National University of Computer and Emerging Sciences

AL-2002: Artificial Intelligence
Lab Manual 04

Department of Computer Science
FAST-NU, Lahore, Pakistan

Table of Contents

Objectives

After performing this lab, students shall be able to understand the following Python concepts and applications:

- ✔ Application of NumPy
- ✔ Application of Pandas
- ✔ Dataset handling
- ✔ Application of Matplotlib

1 Python Libraries for Data Science

Python is an easy-to-learn, easy-to-debug, widely used, object-oriented and open-source language. Python has been built with extraordinary libraries for data science that are used by programmers every day in solving problems. The top 10 Python libraries for data science include:

1. NumPy
2. Pandas
3. SciPy
4. TensorFlow
5. Matplotlib
6. Keras
7. SciKit-Learn
8. PyTorch
9. Scrapy
10. BeautifulSoup

1.1 Numpy

NumPy (Numerical Python) is the fundamental package for numerical computation in Python; it contains a powerful N-dimensional array object. This is the foundation on which almost all the power of Python's data science toolkit is built, and learning NumPy is the first step on any Python data scientist's journey. NumPy also addresses the slowness problem partly by providing these multidimensional arrays as well as providing functions and operators that operate efficiently on these arrays. Here are the top four benefits that NumPy can bring to your code:

1. **More speed:** NumPy uses algorithms written in C that complete in nanoseconds rather than seconds.
2. **Fewer loops:** NumPy helps you to reduce loops and keep from getting tangled up in iteration indices.
3. **Clearer code:** Without loops, your code will look more like the equations you're trying to calculate.
4. **Better quality:** There are thousands of contributors working to keep NumPy fast, friendly, and bug free.

Find more details about NumPy here.

1.1.1 NumPy Installation

- **Install NumPy with pip**

  To install NumPy with **pip**, bring up a terminal window and type:

  ```
  $ pip install numpy
  ```

  This command installs NumPy in the current working Python environment.

- **Install Numpy in Anaconda**
  For simple installation via Anaconda, you can follow the instructions given here.

Import Convention

To use numpy in the program we need to import the module. Generally, numpy package is defined as np of abbreviation for convenience. But you can import it using anything you want. The recommended convention to import numpy is:

```
>>> import numpy as np
```

1.1.2 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.

Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Figure 1 Array Creation

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
import numpy as np

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

Numpy also provides many functions to 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]]"
```

Initial Placeholders

`>>> np.zeros((3,4))`	Create an array of zeros
`>>> np.ones((2,3,4),dtype=np.int16)`	Create an array of ones
`>>> d = np.arange(10,25,5)`	Create an array of evenly spaced values (step value)
`>>> np.linspace(0,2,9)`	Create an array of evenly spaced values (number of samples)
`>>> e = np.full((2,2),7)`	Create a constant array
`>>> f = np.eye(2)`	Create a 2X2 identity matrix
`>>> np.random.random((2,2))`	Create an array with random values
`>>> np.empty((3,2))`	Create an empty array

Figure 2 Numpy Functions

1.1.3 Attributes of Array Object

`ndarray.shape`: The dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with *n* rows and *m* columns, shape will be `(n,m)` .

`ndarray.ndim`: The number of axes (dimensions) of the array.

`ndarray.dtype`: Retrieves the data type of array.

`ndarray.itemsize`: The size in bytes of each element of the array. For example, an array of elements of type float64 has *itemsize* 8 (=64/8), while one of type *complex32* has itemsize 4 (=32/8). It is equivalent to *ndarray.dtype.itemsize.*

`ndarray.size`: The total number of elements of the array. This is equal to the product of the elements of shape.

```
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]"
print(a.ndim)              #Prints 1
print(len (a))             #Prints 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"
```

1.1.4 Array Slicing and Indexing

Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
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"
```

```
a = np.arange(4)**3  # create array a
a[2]                 # member of a array in 2nd position
a[::-1]              # reversed a
a[0:4,1]             # each row in the second column of b
a[1,...]             # same as a[1,:,:] or a[1]
a[a>5]               # a with values greater than 5
x = a[0:4]           # assign x with 4 values of a
x[:]=99              # change the values of x to 99 which will change
the 4 values of a also.
```

