Lab 2-A

Module:

A module is a python file that (generally) has only definitions of variables, functions, and classes.

Example: Module name mymodule.py

```
# Define some variables:
ageofqueen = 78

# define some functions
def printhello():
    print ("hello")
# define a class
class Piano:
    def_init__(self):
        self.type = input("What type of piano?: ")
        self.height = input("What height (in feet)?: ")
        self.price = input("How much did it cost?: ")
        self.age = input("How old is it (in years)?: ")

    def printdetails(self):
        print ("This piano is a/an " + self.height + " foot")
        print (self.type, "piano, " + self.age, "years old and costing " +
self.price + " dollars.")
```

Importing module in main program:

```
### mainprogam.py ##
# IMPORTS ANOTHER MODULE

import mymodule
print (mymodule.ageofqueen )
cfcpiano = mymodule.Piano()
cfcpiano.printdetails()
```

Another way of importing the module is:

```
from mymodule import Piano, ageofqueen
print (ageofqueen)
cfcpiano = Piano()
cfcpiano.printdetails()
```

Lab Journal 2-A:

1. Create a class name basic_calc with following attributes and methods;
   Two integers (values are passed with instance creation)
   Different methods such as addition, subtraction, division, multiplication
   Create another class inherited from basic_calc named s_calc which should have the following additional methods;
   Factorial, x_power_y,log, ln etc

2. Modify the classes created in the above task under as follows:
   Create a module name basic.py having the class name basic_calc with all the attributes and methods defined before.
   Now import the basic.py module in your program and do the inheritance step defined before i.e.
   Create another class inherited from basic_calc named s_calc which should have the following additional methods;
   Factorial, x_power_y, log, ln etc

Lab 2-B

Lists:

Lists are what they seem - a list of values. Each one of them is numbered, starting from zero. You can remove values from the list, and add new values to the end. Example: Your many cats' names. *Compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Listitems need not all have the same type.

```
cats = ['Tom', 'Snappy', 'Kitty', 'Jessie', 'Chester']

print (cats[2])
cats.append("Oscar")
#Remove 2nd cat, Snappy.
del cats[1]
```

Compound datatype:

```
>>> a = ['spam', 'eggs', 100, 1234]
#slicing examples
>>> a[1:-1]    #start at element at index 1, end before last element
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']

>>> a= ['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
```

```
['spam', 'eggs', 123, 1234]
```

Replace some items:

```
>>> a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
```

Remove some:

```
>>> a[0:2] = []
>>> a
[123, 1234]
```

Clear the list: replace all items with an empty list:

```
>>> a[:] = []
>>> a
[]
```

Length of list:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

Nested lists:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
```

Functions of lists:

list.append(x): Add an item to the end of the list; equivalent to a[len(a):] = [‘x’].

list.extend(L): Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L.

list.insert(i, x): Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

list.remove(x): Remove the first item from the list whose value is x. It is an error if there is no such item.

list.pop(i): 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.

list.count(x): Return the number of times x appears in the list.

list.sort(): Sort the items of the list, in place.

list.reverse(): Reverse the elements of the list, in place.

Tuples:

Tuples are just like lists, but you can't change their values. Again, each value is numbered starting from zero, for easy reference. Example: the names of the months of the year.

months = ('January' , 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December')

Index	Value
0	January
1	February
2	March
3	April
4	May
5	June
6	July
7	August
8	September
9	October
10	November
11	December

We can have easy membership tests in Tuples using the keyword in.

```
>>> 'December' in months   # fast membership testing

True
```

Sets:

A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the set() function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary.

```
Example 1:

>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

>>> fruit = set(basket)  # create a set without duplicates

>>> fruit

{'banana', 'orange', 'pear', 'apple' }

>>> 'orange' in fruit   # fast membership testing

True

>>> 'crabgrass' in fruit
```

```
False

Example 2:

>>> # Demonstrate set operations on unique letters from two words

>>> a = set('abracadabra')

>>> b = set('alacazam')

>>> a                                  # unique letters in a

{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b

{'r', 'd', 'b'}

>>> a | b                              # letters in either a or b

{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

>>> a & b                              # letters in both a and b

{'a', 'c'}

>>> a ^ b                              # letters in a or b but not both

{'r', 'd', 'b', 'm', 'z', 'l'}
```

Set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}

>>> a

{'r', 'd'}
```

Dictionaries:

Dictionaries are similar to what their name suggests - a dictionary. In a dictionary, you have an 'index' of words, and for each of them a definition.

