12- URL

1- Introduction to Python:

1.1- What is Python?

1.1.1- Python is a popular programming language. It was created in 1991 by Guido van Rossum.

1.1.2- It is used for :

* web development (server-side).

* software development.

* mathematics.

* system scripting.

1.1.3- Python is also considered a high-level language, meaning it take scare of a lot of the grunt work involved in programming.

1.2- What can Python do?

1.2.1- Python can be used on a server to create web applications.

1.2.2- Python can be used alongside software to create workflows.

1.2.3- Python can connect to database systems. It can also read and modify files.

1.2.4- Python can be used to handle big data and perform complex mathematics.

1.2.5- Python can be used for rapid prototyping, or for production-ready software development.

1.3- Why Python?

1.3.1- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).

1.3.2- Python has a simple syntax similar to the English language.

1.3.3- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.

1.3.4- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.

1.3.5- Python can be treated in a procedural way, an object-orientated way or a functional way.

1.4- Python Syntax compared to other programming languages

1.4.1- Python was designed to for readability, and has some similarities to the English language with influence from mathematics.

1.4.2- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.

1.4.3- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

1.5- Interpreted vs. Compiled

1.5.1- Compiled languages are usually used for low-level programming and languages are compiled, meaning the source codethe developer writes is converted into machine language by the compiler.

1.5.2- The interpreter processes the code line by line and creates abytecode.

(Bytecode is an in-between “language” that isn’t quite machine code but it isn’t the source code)

2- Data types and values:

2.1- Numbers

2.1.1- There are three numeric types in Python:

* int

* float

* complex

- Example (int):

```
x = 1

y = 35656222554887711

z = -3255522
```

- Example (float):

```
x = 1.10

y = 1.0
```

z = -35.59

x = 35e3

y = 12E4

z = -87.7e100

- Example (Complex):

x = 3+5j

y = 5j

z = -5j

2.1.2- Variable Names:

* A variable name must start with a letter or the underscore character.

* A variable name cannot start with a number.

* A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).

* Variable names are case-sensitive (age, Age and AGE are three different variables).

2.2- String

String literals in python are surrounded by either single quotation marks, or double quotation marks.

('hello' is the same as "hello")

- Example (String):

print("Hallow")

Output >>> Hallow

a = "hello"

print(a[1])

Output >>>e

b = "world"

print(b[2:5])

Output >>>rld

```
a = " Hello, World! "
print(a.strip())
```

Output >>>Hello, World!

```
a = "Hello, World!"
print(len(a))
```

Output >>>13

```
a = "Hello, World!"
print(a.lower())
```

Output >>>hello, world!

```
a = "Hello, World!"
print(a.upper())
```

Output >>>HELLO, WORLD!

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Output >>>Jello, World!

```
a = "Hello, World!"
b = a.split(",")
print(b)
```

Output >>>['Hello', 'World!']

2.3- List

* List is an collection which is ordered and changeable. Allows duplicate members.
* contain a series of values. List variables are declared by using brackets [ ] following the variable name.

- Example (List):

```
thislist = ["apple", "banana", "cherry"]
```

```
print(thislist)
```

Output >>>['apple', 'banana', 'cherry']

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

Output >>>['apple', 'blackcurrant', 'cherry']

```
mylist = ['Rhino', 'Grasshopper', 'Flamingo', 'Bongo']
B = len(mylist) # This will return the length of the list which is 3. The index is 0, 1, 2, 3.
printmylist[1]
```

Output >>>'Grasshopper'

```
printmylist[0:2]
```

Output >>>['Rhino', 'Grasshopper', 'Flamingo']

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

Output >>>['apple', 'banana', 'cherry']

```
thislist = list(("apple", "banana", "cherry"))
thislist.append("damson")
print(thislist)
```

Output >>>['apple', 'banana', 'cherry', 'damson']

```
thislist = list(("apple", "banana", "cherry"))
thislist.remove("banana")
print(thislist)
```

Output >>>['apple', 'cherry']

2.4- Tuple

* Tuple is a collection which is ordered and unchangeable. Allows duplicate members.

* Tuples are fixed in size once they are assigned.

- Example (Tuple):

```
thistuple = ("apple", "banana", "cherry")

print(thistuple)
```

Output >>>('apple', 'banana', 'cherry')

```
thistuple = ("apple", "banana", "cherry")

print(thistuple[1])
```

Output >>>banana

```
thistuple = ("apple", "banana", "cherry")

thistuple[1] = "blackcurrant"

print(thistuple)
```

Output >>>('apple', 'banana', 'cherry')

2.5- Dictionary

* Dictionary is a collection which is unordered, changeable and indexed. No duplicate members.

* Dictionaries in Python are lists of Key:Value pairs. This is a very powerful datatype to hold a lot of related information that can be associated through keys.