Figure 3 Indexing and Slicing Techniques

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

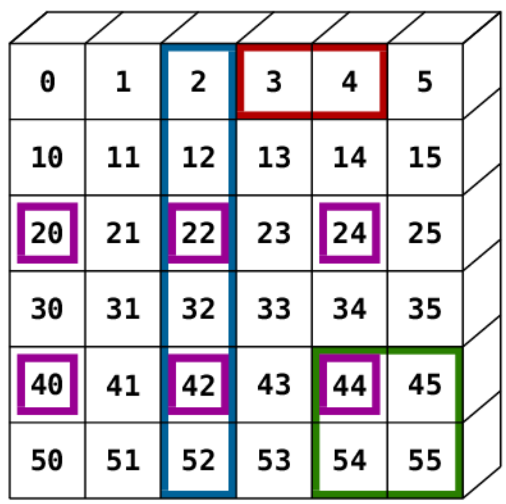


Figure 4 Indexing and slicing illustration using NumPy Array

1.1.5 Basic Array Operations

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
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))
```

```
A = np.array([[1,1],[0,1]])
B = np.array([[2,0],[3,4]])
A+B                 #addition of two array
np.add(A,B)         #addition of two array
A * B               # elementwise product
A @ B               # matrix product
A.dot(B)            # another matrix product
B.T                 #Transpose of B array
A.flatten()         #form 1-d array
B < 3               #Boolean of Matrix B. True for elements less than 3
A.sum()             # sum of all elements of A
A.sum(axis=0)       # sum of each column
A.sum(axis=1)       # sum of each row
A.cumsum(axis=1)    # cumulative sum along each row
A.min()             # min value of all elements
A.max()             # max value of all elements
np.exp(B)           # exponential
np.sqrt(B)          # squre root
A.argmin()          #position of min value of elements
A.argmax()          #position of max value of elements
A[1,1]              #member of a array in (1,1) position
```

Figure 5 Array operations

1.1.6 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

If you want to know the data type of an array, you can query the attributes of *dtype*. An object describing the type of the elements in the array. One can create or specify *dtype* using standard Python types.

Additionally, *numpy* provides types of its own. *numpy.int32*, *numpy.int16*, and *numpy.float64* are some examples.

Data Types

`>>> np.int64`	Signed 64-bit integer types
`>>> np.float32`	Standard double-precision floating point
`>>> np.complex`	Complex numbers represented by 128 floats
`>>> np.bool`	Boolean type storing `TRUE` and `FALSE` values
`>>> np.object`	Python object type
`>>> np.string_`	Fixed-length string type
`>>> np.unicode_`	Fixed-length unicode type

Figure 6 Data Types

Here is an example:

```
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"
```

1.2 Pandas

The Pandas library is one of the most preferred tools for data scientists to do data manipulation and analysis. It can work with data from a wide variety of sources. `Pandas` is suited for many different kinds of data: tabular data, time-series data, arbitrary matrix data with row and column labels, and any other form of observational/statistical data sets.

Features:

1. Enables you to create your own function and run it across a series of data
2. Contains high-level data structures and manipulation tools

1.2.1 Pandas Installation

- **Installation via Terminal**
  To install pandas in your system you can use this command `pip install pandas` or `conda install pandas`.

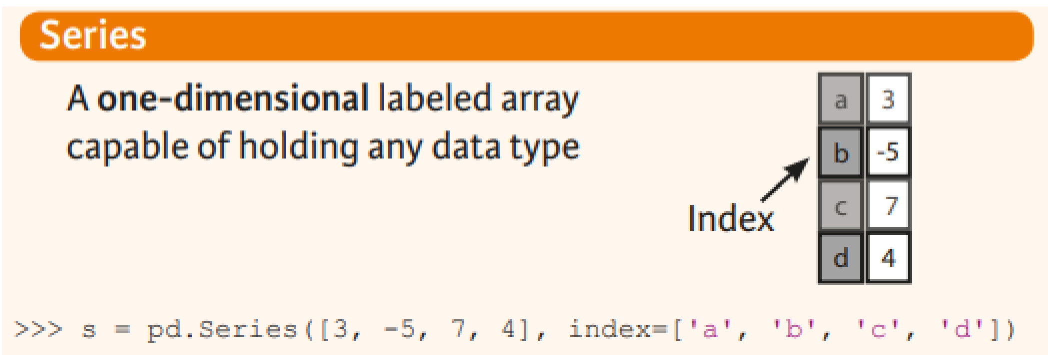
- **Installation in Anaconda**
  Pandas or any other packages can be installed in Anaconda environment by following the tutorial here.

Import Convention