In python, the word is called a 'key', and the definition a 'value'. The values in a dictionary aren't numbered - they aren't in any specific order, either - the key does the same thing.

You can add, remove, and modify the values in dictionaries. Example: telephone book.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing list(d.keys()) on a dictionary returns a list of all the keys used in the dictionary, in

arbitrary order (if you want it sorted, just use sorted(d.keys()) instead). To check whether a single key is in the dictionary, use the in keyword.

At one time, only one value may be stored against a particular key. Storing a new value for an existing key overwrites its old value. If you need to store more than one value for a particular key, it can be done by storing a list as the value for a key.

```
phonebook = {'ali':8806336, 'omer':6784346,'shoaib':7658344, 'saad':1122345 }
#Add the person '' to the phonebook:
phonebook['usama'] = 1234567
print("Original Phonebook")
print(phonebook)
# Remove the person 'shoaib' from the phonebook:
del phonebook['shoaib']

print("'shoaib' deleted from phonebook")
print(phonebook)

phonebook = {'Andrew Parson':8806336,'Emily Everett':6784346, 'Peter Power':7658344,
'Louis Lane':1122345}

print("New phonebook")
print(phonebook)

#Add the person 'Gingerbread Man' to the phonebook:
phonebook['Gingerbread Man'] = 1234567

list(phonebook.keys())

sorted(phonebook.keys())

print( 'waqas' in phonebook)
print( 'Emily Everett' in phonebook)

#Delete the person 'Gingerbread Man' from the phonebook:
del phonebook['Gingerbread Man']
```

Lab Journal 2-B:

1. Create list of Fibonacci numbers after calculating Fibonacci series up to the number n which you will pass to a function as an argument. The number n must be input by the user. They are calculated using the following formula: The first two numbers of the series is always equal to 1, and each consecutive number returned is the sum of the last two numbers. Hint: Can you use only two variables in the generator function?
   ```
   a = 1
   b = 2
   a, b = b, a
   ```
   will simultaneously switch the values of a and b.
   The first number in the series should be 1. (The output will start like 1,1,2,3,5,8,…)

2. Write a program that lets the user enter some English text, then converts the text to Pig-Latin. To review, Pig-Latin takes the first letter of a word, puts it at the end, and appends –ay|. The only exception is if the first letter is a vowel, in which case we keep it as it is and append –hay| to the end. For example: –hello| -> –ellohay|, and –image| >

–imagehay|

It will be useful to define a list or tuple at the top called VOWELS. This way, you can check if a letter *x* is a vowel with the expression x in VOWELS.

It‘s tricky for us to deal with punctuation and numbers with what we know so far, so instead, ask the user to enter only words and spaces. You can convert their input from a string to a list of strings by calling split on the string:
–My name is John Smith|.split(– |) -> [–My|, –name|, –is|, –John|, –Smith|]

Bonus Task:

1. Add the following functions to the s_calc you created in Task 2A-1 and Task2A-2.
    a. sin(x)
    b. cos(x)
    c. tan(x)
    d. sqrt

Lab 2-C

Numpy:

Scientific Python code uses a fast array structure, called the numpy array. Those who have worked in Matlab will find this very natural. For reference, the numpy documention can be found https://numpy.org/doc/stable/reference/

Let’s make a numpy array:

```
import numpy as np
my_array = np.array([1, 2, 3, 4])
print(my_array)
```

Numpy arrays are listy! Below we compute length, slice, and iterate.

```
print(len(my_array))
print(my_array[2:4])
for ele in my_array:
    print(ele)
```

In general you should manipulate numpy arrays by using numpy module functions (np.mean, for example). This is for efficiency purposes, and a discussion follows below this section.

You can calculate the mean of the array elements either by calling the method .mean on a numpy array or by applying the function np.mean with the numpy array as an argument.

```
print(my_array.mean())
print(np.mean(my_array))
```

The way we constructed the numpy array above seems redundant..after all we already had a regular python list. Indeed, it is the other ways we have to construct numpy arrays that make them super useful.