- Example (Dictionary):

```
thisdict ={"apple": "green","banana": "yellow","cherry": "red"}

print(thisdict)
```

Output >>>{'apple': 'green', 'banana': 'yellow', 'cherry': 'red'}

```
thisdict ={"apple": "green","banana": "yellow","cherry": "red"}

thisdict["apple"] = "red"

print(thisdict)
```

Output >>>{'apple': 'red', 'banana': 'yellow', 'cherry': 'red'}

```
thisdict = dict(apple="green", banana="yellow", cherry="red")
print(thisdict)
```

Output >>>{'apple': 'green', 'banana': 'yellow', 'cherry': 'red'}

```
thisdict = dict(apple="green", banana="yellow", cherry="red")
thisdict["damson"] = "purple"
print(thisdict)
```

Output >>>{'apple': 'green', 'banana': 'yellow', 'cherry': 'red', 'damson': 'purple'}

```
thisdict = dict(apple="green", banana="yellow", cherry="red")
del(thisdict["banana"])
print(thisdict)
```

Output >>>{'apple': 'green', 'cherry': 'red'}

```
thisdict = dict(apple="green", banana="yellow", cherry="red")
print(len(thisdict))
```

Output >>>3

```
room_num = {'john': 425, 'tom': 212}
room_num['john'] = 645
```

Output >>>

```
print (room_num['tom']) # print the value of the 'tom' key.
room_num['isaac'] = 345 # Add a new key 'isaac' with the associated value.
print (room_num.keys()) # print out a list of keys in the dictionary.
print ('isaac' in room_num) # test to see if 'issac' is in the dictionary.  This returns true.
```

2.6- Sets

* Set is a collection which is unordered and unindexed. No duplicate members.

- Example (Sets):

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Output >>>{'cherry', 'banana', 'apple'}

```
thisset = set(("apple", "banana", "cherry"))
print(thisset)
```

Output >>>{'apple', 'banana', 'cherry'}

```
thisset = set(("apple", "banana", "cherry"))
thisset.add("damson")
print(thisset)
```

Output >>>{'cherry', 'banana', 'apple', 'damson'}

```
thisset = set(("apple", "banana", "cherry"))
thisset.remove("banana")
print(thisset)
```

Output >>>{'cherry', 'apple'}

```
thisset = set(("apple", "banana", "cherry"))
print(len(thisset))
```

Output >>>3

3- Conditionals and operators

3.1- Python supports the usual logical conditions from mathematics:

Equals: (a == b)

Not Equals: (a != b)

Less than: (a < b)

Less than or equal to: (a <= b)

Greater than: (a > b)

Greater than or equal to: (a >= b)

3.2- The elif keyword is pythons way of saying "if the previous conditions were not true, then do this condition".

- Example (if ,Elif ,else):

```
a = 33
b = 33
if b > a:
        print("b is greater than a")
elif a == b:
        print("a and b are equal")
```

Output >>>a and b are equal

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```

Output >>>a is greater than b

3.3- Python Operators:

3.3.1- Operators are used to perform operations on variables and values.

3.3.2- Python divides the operators in the following groups:

*Arithmetic operators

*Assignment operators

*Comparison operators

*Logical operators

*Identity operators

*Membership operators

*Bitwise operators

- Example (operators**):**

* Python Comparison Operators:

SN	Operator	Name	Example
1	==	Equal	x = y
2	!=	Not equal	x != y
3	<>	Not equal	x <> y
4	>	Greater than	x > y
5	<	Less than	x < y
6	>=	Greater than or equal to	x >= y
7	<=	Less than or equal to	x <= y

* Python Logical Operators

SN	Operator	Name	Example
1	and	Returns True if both statements are true	x < 5 and x < 10
2	or	Returns True if one of the statements is true	x < 5 or x < 4
3	not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

* Python Identity Operators

SN	Operator	Name	Example
1	Is	Returns true if both variables are the same object	x is y
2	is not	Returns false if both variables are the same object	x is not y

* Python Membership Operators

SN	Operator	Name	Example
1	in	Returns True if a sequence with the specified value	x in y
2	not in	Returns False if a sequence with the specified value	x not in y

* Python Bitwise Operators

SN	Operator	Name	Example
1	&	AND	Sets each bit to 1 if both bits
2	|	OR	Sets each bit to 1 if one of two
3	^	XOR	Sets each bit to 1 if only one of
4	~	NOT	Inverts all the bits
5	<<	Zero fill left shift	Shift left by pushing zeros in
6	>>	Signed right shift	Shift right by pushing copies

4- Building loops

4.1- Python has two primitive loop commands:

4.1.1- while loops

* With the while loop we can execute a set of statements as long as a condition is true.

4.1.2- for loops

* A for loop is used for iterating over a sequence.

* With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

- Example (while loops):

```
i = 1
whilei< 6:
    print(i)
    i += 1
```

Output >>1 2 3 4 5

```
i = 1
whilei< 6:
    print(i)
    ifi == 3:
    break
    i += 1
```

Output >>1 2 3

```
i = 0
whilei< 6:
    i += 1
    ifi == 3:
    continue
    print(i)
```

Output >>1 2 3 4 5 6

- Example (for loops):

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

Output >> apple banana cherry

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
    break
    print(x)
```

Output >> apple

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
    continue
    print(x)
```

Output >> apple cherry

```
for x in range(2, 6):
    print(x)
```

Output >> 2 3 4 5

```
for x in range(2, 30, 3):
    print(x)
```

Output >> 2 5 8 11 14 17 20 23 26 29

5- Defining functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

5.1- Creating a Function

- Example (Creating a Function):

* In Python a function is defined using the **def** keyword.

```
def my_function():
    print("Hello from a function")
```

5.2- Calling a Function

* To call a function, use the function name followed by parenthesis.

- Example (Calling a Function):

```
def my_function():
    print("Hello from a function")
my_function()
```

Output >>Hello from a function

5.3- Parameters

* Information can be passed to functions as parameter.

* Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

- Example (Parameters):

```
defmy_function(fname):
print(fname + " Refsnes")
my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Output >>Emil Refsnes Tobias Refsnes Linus Refsnes

5.4- Default Parameter Value

* The following example shows how to use a default paramter value.

* If we call the function without parameter, it uses the default value.

- Example (Default Parameter):

```
defmy_function(country = "Norway"):
    print("I am from " + country)
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil"):
```

Output >>

I am from Sweden

I am from India

I am from Norway

I am from Brazil

5.5- Return Values

* To let a function return a value, use the return statement.

- Example (Return Values):

```
defmy_function(x):
    return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Output >> 15 25 45

5.6- Lambda Functions

* In python, the keyword lambda is used to create what is known as anonymous functions.

