LAHORE UNIVERSITY OF MANAGEMENT SCIENCES
Syed Babar Ali School of Science and Engineering

EE212 Mathematical Foundations for Machine Learning and Data Science
Fall Semester 2021

Laboratory 0 – Introduction to Python

Issued: Tuesday 7 September, 2021

Total Marks: 0

Contribution to Final Assessment: 0%

Submission: None.

Goal

The goal of this laboratory is to set up python installation and perform some basic operations in Python.

Python Installation

Setting up Anaconda

Anaconda is an open-source distribution of Python and R programming languages for scientific computing. Installing Anaconda will automatically install Python and the necessary packages for development and analysis. Click here to proceed to the official Anaconda page. Download the installer compatible with your system to install a fresh copy of Anaconda. Run the installer and follow the instructions there. Additionally, the following two resources present detailed instructions for installation:

- Anaconda for Windows
- Anaconda for Mac

Verifying Installation

You may use any text editor to write Python code. One such text editor is available here. However, be sure to save your code file with .py extension. For instance, your filename should read filename.py for it to be recognized as python file. To verify your installation, we will print the version of a few packages in the console. Write the following code in a file and save it:

```
import scipy
import numpy

print("scipy: %s" % scipy.__version__)
print("numpy: %s" % numpy.__version__)
```

Execute the code by running the following commands in the Terminal (Mac users) or Conda Prompt (Windows users). Make sure to first change the directory in which you are.

For Windows, change directory by typing in (replace "directory-path" with your own directory path where you saved the file):

```
cd "directory-path"
```

Run the file in that directory by typing in the command:

```
python filename.py
```

Note: the above command assumes you named your file as filename.py

Checking Libraries

Before checking the libraries, we will also install one more library. Run the following command in your terminal:

```
pip install python-mnist
```

There should be a confirmation message telling you that the library is installed successfully.

Now, make a Python file with your text editor and run the following code to check if the relevant libraries are installed on your machine:

```
import numpy as np
import scipy as sp
from scipy.linalg import null_space
from numpy.linalg import inv
from matplotlib.image import imread
import matplotlib.pyplot as plt
import pandas as pd
import math
from scipy import stats
import random
from numpy.random import seed
from numpy.random import rand
from numpy.random import randn
from scipy.stats import pearsonr
from scipy.stats import ttest_ind
from scipy.stats import mannwhitneyu
from mnist import MNIST
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
from sklearn import metrics
from skimage.feature import hog
from sklearn.decomposition import PCA
from scipy.io import wavfile
from scipy.io.wavfile import write
import wave
import sys
```

Let the TA know if any if there is an error in running this code or if any of the libraries is not installed.

1 Task - String Operations

We can make use of quotations marks " " to define strings. For example:

```
"Michael Jackson"
```

And these strings could be assigned to a variable. For instance:

```
mystring = "Michael Jackson"
```

It's important to understand indexing in Python. Importantly, it starts with index 0. So in order to access the first element 'M' of the string"Michael Jackson", you would use:

```
mystring[0]
```

And to visualize it on the console, you could print it:

```
print(mystring[0])
```

Python also supports negative indexing. The last element of the string could be accessed as:

```
print(mystring[-1])
```

and the first element as:

```
print(mystring[-15])
```

You could also slice through the variable mystring to access only some alphabets in the string. For example, if you were interested in only the first four alphabets, you could access them by getting values from index 0 to index 3:

```
print(mystring[0:4])
```

Or if you are interested in the slice of variable mystring from index 8 to index 11:

```
print(mystring[8:12])
```

Python also facilitates string concatenation. You could simply concatenate strings in the following way:

```
mystring = "Michael Jackson"
mystring = mystring + " is the best"
print(mystring)
```

Back slashes represent the beginning of escape sequences. Escape sequences represent strings that may be difficult to input. For example, back slash"n" represents a new line while back slash "t" represents a tab sequence:

```
print("Michael Jackson \n is the best" )
print("Michael Jackson \t is the best" )
```

There are many string operation methods in Python that can be used to manipulate the data. The method "upper" converts lower case characters to upper case characters:

```
A = "Thriller is the sixth studio album"
print("before upper:", A)
B = A.upper()
print("After upper:", B)
```

The method "replace" replaces a segment of the string with the second argument provided:

```
A = "Michael Jackson is the best"
print(A)
B = A.replace('Michael', 'Janet')
print(B)
```