There are many such numpy array *constructors*. Here are some commonly used constructors. Look them up in the documentation.

```
np.ones(10) # generates 10 floating point ones
```

Numpy gains a lot of its efficiency from being typed. That is, all elements in the array have the same type, such as integer or floating point. The default type, as can be seen above, is a float of size appropriate for the machine (64 bit on a 64 bit machine).

```
np.dtype(float).itemsize # in bytes
np.ones(10, dtype='int') # generates 10 integer ones
np.zeros(10)
```

Often you will want random numbers. Use the random constructor!

```
np.random.random(10) # uniform on [0,1]
```

You can generate random numbers from a normal distribution with mean 0 and variance 1:

```
normal_array = np.random.randn(1000)
print("The sample mean and standard devation are %f and %f, respectively."
%(np.mean(normal_array), np.std(normal_array)))
```

2D Arrays:

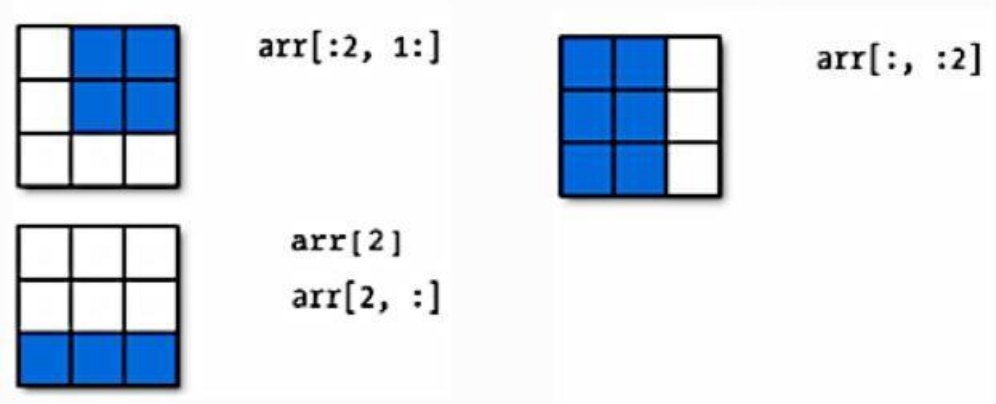
Similarly, we can create two-dimensional arrays.

```
my_array2d = np.array([ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ])
# 3 x 4 array of ones
ones_2d = np.ones([3, 4])
print(ones_2d)
# 3 x 4 array of ones with random noise
ones_noise = ones_2d + .01*np.random.randn(3, 4)
print(ones_noise)
# 3 x 3 identity matrix
my_identity = np.eye(3)
print(my_identity)
```

Like lists, numpy arrays are 0-indexed. Thus we can access the nnth row and the mmth column of a two-dimensional array with the indices [n−1,m−1][n−1,m−1].

```
print(my_array2d)
my_array2d[2, 3]
```

Numpy arrays are listy! They have set length (array dimensions), can be sliced, and can be iterated over with loop. Below is a schematic illustrating slicing two-dimensional array.



Earlier when we generated the one-dimensional arrays of ones and random numbers, we gave ones and random the number of elements we wanted in the arrays. In two dimensions, we need to provide the shape of the array, i.e., the number of rows and columns of the array.

```
onesarray = np.ones([3,4])
onesarray
```

You can transpose the array:

```
onesarray.shape
onesarray.T
onesarray.T.shape
```

Matrix multiplication is accomplished by np.dot. The * operator will do element-wise multiplication.