* These are essentially functions with no pre-defined name.

* They are good for constructing adaptable functions, and thus good for event handling.

- Example (Lambda Functions):

```
myfunc = lambda i: i*2
print(myfunc(2))
```

Output >> 4

```
myfunc = lambda x,y: x*y
print(myfunc(3,6))
```

Output >> 18

```
defmyfunc(n):
    return lambda i: i*n
doubler = myfunc(2)
tripler = myfunc(3)
val = 11
```

print("Doubled: " + str(doubler(val)) + ". Tripled: " +str(tripler(val)))

Output >>Doubled: 22. Tripled: 33

6- Creating modules

* Consider a module to be the same as a code library.

* A file containing a set of functions you want to include in your application.

6.1- Create a Module

* To create a module just save the code you want in a file with the file extension .py.

- Example (Create a Module):

```
defSayText(name):
    print("Hi, " + name)
```

Output >>Save this code in a file named [mod.py]

6.2- Use a Module

* Now we can use the module we just created, by using the import statement.

- Example (Use a Module):

```
importmod
mod.SayText("Aly"):
```

Output >>Hi, Aly

6.3- Object in Module

* The module can contain functions, as already described, but also variables and objects.

- Example (Objectin Module):

- Save this code in the file mod.py.

```
class Person:
    name = "Aly"
    age = "36"
    country = "EG"
```

```
import mod
    a = mod.age
    print(a)
```

Output >>36

6.4- Re-naming a Module

* You can name the module file whatever you like, but it must have the file extension .py.

* You can create an alias when you import a module, by using the **as** keyword.

- Example (Re-naming a Module):

```
importmod as azz
    a = azz.Person.age
    print(a)
```

Output >> 36

6.5- Built-in Modules

* There are several built-in modules in Python, which you can import whenever you like.

- Example (Built-in Modules):

```
import platform
    x = platform.system()
    print(x)
```

Output >>Windows

6.6- Using the dir() Function

* There is a built-in function to list all the function names (or variable names) in a module [dir()].

* List all the defined names belonging to the platform module

- Example (dir() Function):

```
import platform
    x = dir(platform)
```

```
print(x)
```

Output >>['DEV_NULL', '_UNIXCONFDIR', 'WIN32_CLIENT_RELEASES', 'WIN32_SERVER_RELEASES', '__builtins__', '__cached__', '__copyright__', '__doc__', '__file__', '__loader__', '__name__', '__package __', '__spec__', '__version__', '_default_architecture', '_dist_try_harder', '_follow_symlinks', '_ironpython26_sys_version_parser', '_ironpython_sys_version_parser', '_java_getprop', '_libc_search', '_linux_distribution', '_lsb_release_version', '_mac_ver_xml', '_node', '_norm_version', '_perse_release_file', '_platform', '_platform_cache', '_pypy_sys_version_parser', '_release_filename', '_release_version', '_supported_dists', '_sys_version', '_sys_version_cache', '_sys_version_parser', '_syscmd_file', '_syscmd_uname', '_syscmd_ver', '_uname_cache', '_ver_output', 'architecture', 'collections', 'dist', 'java_ver', 'libc_ver', 'linux_distribution', 'mac_ver', 'machine', 'node', 'os', 'platform', 'popen', 'processor', 'python_branch', 'python_build', 'python_compiler', 'python_implementation', 'python_revision', 'python_version', 'python_version_tuple', 're', 'release', 'subprocess', 'sys', 'system', 'system_aliases', 'uname', 'uname_result', 'version', 'warnings', 'win32_ver']

7- File input/output (I/O)

* Python has several functions for creating, reading, updating, and deleteing files.

* File Handling:

- The key function for working with files in Python is the open() function.

- The open() function takes two parameters; filename, and mode.

- There are four different methods (modes) for opening a file:

- "r" - Read - Default value. Opens a file for reading, error if the file does not exist

- "a" - Append - Opens a file for appending, creates the file if it does not exist

- "w" - Write - Opens a file for writing, creates the file if it does not exist

- "x" - Create - Creates the specified file, returns an error if the file exist

- In addition you can specify if the file should be handled as binary or text mode

- "t" - Text - Default value. Text mode

- "b" - Binary - Binary mode (e.g. images)

- To open a file for reading it is enough to specify the name of the file:

- Because "r" for read, and "t" for text are the default values, you do not need to specify them.

7.1- File Handling

- Example (File Handling):

```
f = open("demofile.txt")

f = open("demofile.txt", "rt")
```

7.2- Open a File on the Server

* To open the file, use the built-in open() function.
* The open() function returns a file object, which has a read() method for reading the content of the file.
* By default the read() method returns the whole text, but you can also specify how many character you want to return.
* You can return one line by using the readline() method.
* Read two lines of the file.
* By looping through the lines of the file, you can read the whole file, line by line.

- Example (Open a File on the Server):

* Asume we have the following file, located in the same folder as Python.
* demo.txt

/**************************/

Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!

/**************************/

```
f = open("demofile.txt", "r")
print(f.read())
```

Output >>

Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Output >>Hello

```
f = open("demofile.txt", "r")
print(f.readline())
```

Output >> Hello! Welcome to demofile.txt

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

Output >> Hello! Welcome to demofile.txt

This file is for testing purposes.

```
f = open("demofile.txt", "r")
for x in f:
    print(x):
```

Output >> Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!g purposes.