```
>>> import pandas as pd
```

1.2.2 Series

Pandas series works the same way both in list and numpy array as well as dictionary. To make series in `pandas` we need to use `pd.Series(data, index)` format where `data` are input data and `index` are selected index for data. A one-dimensional labeled array capable of holding any data type:




```
>>> s = pd.Series([3, -5, 7, 4],  index=['a',  'b',  'c',  'd'])
```

A	3
B	5
C	7
D	4

Pandas series along with numPy array:

```
import numpy as np           #importing numpy
import pandas as pd          #importing pandas
arr=np.array([1,3,5,7,9])    #create arr array
```

```
s2=pd.Series(arr)            #create pandas series s2
print(s2)                    #print s2
print(type(s2))              #print type of s2
```

Output:

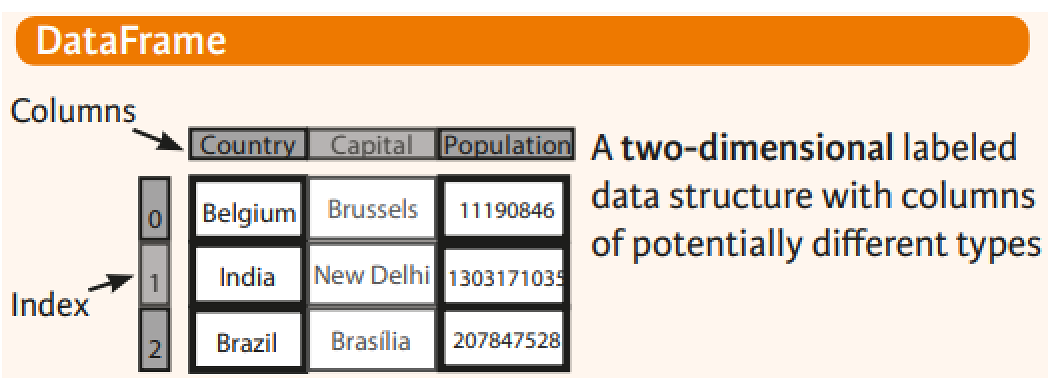
```
0     1
1     3
2     5
3     7
4     9
dtype: int64
<class 'pandas.core.series.Series'>
```

1.2.3 Dataframe

Pandas DataFrame is a way to store data in rectangular grids that can easily be overviewed. It's like a tabular data structure with labeled axes (rows and columns). The default format of a DataFrame would be `pd.Dataframe(data, index, column)` . You need to mention the data, index and columns value to generate a DataFrame. Data should be at least *two-dimensional*, *index* will be the row name and *columns* values for the columns. In general, you could say that the Pandas DataFrame consists of three main components: the data, the index, and the columns.

The DataFrame can contain data that is:

1. Pandas DataFrame
2. Pandas Series: An example of a Series object is one column from a DataFrame.
3. a NumPy ndarray, which can be a record or structured
4. a two-dimensional ndarray
5. lists, dictionaries or Series.




```
>>> data = {'Country': ['Belgium',  'India',  'Brazil'],
'Capital': ['Brussels',  'New Delhi',  'Brasilia'],
```

```
'Population': [11190846, 1303171035, 207847528]}
>>> df = pd.DataFrame(data,columns=['Country', 'Capital', 'Population'])
```

To get subset of a Data Frame:

```
>>> df[1:]
Country     Capital   Population
1  India    New Delhi 1303171035
2  Brazil   Brasilia  207847528
```

```
import pandas as pd
s2 = pd.Series([10, 20, 30])
print(s2)
print(type(s2))
s3=pd.DataFrame([[1,2],[3,4]],columns=['A','B'], index = ['C', 'D'])
print(s3)
print(type(s3))
```

```
0    10
1    20
2    30
dtype: int64
<class 'pandas.core.series.Series'>
   A  B
C  1  2
D  3  4
<class 'pandas.core.frame.DataFrame'>
```

Figure 7 Example of Pandas Series and DataFrame

1.2.4 Indexing

- `.loc[]` works on labels of your index. This means that if you give in `loc[2]`, you look for the values of your DataFrame that have an index labeled `2`.
- `.iloc[]` works on the positions in your index. This means that if you give in `iloc[2]`, you look for the values of your DataFrame that are at index '2'.

1.2.5 Input Output Operations

Read and Write to CSV

```
pd.read_csv('file.csv', header=None, nrows=5)
```

```
df.to_csv('myDataFrame.csv')
```

1.2.6 Other Operations

- Set index a of Series s to 6