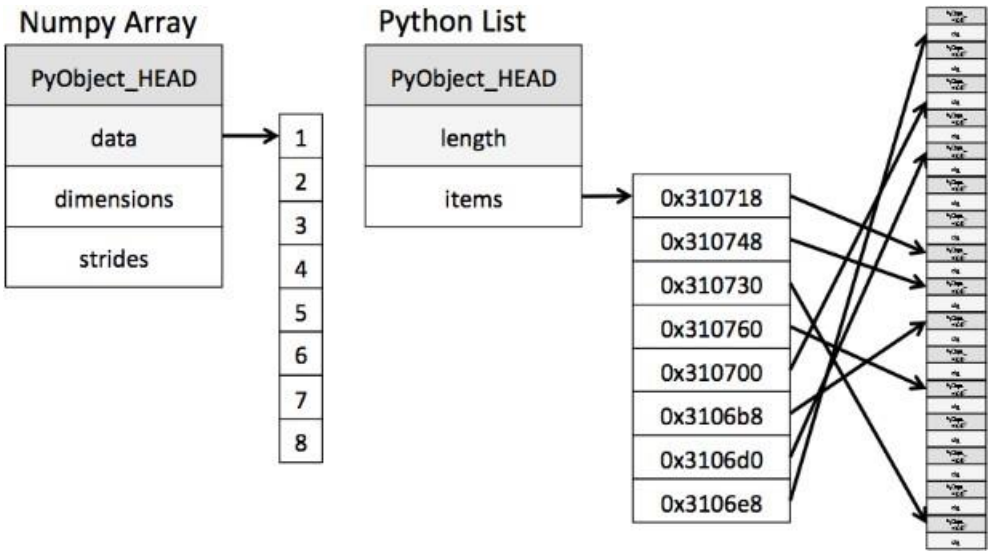
```
print(np.dot(onesarray, onesarray.T)) # 3 x 3 matrix
np.dot(onesarray.T, onesarray) # 4 x 4 matrix
```

Numpy functions will by default work on the entire array:

The axis 0 is the one going downwards (the y-axis, so to speak), whereas axis 1 is the one going across (the xx-axis). You will often use functions such as mean, sum, with an axis.

```
np.sum(onesarray, axis=0)
np.sum(onesarray, axis=1)
```

In the last lab we said that Python lists may contain items of different types. This flexibility comes at a price: Python lists store *pointers* to memory locations. On the other hand, numpy arrays are typed, where the default type is floating point. Because of this, the system knows how much memory to allocate, and if you ask for an array of size 100, it will allocate one hundred contiguous spots in memory, where the size of each spot is based on the type. This makes access extremely fast.



Pandas:

Before loading the data, lets understand the 2 key data structures in Pandas – Series and DataFrames

- Series can be understood as a 1 dimensional labelled / indexed array. You can access individual elements of this series through these labels.

- A dataframe is similar to Excel workbook – you have column names referring to columns and you have rows, which can be accessed with use of row numbers. The essential difference being that column names and row numbers are known as column and row index, in case of dataframes.

Often data is stored in comma separated values (CSV) files. For the remainder of this lab, we'll be working with automobile data, where we've extracted relevant parts below.

Note that CSV files can be output by any spreadsheet software, and are plain text, hence are a great way to share data.

Importing data from separate file:

Now let's read in our automobile data as a pandas *dataframe* structure.

```
# Read in the csv files
dfcars=pd.read_csv("../data/mtcars.csv")
type(dfcars)

dfcars.head()
```

What we have now is a spreadsheet with indexed rows and named columns, called a *dataframe* in pandas. dfcars is an *instance* of the pd.DataFrame *class*, created by calling the pd.read_csv "constructor function".

The take-away is that dfcars is a dataframe object, and it has methods (functions) belonging to it. For example, df.head() is a method that shows the first 5 rows of the dataframe.

A pandas dataframe is a set of columns pasted together into a spreadsheet, as shown in the schematic below, which is taken from the cheatsheet above. The columns in pandas are called *series* objects.

In [ ]:
```
#Read csv file
df = pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/Salaries.csv")
```

Note: The above command has many optional arguments to fine-tune the data import process.

There is a number of pandas commands to read other data formats:

```
pd.read_excel('myfile.xlsx',sheet_name='Sheet1', index_col=None, na_values=['NA'])
pd.read_stata('myfile.dta')
pd.read_sas('myfile.sas7bdat')
pd.read_hdf('myfile.h5','df')
```

Pandas Datatypes:

Pandas Type	Native Python Type	Description
object	string	The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings).
int64	int	Numeric characters. 64 refers to the memory allocated to hold this character.
float64	float	Numeric characters with decimals. If a column contains numbers and NaNs(see below), pandas will default to float64, in case your missing value has a decimal.
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	Values meant to hold time data. Look into these for time series experiments.