The find method returns the index of the sub-string you intend to find in the main string. This is also very useful when finding a particular element in an array:

```
mystring = "Michael Jackson"
print(mystring.find('el'))
```

---

2 Task - List and Tuples

List is another kind of data structure and has it's own functions/methods. For those of you who are familiar with C++ and MATLAB, a list is much like an array with some added flexibility. To define a list we use square brackets [ ], with items separated by commas. For example:

```
listA=[1,2,3,4,5,3]
```

The flexibility of lists is that the items can be numbers, strings, other lists or even a mixture of all of these! The following are valid lists, try them out and print them to verify:

```
listB=["This","is","a","list","of","strings"]
list_mixed=["apples","oranges",15,"grapes",3.4,[1, "Hello World"]]
```

The indexing is the same as it was for strings, where you can access a particular index, a range of indices, or even use negative indexing as shown in the previous section.

You can add elements to a list in a number of ways. One way is to use the function .extend(). You write the list that you want to use to extend the present list. For example:

```
listA=[1,2,3,4,5]
listA.extend([1,7])
```

If you print listA on the console, you will observe:

```
[1,2,3,4,5,1,7]
```

Another way to add elements is to use the function .append(). In this you write the element that you want to append to the list. For example:

```
listA=[1,2,3,4,5]
listA.append(22)
```

If you print listA on the console, you will observe:

```
[1,2,3,4,5,22]
```

You can also use a simple "+" operation to concatenate lists, as you did for strings before.

Notice how these functions add elements at the end of the list. To add elements at any index of your choice, you can use the function .insert(). The arguments of the function are the index at which you want to insert the element, and the element itself. For example if you want to add the element "f" at index 2, you do the following operation:

```
listC=["a","b","c","d"]
listC.insert(2,"f")
```

The result on the console would be:

```
["a","b","f","c","d"]
```

To delete a particular element in a list at a particular index, you can use the "del" function. For example, to delete an element at index 4, you would write:

```
del listC[4]
```

There are many other functions in a list that you can explore on your own. Such as .count("item"), which returns tells us how many "item" are in the list. The .clear() function clears the entire list and the .reverse() function reverses the order. Do explore these functions on your own.

A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list. Just like lists, you can have empty tuples, a tuple of integers, strings or even a nested tuple (with different data types). You use parenthesis ( ) to create a tuple. Following are examples of valid tuples:

```
tupleA=()
tupleB=(1,"text",5)
tupleC=(1,["a","list"],("tuple",3.4))
```

A small caveat is that if you want to make a tuple containing a single element, you can not simply create it like we do normally, instead, there needs to be a comma after you have written the element, for example:

```
tuple_single=("single",)
```

If we do not put that tailing comma at the end, it would just be declared as a string type. Indexing in a tuple is the same as for strings and lists, you use the square brackets [ ] to access an element. If the element you are accessing is a list or a tuple itself, then you can further access it by using the [ ] consecutively. An example of accessing elements is:

```
tupleC=(1,["this","is","a","list"],("tuple",3.4))
print(tupleC[0])
print(tupleC[1][2])
```

The first line returns 1, while the second line returns the element at index 2 of the element at index 1. Hence it returns "a". Negative indexing and slicing is also supported by tuples, just like strings and lists. We can use concatenation in the same manner as we did for strings and lists, using the "+" operator.

Unlike lists, tuples are immutable. This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like list, its nested items can be changed. For example:

```
tupleD=(1,15,4,[5,10])    # Declare it with some values
tupleD[2]=4               # change value from 4 to 1 (illegal)
tupleD[3][0]=4            # Change value in the list
```

The second line is not valid, as you can not change values of a tuple, however you can change values in a list in a tuple, like we did in the third line above. Since we can not change a value in a tuple, hence we can not delete a particular value in a tuple like we did for lists. We can only delete the entire tuple in the following way:

```
del tupleD
```

Tuple has it's own methods, just like lists. To count the occurrences of a particular item, or to find it's index, you can use the following functions:

```
tupleE=("a","p","p","l","e")
print(tupleE.count("p"))
print(tupleE.index("l"))
```

The first output is 2 while the second output is 3. Do try it on your own. Another useful method that is unique to a tuple, is the one that allows you to check if a certain element is present in a tuple. You can check it in the following manner:

```
print("a" in tupleE)
print("b" in tupleE)
```

The first one will return "True" while the second one will return "False".