```
>>> s['a'] = 6
```

- Sort by the values along an axis

```
>>> df.sort_values(by='Country')
```

- Assign ranks to entries

```
>>> df.rank()
```

1.2.7 Retrieving Series/DataFrame Information

- Basic Information (rows, columns)

```
>>> df.shape
```

- Describe index

```
>>> df.index
```

- Describe DataFrame columns

```
>>> df.columns
```

- Info on DataFrame

```
>>> df.info()
```

- Number of non-NA values

```
>>> df.count()
```

- Sum of values

```
>>> df.sum()
```

- Cumulative sum of values

```
>>> df.cumsum()
```

- Minimum/maximum values

```
>>> df.min()/df.max()
```

- Minimum/Maximum index value

```
>>> df.idxmin()/df.idxmax()
```

1.2.8 Application of Functions

```
>>> f = lambda x: x*2
```

Apply function

```
>>> df.apply(f)
```

Find more details about Pandas here.

2 Dataset Handling

2.1 Significance of Data

A data set is a set or collection of data. This set is normally presented in a tabular pattern. Every column describes a particular variable. And each row corresponds to a given member of the data set, as per the given question. The data are essentially organized to a certain model that helps to process the needed information. This set of data is any permanently saved collection of information that usually contains either case-level, gathered data, or statistical guidance level data.

In Machine Learning projects, it is impossible for an "AI" to learn without data. During an AI development, we always rely on data. From training, tuning, model selection to testing, we use three different data sets: the training set, the validation set and the testing set.

- **Training data set**: The training data set is the one used to train an algorithm to understand how to apply concepts such as neural networks, to learn and produce results. It is the actual **data set** used to train the model for performing various actions. It includes both input data and the expected output.

- **Test data set**: The test data set is used to evaluate how well your algorithm was trained with the training data set. In AI projects, we can't use the training data set in the testing stage because the algorithm will already know in advance the expected output which is not our goal.

2.2 Types of Data Sets

In Statistics, we have different types of data sets available for different types of information. They are:

- Numerical data sets
- Bivariate data sets
- Multivariate data sets
- Categorical data sets
- Correlation data sets

Let us discuss all these data sets with examples.

2.2.1 Numerical Data Sets

A set of all numerical data. It deals only with numbers. Some of the examples are;

- Weight and height of a person
- The count of RBC in a medical report
- Number of pages present in a book

2.2.2 Bivariate Data Sets

A data set that has two variables is called a Bi-variate data set. It deals with the relationship between the two variables.

Example: To find the percentage score and age of the students in a class. Score and age can be considered as two variables.

2.2.3 Multivariate Data Sets

A data set with multiple variables.

Example: If we have to measure the length, width, height, volume of a rectangular box, we have to use multiple variables to distinguish between those entities.

2.2.4 Categorical Data Sets

Categorical data sets represent features or characteristics of a person or an object.

Example:

- A person's gender (male or female)
- Marital status (married/unmarried)

2.2.5 Correlation Data Sets

The set of values that demonstrate some relationship with each other indicates correlation data sets. Here the values are found to be dependent on each other.

Example: A tall person is considered to be heavier than a short person. So here the weight and height variables are dependent on each other.

2.3 Mean, Median, Mode and Range of Data-Sets

The mean, median and mode along with range are the major topics in Statistics. Let us get through with respect to data-sets here.

Mean of a data-set is the average of all the observations present in the table. It is the ratio of the sum of observations to the total number of elements present in the data-set. The formula of mean is given by;

Mean = Sum of Observations / Total Number of Elements in Data Set

Median of a data-set is the middle value of the collection of data when arranged in ascending order and descending order.

Mode of a data-set is the variable or number or value which is repeated maximum number of times in the set.