7.3- Python File Write

* To write to an existing file, you must add a parameter to the open() function:

* "a" - Append - will append to the end of the file.

* "w" - Write - will overwrite any existing content.

* Open the file "demo.txt" and append content to the file.

* Open the file "demo.txt" and overwrite the content.

* To create a new file in Python, use the open() method, with one of the following parameters:

* "x" - Create - will create a file, returns an error if the file exist

* "a" - Append - will create a file if the specified file does not exist

* "w" - Write - will create a file if the specified file does not exist

- Example (Write to an Existing File):

```
f = open("demofile.txt", "a")
f.write("Now the file has one more line!")
f = open("demofile.txt", "w")
f.write("Woops! I have deleted the content!")
f = open("myfile.txt", "x")
f = open("myfile.txt", "w")
```

7.4- Python Delete File

* To delete a file, you must import the OS module, and run its os.remove() function.
* To avoid getting an error, you might want to check if the file exist before you try to delete it.
* To delete an entire folder, use the os.rmdir() method.

- Example (Python Delete File):

```
import os
os.remove("demofile.txt")
import os
if os.path.exists("demofile.txt"):
    os.remove("demofile.txt")
else:
    print("The file does not exists")
import os
os.rmdir("myfolder")
```

8- working with number arrays

* Arrays are fundamental part of most programming languages. It is the collection of elements of a single data type, eg. array of int, array of string.
* However, in Python, there is no native array data structure. So, we use Python lists instead of an array.

* Note: If you want to create real arrays in Python, you need to use NumPy's array data structure. For mathematical problems, NumPy Array is more efficient.

* Unlike arrays, a single list can store elements of any data type and does everything an array does. We can store an integer, a float and a string inside the same list. So, it is more flexible to work with.

*** Python Array Methods:**

Methods	Functions
append()	to add element to the end of the list
extend()	to extend all elements of a list to the another list
insert()	to insert an element at the another index
remove()	to remove an element from the list
pop()	to remove elements return element at the given index
clear()	to remove all elements from the list
index()	to return the index of the first matched element
count()	to count of number of elements passed as an argument
sort()	to sort the elements in ascending order by default
reverse()	to reverse order element in a list
copy()	to return a copy of elements in a list

- Example (Create an Array):

```
arr = [10, 20, 30, 40, 50]
```

- Example (Accessing elements of array using indexing):

```
arr = [10, 20, 30, 40, 50]

print(arr[0])

print(arr[1])

print(arr[2])
```

Output >> **10**

20

30

- Example (Accessing elements of array using negative indexing):

```
arr = [10, 20, 30, 40, 50]
print(arr[-1])
print(arr[-2])
```

Output >> **50**

40

- Example (Find length of an array using len()):

```
brands = ["Coke", "Apple", "Google", "Microsoft", "Toyota"]
num_brands = len(brands)
print(num_brands)
```

Output >> **5**

- Example (Adding an element in an array using append()):

```
add = ['a', 'b', 'c']
add.append('d')
print(add)
```

Output >> **['a', 'b', 'c', 'd']**

- Example (Removing elements of an array using del, remove() and pop()):

```
colors = ["violet", "indigo", "blue", "green", "yellow", "orange", "red"]
del color[4]
colors.remove("blue")
colors.pop(3)
print(color)
```

Output >> ['violet', 'indigo', 'green', 'red']

- Example (Modifying elements of an array using Indexing):

```
fruits = ["Apple", "Banana", "Mango", "Grapes", "Orange"]
fruits[1] = "Pineapple"
fruits[-1] = "Guava"
print(fruits)
```

Output >> ['Apple', 'Pineapple', 'Mango', 'Grapes', 'Guava']

- Example (Concatenating two arrays using + operator):

```
concat = [1, 2, 3]
concat += [4,5,6]
print(concat)
```

Output >> [1, 2, 3, 4, 5, 6]

- Example (Repeating elements in array using * operator):

```
repeat = ["a"]
repeat = repeat * 5
print(repeat)
```

Output >> ['a', 'a', 'a', 'a', 'a']

- Example (Slicing an array using Indexing):

```
fruits = ["Apple", "Banana", "Mango", "Grapes", "Orange"]
print(fruits[1:4])
print(fruits[ : 3])
print(fruits[-4:])
print(fruits[-3:-1])
```

Output >> ['Banana', 'Mango', 'Grapes']

['Apple', 'Banana', 'Mango']

['Banana', 'Mango', 'Grapes', 'Orange']

['Mango', 'Grapes']

- Example (Create a two-dimensional array using lists):

```
multd = [[1,2], [3,4], [5,6], [7,8]]
print(multd[0])
print(multd[3])
print(multd[2][1])
print(multd[3][0])
```

Output >> **[1, 2]**

[7, 8]

6

7

9- Using pandas (library for manipulating time series) and matplotlib (to make plots in Python)

* Pandas stands for Python Data Analysis Library". According to the Wikipedia page on Pandas, "the name is derived from the term "panel data", an econometrics term for multidimensional structured data sets."

* Pandas is quite a game changer when it comes to analyzing data with Python and it is one of the most preferred and widely used tools.

* What's cool about Pandas is that it takes data (like a CSV or TSV file, or a SQL database) and creates a Python object with rows and columns called data frame that looks very similar to table in a statistical software (think Excel or SPSS for example. People who are familiar with R would see similarities to R too). This is so much easier to work with in comparison to working with lists and/or dictionaries through for loops or list.