---

3 Task - Loops

After taking any introductory course in programming, you should all be familiar with the concepts of loops, mainly "for loops" and "while loops". Loops in Python are no different, except for a few changes here in there in the syntax and some subtleties.

The for loop, as you know, is for iterating over a sequence. That sequence can be manually defined or defined by a data structure such as a list or a tuple. To display all items in a list, we would use the following for loop (pay close attention to the syntax):

```
listA=["a","b","c","d","e"]
for item in listA:
    print(item)
```

Notice how there is no keyword for ending a for loop. In python, indenting matters **a lot**. After the ":" in the for loop, all the lines below, with the indenting as shown, will be considered as part of the loop. If you write the following code, with indenting aligned with the "for" in the statement and not one step further, then that code is considered to be written outside the loop.

If you want to traverse the list manually, or just set a loop to run a fixed number of times, you can use the following syntax:

```
for i in range(1,20):
    print(i*i)
```

This will print the square of all values from 1 till 19, 20 is NOT included, so be very careful while using the range() for defining the sequence and make sure your edge points are included. We can also use the range function to iterate through a list, coupled with the len() function:

```
genre = ["pop", "rock", "jazz"]
for i in range(len(genre)):
    print("I like", genre[i])
```

The len() function will give the length of the list and the for loop will hence iterate over the complete list. This code will give the following output:

```
I like pop
I like rock
I like jazz
```

Keywords like break can be used in the same fashion in a for loop as they are used in C++ or MATLAB.

While loop is very similar to a for loop, except that in a while loop, the iteration is not over a sequence. It can be broken by a stopping condition of the loop or within it's body, otherwise it keeps on looping endlessly. You also have to manually initialize and update the loop variable. An example of such loop is:

```
n = 10
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i+1    # update counter

print("The sum is", sum)
```

This loops goes on until $i == n$, and then break when that condition is false. If you update the variable as $i = i - 1$, then notice that "i" will always be less that "n" and hence the loop goes on endlessly, thus you have to be extra careful with while loops.

---

4 Task - Functions

A function is a reusable block of code which performs operations specified in the function. It lets you break down tasks and allows you to reuse your code in different programs. Functions blocks begin with 'def' followed by the function 'name' and parentheses (). There is a body within every function that starts with a colon ':' and is indented. The statement 'return' exits a function, optionally passing back a value. If you do not need

to return anything, you may use the keyword 'None' as the return value. Example of an Add function is shown below:

```
def add(a):
    b = a + 1
    print(a, "if you add one", b)
    return(b)
```

We can simply call the function by executing:

```
add(1)
```

Similarly, a multiply function could be defined as:

```
def Mult(a, b):
    c = a * b
    return(c)
```

And it could be called by executing:

```
Mult(2,3)
```

There are many pre-defined functions in Python. The print() function that we used earlier is one such example. sum() and len() are some other examples.

```
album_ratings = [10.0, 8.5, 9.5, 7.0, 7.0, 9.5, 9.0, 9.5]
print(album_ratings)
sum(album_ratings)
len(album_ratings)
```

---

5 Task - Numpy

A numpy array is similar to a list. It's usually fixed in size and each element is of the same type. We can cast a list to a numpy array by importing numpy. The example below defines a numpy array and prints the values stored in it as well as explores the data type of the array by printing it to the console.

```
import numpy as np

a = np.array([0, 1, 2, 3, 4])
print("a[0]:", a[0])
print("a[1]:", a[1])
print("a[2]:", a[2])
print("a[3]:", a[3])
print("a[4]:", a[4])
print(type(a))

b = np.array([3.1, 11.02, 6.2, 213.2, 5.2])
print(type(b))
```

Below is a demonstration to understand some of the key attributes of a numpy array. You may print these values in the console to better understand them.

```
a = np.array([0, 1, 2, 3, 4])
a.size              # size of the array
a.ndim              # dimension of the array
a.shape             # shape of the array
a.mean()            # mean of the array
a.max()             # maximum value in the array
a.min()             # minimum value in the array
```

Furthermore, numpy arrays are simple to use when taking products. Consider the following illustration:

```
u = np.array([1, 2])
v = np.array([3, 2])
z = u * v           # product of two arrays
np.dot(u, v)        # dot product of two arrays
```

These attributes of numpy arrays are well appreciated when working with matrices or arrays with higher dimensions.