Dataframe Attributes:

Python objects have *attributes* and *methods*.

df.attribute	description
dtypes	list the types of the columns
columns	list the column names
axes	list the row labels and column names
ndim	number of dimensions
size	number of elements
shape	return a tuple representing the dimensionality
values	numpy representation of the data

Dataframe Methods:

Unlike attributes, python methods have *parenthesis*.
All attributes and methods can be listed with a *dir()* function: `dir(df)`

df.method()	description
head([n]), tail([n])	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample([n])	returns a random sample of the data frame
dropna()	drop all the records with missing values

Dataframe Filtering:

Filtering is used to select the subset of rows with certain values. For example, if we want to subset the rows in which the salary value is greater than $120k. Any boolean operator can be used to subset the data such as >, < , = =, <=, >= and !=.

```
df_sub=df[df['salary']>12000]
```

if we need to select range of rows, we can specify the range using ":".

```
df[10:20]
```

Aggregation Function in Pandas:
Computing a summary statistic about each group, i.e.

- compute group sums or means
- compute group sizes/counts

Common aggregation functions: min, max, count, sum, prod, mean, median, mode, mad, std, var

```
In [ ]: flights[['dep_delay','arr_delay']].agg(['min','mean','max'])
```

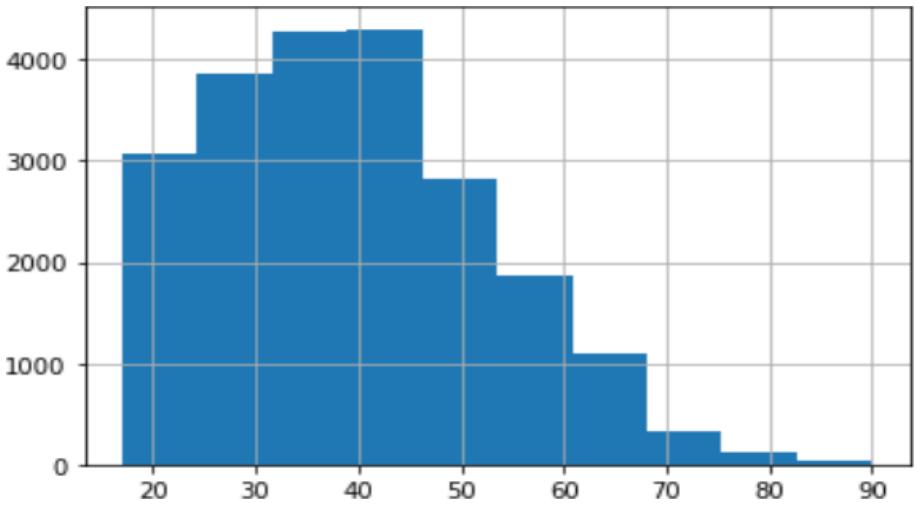
Out[ ]:

	dep_delay	arr_delay
min	-16.000000	-62.000000
mean	9.384302	2.298675
max	351.000000	389.000000

df.method()	description
describe	Basic statistics (count, mean, std, min, quantiles, max)
min, max	Minimum and maximum values
mean, median, mode	Arithmetic average, median and mode
var, std	Variance and standard deviation
sem	Standard error of mean
skew	Sample skewness
kurt	kurtosis

Creating a Histogram Using Pandas:

```
In [19]: ml_age.hist( histtype = 'stepfilled', bins = 10)
Out[19]: <AxesSubplot:>
```



Lab Journal 2-C:

1. Write a Pandas program to compute the Euclidean distance between two given series. The Euclideandistance or Euclidean metric is the "ordinary" straight-line distance between two points in Euclideanspace. With this distance, Euclidean space becomes a metric space.

   Series-1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
   Series-2: [11, 8, 7, 5, 6, 5, 3, 4, 7, 1]
2. Import onlineretail.csv dataset (https://www.kaggle.com/vijayuv/onlineretail)
   - display data
   - display summary of numerical fields
   - display first and last column
   - Find which columns have null/Nan values in them and fill those null/Nan with the mean of that column. (Do not drop the rows with Nan/Null you have to fill them)
   - Find out which countries contribute the most to the online retailer's sales revenue.
   - Analyze data by plotting histogram