9.1- Installation and Getting Started

9.1.1- In order to "get" Pandas you would need to install it. You would also need to have **Python 2.7** and above as a pre-requirement for installation.

9.1.2- It is also dependent on other libraries (like Numpy) and has optional dependancies (like Matplotlib for plotting).

9.1.3- Therefore, I think that the easiest way to get Pandas set up is to install it through a package like the **Anaconda distribution**.

9.1.4- In order to use Pandas in your Python IDE (Integrated Development Environment) like Jupyter Notebook or Spyder (both of them come with Anaconda by default), you need to import the Pandas library first.

9.1.5- Importing a library means loading it into the memory and then it's there for you to work with. In order to import Pandas all you have to do is run the following code:

- Example (Importing a library):

```
import pandas as pd

import numpy as np
```

9.1.6- Usually you would add the second part ('as pd') so you can access Pandas with 'pd.command' instead of needing to write 'pandas.command' every time you need to use it.

9.1.7- Also, you would import **numpy** as well, because it is very useful library for scientific computing with Python.

9.2- Working with Pandas

9.2.1- Loading and Saving Data with Pandas:

* When you want to use Pandas for data analysis, you'll usually use it in one of three different ways:

1- Convert a Python's list, dictionary or Numpy array to a Pandas data frame.

2- Open a local file using Pandas, usually a CSV file, but could also be a delimited text file (like TSV), Excel, etc.

3- Open a remote file or database like a CSV or a JSONon a website through a URL or read from a SQL table/database.

* There are different commands to each of these options, but when you open a file, they would look like this:

```
pd.read_filetype()
```

* There are different filetypes Pandas can work with, so you would replace "filetype" with the actual, well, filetype (like CSV).

* You would give the path, filename etc inside the parenthesis. Inside the parenthesis you can also pass different arguments that relate to how to open the file.

* There are numerous arguments and in order to know all you them, you would have to read the documentation (for example, the documentation for pd.read_csv() would contain all the arguments you can pass in this Pandas command).

* In order to convert a certain Python object (dictionary, lists etc) the basic command is:

pd.DataFrame()

* Inside the parenthesis you would specify the object(s) you're creating the data frame from. This command also has different arguments (clickable link).

* You can also save a data frame you're working with/on to different kinds of files (like CSV, Excel, JSON and SQL tables). The general code for that is:

df.to_filetype(filename)

9.2- Viewing and Inspecting Data

9.2.1- Now that you've loaded your data, it's time to take a look. How does the data frame look? Running the name of the data frame would give you the entire table, but you can also get the first n rows with df.head(n) or the last n rows with df.tail(n).

9.2.2- df.shape would give you the number of rows and columns. df.info() would give you the index, datatype and memory information.

9.2.3- The command s.value_counts(dropna=False) would allow you to view unique values and counts for a series (like a column or a few columns).

9.2.4- A very useful command is df.describe() which inputs summary statistics for numerical columns. It is also possible to get statistics on the entire data frame or a series (a column etc):

- Example (inputs summary statistics for numerical columns):

df.mean()Returns the mean of all columns

df.corr()Returns the correlation between columns in a data frame

df.count()Returns the number of non-null values in each data frame column

df.max()Returns the highest value in each column

df.min()Returns the lowest value in each column

df.median()Returns the median of each column

df.std()Returns the standard deviation of each column

9.3- Selection of Data

9.3.1- One of the things that is so much easier in Pandas is selecting the data you want in comparison to selecting a value from a list or a dictionary.

9.3.2- You can select a column (df[col]) and return column with label col as Series or a few columns (df[[col1, col2]]) and returns columns as a new DataFrame.

9.3.3- You can select by position (s.iloc[0]), or by index (s.loc['index_one']) .

9.3.4- In order to select the first row you can use df.iloc[0,:] and in order to select the first element of the first column you would run df.iloc[0,0].

9.3.5- These can also be used in different combinations, so I hope it gives you an idea of the different selection and indexing you can perform in Pandas.

9.4- Filter, Sort and Groupby

9.4.1- You can use different conditions to filter columns. For example, df[df[year] > 1984] would give you only the column year is greater than 1984. You can use & (and) or | (or) to add different conditions to your filtering. These is also called boolean filtering.

9.4.2- It is possible to sort values in a certain column in an ascending order using df.sort_values(col1) ; and also in a descending order using df.sort_values(col2,ascending=False).

9.4.3- Furthermore, it's possible to sort values by col1 in ascending order then col2 in descending order by using df.sort_values([col1,col2],ascending=[True,False]).

9.4.4- The last command in this section is groupby. It involves splitting the data into groups based on some criteria, applying a function to each group independently and combining the results into a data structure.

9.4.5- df.groupby(col) returns a groupby object for values from one column while df.groupby([col1,col2]) returns a groupby object for values from multiple columns.

9.5- Data Cleaning

9.5.1- Data cleaning is a very important step in data analysis.

9.5.2- For example, we always check for missing values in the data by running pd.isnull() which checks for null Values, and returns a boolean array (an array of true for missing values and false for non-missing values).

9.5.3- In order to get a sum of null/missing values, run pd.isnull().sum(). pd.notnull() is the opposite of pd.isnull().

9.5.4- After you get a list of missing values you can get rid of them, or drop them by using df.dropna() to drop the rows or df.dropna(axis=1) to drop the columns.

9.5.5- A different approach would be to fill the missing values with other values by using df.fillna(x) which fills the missing values with x (you can put there whatever you want) or s.fillna(s.mean()) to replace all null values with the mean (mean can be replaced with almost any function from the statistics section).

9.5.6- It is sometimes necessary to replace values with different values.

9.5.7- For example, s.replace(1,'one') would replace all values equal to 1 with 'one'. It's possible to do it for multiple values: s.replace([1,3],['one','three']) would replace all 1 with 'one' and 3 with 'three'.

9.5.8- You can also rename specific columns by running: df.rename(columns={'old_name': 'new_ name'}) or use df.set_index('column_one') to change the index of the data frame.

9.6- Join/Combine

9.6.1- The last set of basic Pandas commands are for joining or combining data frames or rows/columns.

9.6.2- The three commands are:

* df1.append(df2)— add the rows in df1 to the end of df2 (columns should be identical)

* df.concat([df1, df2],axis=1)—add the columns in df1 to the end of df2 (rows should be identical)

* df1.join(df2,on=col1,how='inner')—SQL-style join the columns in df1 with the columns on df2 where the rows for colhave identical values. how can be equal to one of: 'left', 'right', 'outer', 'inner'

10- Debugging (Reading error messages and learning how to debug)

* As a programmer, one of the first things that you need for serious program development is a debugger.

* Python has a debugger, which is available as a module called pdb (for "Python DeBugger", naturally!).

* Unfortunately, most discussions of pdb are not very useful to a Python newbie most are very terse and simply rehash the description of pdb in the Python library reference manual.

* It assumes that you are not using any IDE that you're coding Python with a text editor and running your Python programs from the command line.

* Getting started — pdb.set_trace()

* To start, I'll show you the very simplest way to use the Python debugger:

- Example (use the Python debugger):

1. Let's start with a simple program, epdb1.py.

```
# epdb1.py -- experiment with the Python debugger, pdb

a = "aaa"

b = "bbb"

c = "ccc"

final = a + b + c

print final
```

2. Insert the following statement at the beginning of your Python program. This statement imports the Python debugger module, pdb.

```
import pdb
```

3. Now find a spot where you would like tracing to begin, and insert the following code:

```
pdb.set_trace()
```

So now your program looks like this:

- Example (Python debugger):

```
# epdb1.py -- experiment with the Python debugger, pdb

import pdb

a = "aaa"

pdb.set_trace()

b = "bbb"

c = "ccc"

final = a + b + c

print final
```

4. Now run your program from the command line as you usually do, which will probably look something like this:

PROMPT> python epdb1.py

* When your program encounters the line with pdb.set_trace() it will start tracing.

* That is, it will (1) stop, (2) display the "current statement" (that is, the line that will execute next) and (3) wait for your input.

* You will see the pdb prompt, which looks like this:

(Pdb)

* Execute the next statement… with "n" (next)

* At the (Pdb) prompt, press the lower-case letter "n" (for "next") on your keyboard, and then press the **ENTER key**. This will tell pdb to execute the current statement. Keep doing this pressing "n", then **ENTER**.

* Eventually you will come to the end of your program, and it will terminate and return you to the normal command prompt.

* Congratulations! You've just done your first debugging run!

* Repeating the last debugging command… with **ENTER**

* This time, do the same thing as you did before. Start your program running. At the (Pdb) prompt, press the lower-case letter "**n**" (for "next") on your keyboard, and then press the **ENTER key**.

* But this time, after the first time that you press "n" and then **ENTER**, don't do it any more. Instead, when you see the (Pdb) prompt, just press **ENTER**. You will notice that pdb continues, just as if you had pressed "**n**". So this is Handy Tip #1:

* If you press **ENTER** without entering anything, pdb will re-execute the last command that you gave it.

* In this case, the command was "**n**", so you could just keep stepping through the program by pressing **ENTER**.

* Notice that as you passed the last line (the line with the "print" statement), it was executed and you saw the output of the print statement ("aaabbbccc") displayed on your screen.

* Quitting it all… with "**q**" (**quit**)

* The debugger can do all sorts of things, some of which you may find totally mystifying. So the most important thing to learn now — before you learn anything else — is how to quit debugging!

* It is easy. When you see the (Pdb) prompt, just press "**q**" (for "quit") and the **ENTER key**. Pdb will quit and you will be back at your command prompt. Try it, and see how it works.

* Printing the value of variables… with "**p**" (**print**)

* The most useful thing you can do at the (Pdb) prompt is to print the value of a variable. Here's how to do it.

* When you see the (Pdb) prompt, enter "**p**" (for "print") followed by the name of the variable you want to print. And of course, you end by pressing the **ENTER key**.

* Note that you can print multiple variables, by separating their names with commas (just as in a regular Python "print" statement). For example, you can print the value of the variables a, b, and c this way:

p a, b, c

* When does pdb display a line?

* Suppose you have progressed through the program until you see the line

final = a + b + c

and you give pdb the command

p final

* You will get a NameError exception. This is because, although you are seeing the line, it has not yet executed. So the final variable has not yet been created.

* Now press "**n**" and **ENTER** to continue and execute the line. Then try the "p final" command again. This time, when you give the command "p final", pdb will print the value of final, which is "aaabbbccc".

* Turning off the (Pdb) prompt… with "**c**" (continue)

* You probably noticed that the "**q**" command got you out of pdb in a very crude way — basically, by crashing the program.

* If you wish simply to stop debugging, but to let the program continue running, then you want to use the "**c**" (for "continue") command at the (Pdb) prompt.

* This will cause your program to continue running normally, without pausing for debugging. It may run to completion. Or, if the pdb.set_trace() statement was inside a loop, you may encounter it again, and the (Pdb) debugging prompt will appear once more.

* Seeing where you are… with "l" (list)

* As you are debugging, there is a lot of stuff being written to the screen, and it gets really hard to get a feeling for where you are in your program. That's where the "**l**" (for "list") command comes in. (Note that it is a lower-case "**L**", not the numeral "one" or the capital letter "I".)

* "**l**" shows you, on the screen, the general area of your program's souce code that you are executing. By default, it lists 11 (eleven) lines of code. The line of code that you are about to execute (the "current line") is right in the middle, and there is a little arrow "–>" that points to it.

* So a typical interaction with pdb might go like this

* The pdb.set_trace() statement is encountered, and you start tracing with the (Pdb) prompt

* You press "**n**" and then **ENTER**, to start stepping through your code.

* You just press **ENTER** to step again.

* You just press **ENTER** to step again.

* You just press **ENTER** to step again. etc. etc. etc.

* Eventually, you realize that you are a bit lost. You're not exactly sure where you are in your program any more. So…

* You press "**l**" and then **ENTER**. This lists the area of your program that is currently being executed.

* You inspect the display, get your bearings, and are ready to start again. So….

* You press "**n**" and then **ENTER**, to start stepping through your code.

* You just press **ENTER** to step again.

* You just press **ENTER** to step again. etc. etc. etc.

* Stepping into subroutines… with "**s**" (step into)

* Eventually, you will need to debug larger programs — programs that use subroutines. And sometimes, the problem that you're trying to find will lie buried in a subroutine.

- Example (debug larger programs):

```
# epdb2.py -- experiment with the Python debugger, pdb

import pdb

def combine(s1,s2):      # define subroutine combine, which...

    s3 = s1 + s2 + s1    # sandwiches s2 between copies of s1, ...
```

```
    s3 = "" + s3 +""   # encloses it in double quotes,...
    return s3          # and returns it.
a = "aaa"
pdb.set_trace()
b = "bbb"
c = "ccc"
final = combine(a,b)
print final
```

* As you move through your programs by using the "**n**" command at the (Pdb) prompt, you will find that when you encounter a statement that invokes a subroutine the final = combine(a,b) statement, for example pdb treats it no differently than any other statement. That is, the statement is executed and you move on to the next statement in this case, to print final.

* But suppose you suspect that there is a problem in a subroutine. In our case, suppose you suspect that there is a problem in the combine subroutine. What you want when you encounter the final = combine(a,b) statement is some way to step into the combine subroutine, and to continue your debugging inside it.

* Well, you can do that too. Do it with the "s" (for "step into") command.

* When you execute statements that do not involve function calls, "**n**" and "**s**" do the same thing — move on to the next statement. But when you execute statements that invoke functions, "s", unlike "**n**", will step into the subroutine.

```
final = combine(a,b)
```

* statement using "**s**", then the next statement that pdb would show you would be the first statement in the combine subroutine:

```
def combine(s1,s2):
```

* and you will continue debugging from there.

* Continuing… but just to the end of the current subroutine… with "**r**" (return)

* When you use "**s**" to step into subroutines, you will often find yourself trapped in a subroutine. You have examined the code that you're interested in, but now you have to step through a lot of uninteresting code in the subroutine.

* In this situation, what you'd like to be able to do is just to skip ahead to the end of the subroutine. That is, you want to do something like the "**c**" ("continue") command does, but you want just to continue to the end of the subroutine, and then resume your stepping through the code.

* You can do it. The command to do it is "**r**" (for "return" or, better, "continue until return"). If you are in a subroutine and you enter the "**r**" command at the (Pdb) prompt, pdb will continue executing until the end of the subroutine. At that point the point when it is ready to return to the calling routine it will stop and show the (Pdb) prompt again, and you can resume stepping through your code.

* You can do anything at all at the (Pdb) prompt …

* Sometimes you will be in the following situation — You think you've discovered the problem. The statement that was assigning a value of, say, "aaa" to variable var1 was wrong, and was causing your program to blow up. It should have been assigning the value "bbb" to var1.

… at least, you're pretty sure that was the problem…

* What you'd really like to be able to do, now that you've located the problem, is to assign "bbb" to var1, and see if your program now runs to completion without bombing.

It can be done!

* One of the nice things about the (Pdb) prompt is that you can do anything at it you can enter any command that you like at the (Pdb) prompt. So you can, for instance, enter this command at the (Pdb) prompt.

(Pdb) var1 = "bbb"

* You can then continue to step through the program. Or you could be adventurous use "**c**" to turn off debugging, and see if your program will end without bombing!

… but be a little careful!

* Since you can do anything at all at the (Pdb) prompt, you might decide to try setting the variable b to a new value, say "BBB", this way:

(Pdb) b = "BBB"

* If you do, pdb produces a strange error message about being unable to find an object named '= "BBB" '. Why???

* What happens is that pdb attempts to execute the pdb b command for setting and listing breakpoints. It interprets the rest of the line as an argument to the b command, and can't find the object that (it thinks) is being referred to. So it produces an error message.

* So how can we assign a new value to b? The trick is to start the command with an exclamation point (!).

(Pdb)!b = "BBB"

* An exclamation point tells pdb that what follows is a Python statement, not a pdb command.

11- working with Numpy

* Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy.

11.1- Arrays

* A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

- Example (initialize numpy arrays from nested Python lists):

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array

print(type(a))           # Prints "<class 'numpy.ndarray'>"

print(a.shape)           # Prints "(3,)"

print(a[0], a[1], a[2])   # Prints "1 2 3"

a[0] = 5                 # Change an element of the array

print(a)                 # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array

print(b.shape)                     # Prints "(2, 3)"

print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

- Example (create arrays):

```
import numpy as np
a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #         [ 0.  0.]]"
b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"
c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #         [ 7.  7.]]"
d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #         [ 0.  1.]]"
e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #              [ 0.68744134  0.87236687]]"
```

- Example (Slicing):

```
import numpy as np
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
```

```
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

- Example (mix integer indexing with slice indexing):

```
import numpy as np
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
```

```
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #         [ 6]
                             #         [10]] (3, 1)"
```

- Example (Integer array indexing):

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"
# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"
# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

- Example (with integer array indexing is selecting or mutating one element from each row of a matrix):

```
import numpy as np
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
```

```
#              [ 7,  8,  9],
#              [10, 11, 12]])"
# Create an array of indices
b = np.array([0, 2, 0, 1])
# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10
print(a)  # prints "array([[11,  2,  3],
#              [ 4,  5, 16],
#              [17,  8,  9],
#              [10, 21, 12]])
```

- Example (Boolean array indexing):

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.
print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
```

```
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:

print(a[a > 2])     # Prints "[3 4 5 6]"
```

- Example (Datatypes):

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype

print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype

print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype

print(x.dtype)                         # Prints "int64"
```

- Example (Array math):

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)

y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array

# [[ 6.0  8.0]

#  [10.0 12.0]]

print(x + y)

print(np.add(x, y))

# Elementwise difference; both produce the array

# [[-4.0 -4.0]

#  [-4.0 -4.0]]

print(x - y)

print(np.subtract(x, y))
```

```
# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

- Example (We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices):

```
import numpy as np
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
v = np.array([9,10])
w = np.array([11, 12])
# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
# Matrix / vector product; both produce the rank 1 array [29 67]
```

```
print(x.dot(v))
print(np.dot(x, v))
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

- Example (sum)

```
import numpy as np
x = np.array([[1,2],[3,4]])
print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

- Example (transposing a matrix)

```
# to transpose a matrix, simply use the T attribute of an array object
import numpy as np
x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #         [3 4]]"
print(x.T)  # Prints "[[1 3]
            #         [2 4]]"
# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

- Example (Broadcasting)

* Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

```
import numpy as np

# We will add the vector v to each row of the matrix x,

# storing the result in the matrix y

x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

v = np.array([1, 0, 1])

y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop

for i in range(4):

    y[i, :] = x[i, :] + v

# Now y is the following

# [[ 2  2  4]

#  [ 5  5  7]

#  [ 8  8 10]

#  [11 11 13]]

print(y)
```

- Example (matrix is very large and computing an explicit loop in Python could be slow)

* This works; however when the matrix x is very large, computing an explicit loop in Python could be slow. Note that adding the vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically, then performing elementwise summation of x and vv. We could implement this approach like this.

```
import numpy as np

# We will add the vector v to each row of the matrix x,

# storing the result in the matrix y
```

```
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

- Example (perform this computation without actually creating multiple copies of v)

```
import numpy as np
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

- Example (Broadcasting applications)

```
import numpy as np

# Compute outer product of vectors

v = np.array([1,2,3])  # v has shape (3,)

w = np.array([4,5])    # w has shape (2,)

# To compute an outer product, we first reshape v to be a column

# vector of shape (3, 1); we can then broadcast it against w to yield

# an output of shape (3, 2), which is the outer product of v and w:

# [[ 4  5]

#  [ 8 10]

#  [12 15]]

print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix

x = np.array([[1,2,3], [4,5,6]])

# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),

# giving the following matrix:

# [[2 4 6]

#  [5 7 9]]

print(x + v)

# Add a vector to each column of a matrix

# x has shape (2, 3) and w has shape (2,).

# If we transpose x then it has shape (3, 2) and can be broadcast

# against w to yield a result of shape (3, 2); transposing this result

# yields the final result of shape (2, 3) which is the matrix x with

# the vector w added to each column. Gives the following matrix:
```

```
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))
# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

12- URL

https://www.digitalocean.com/community/tutorials/how-to-use-the-python-debuggerhttps://www.python.org/dev/peps/

http://pyvideo.org/

https://www.python.org/downloads/

https://www.guru99.com/python-tutorials.html

https://www.jetbrains.com/pycharm/

https://www.jetbrains.com/pycharm/download/download-thanks.html?platform=windows

https://eric-ide.python-projects.org/

https://github.com/codingforentrepreneurs/30-Days-of-Python

http://slproweb.com/products/Win32OpenSSL.html

https://www.w3schools.com/python/python_intro.asp

http://www.pythonforbeginners.com/loops/for-while-and-nested-loops-in-python

https://pythonconquerstheuniverse.wordpress.com/2009/09/10/debugging-in-python/

https://docs.python.org/2/library/pdb.html

https://www.python.org/download/releases/2.7/

https://www.python.org/doc/

https://docs.python.org/release/2.7/

https://docs.python.org/release/2.7/library/index.html

https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.array.html

https://docs.scipy.org/doc/numpy-1.13.0/user/index.html

https://www.tutorialspoint.com/python_pandas/python_pandas_panel.htm

https://towardsdatascience.com/a-quick-introduction-to-the-pandas-python-library-f1b678f34673

https://www.tutorialspoint.com/python_pandas/python_pandas_window_functions.htm

https://pandas.pydata.org/pandas-docs/stable/tutorials.html

https://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs

https://jakevdp.github.io/PythonDataScienceHandbook/02.02-the-basics-of-numpy-arrays.html

https://www.anaconda.com/download/

https://docs.anaconda.com/anaconda/user-guide/getting-started