Range of a data set is the difference between the maximum value and minimum value.

Range = Maximum Value – Minimum Value

Example:
Find the mean, mode, median and range of the given data set.
{2, 4, 6, 8, 2, 10, 12}
Solution:
Given, {2, 4, 6, 8, 2, 10, 12} is a set of data.
Mean = 2+4+6+8+2+10+12/7 = 44/7
To find median we have to first arrange the given data in ascending or descending order
So, {2,2,4,6,8,10,12}. Thus,
Median = 6
Mode = 2
Range = 12-2 = 10

2.4 Types of Attributes

1. **Nominal**
   Examples: ID numbers, eye color, zip codes
2. **Ordinal**
   Examples: rankings (e.g., taste of potato chips on a scale from 1-10), grades, height in {tall, medium, short}
3. **Interval**
   Examples: calendar dates, temperatures in Celsius or Fahrenheit.
4. **Ratio**
   Examples: temperature in Kelvin, length, time, counts

2.5 Dataset Repositories

Datasets for analysis can be downloaded from common repositories like:

1. Kaggle
2. UCI

Check out this & this links for a list of various data repositories.

2.6 Properties of Attributes in a dataset

2.6.1 Missing Values

Some entries can be missing because of the following reasons:

1. **Data Extraction**: It is possible that there are problems with extraction process. In such cases, we should double-check for correct data with data guardians. Errors at data extraction stage are typically easy to find and can be corrected easily as well.
2. **Data collection**: These errors occur at time of data collection and are harder to correct.

2.6.2 Outliers

Outlier is an observation that appears far away and diverges from an overall pattern in a sample. Let's take an example, we do customer profiling and find out that the average annual income of customers is $1 lakh. But, there are two customers having annual income of $4 and $4.2 million. These two customers annual income is much higher than rest of the population. These two observations will be seen as Outliers.

Outliers can be due to the following reasons:

1. **Data Entry Errors:** Human errors such as errors caused during data collection, recording, or entry can cause outliers in data. For example: Annual income of a customer is $100,000. Accidentally, the data entry operator puts an additional zero in the figure. Now the income becomes $1,000,000 which is 10 times higher. Evidently, this will be the outlier value when compared with rest of the population.
2. **Measurement Error:** It is the most common source of outliers. This is caused when the measurement instrument used turns out to be faulty. For example: There are 10 weighing machines. 9 of them are correct, 1 is faulty. Weight measured by people on the faulty machine will be higher / lower than the rest of people in the group. The weights measured on faulty machine can lead to outliers.
3. **Experimental Error:** Another cause of outliers is experimental error. For example: In a 100m sprint of 7 runners, one runner missed out on concentrating on the 'Go' call which caused him to start late. Hence, this caused the runner's run time to be more than other runners. His total run time can be an outlier.
4. **Data Processing Error:** Whenever we perform data mining, we extract data from multiple sources. It is possible that some manipulation or extraction errors may lead to outliers in the dataset.
5. **Sampling error:** For instance, we have to measure the height of athletes. By mistake, we include a few basketball players in the sample. This inclusion is likely to cause outliers in the dataset.

2.7 Iris Flower Dataset

The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced by the British statistician. The dataset contains 150 observations of iris flowers. There are four columns of measurements of the flowers in centimeters. The fifth column is the species of the flower observed. All observed flowers belong to one of three species. The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor).

To gain more insights about this dataset check [this link].

You can download this dataset from here.

2.8 Load the Dataset

In order to load the dataset, run Anaconda Navigator.

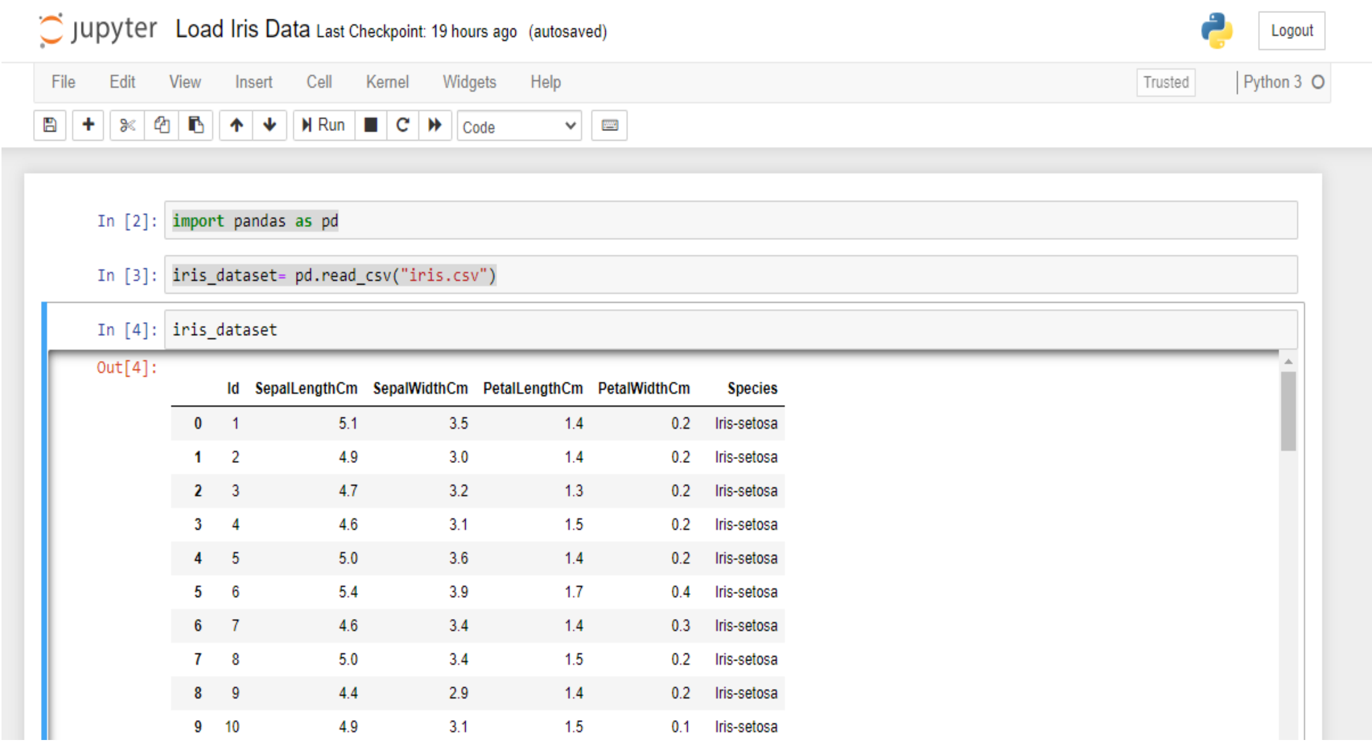
1. Launch Jupyter Notebook
2. Click on “Upload” button as shown in fig. 10
3. Select the dataset file from your directory
4. Create a new Python3 notebook by selecting “New” on the Jupyter host page
5. Follow the instructions given in the code snippet provided next

Figure 8 Load Dataset in Anaconda

Open the newly created notebook and implement the code given below:

```
import pandas as pd

iris_dataset= pd.read_csv("iris.csv")
iris_dataset
```

*Figure SEQ Figure * ARABIC9 Code instructions*




```
In [2]: import pandas as pd

In [3]: iris_dataset= pd.read_csv("iris.csv")

In [4]: iris_dataset
Out[4]:
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa
5	6	5.4	3.9	1.7	0.4	Iris-setosa
6	7	4.6	3.4	1.4	0.3	Iris-setosa
7	8	5.0	3.4	1.5	0.2	Iris-setosa
8	9	4.4	2.9	1.4	0.2	Iris-setosa
9	10	4.9	3.1	1.5	0.1	Iris-setosa

For more information about file handling using Pandas, visit this site.

3 Visualization

3.1 Matplotlib

Matplotlib is a low-level graph plotting library in python that serves as a visualization utility. Matplotlib was created by John D. Hunter. Matplotlib is open source and we can use it freely. Matplotlib is mostly written in python, a few segments are written in C, Objective-C and JavaScript for Platform compatibility.

3.1 Installation of Matplotlib

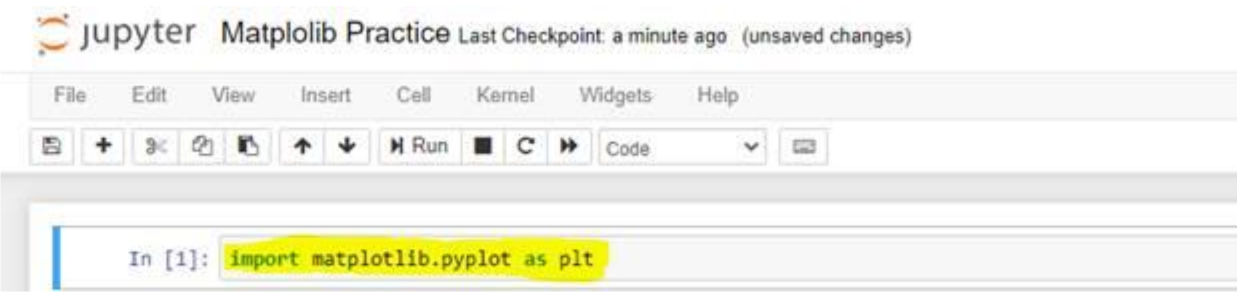
If you have Python and PIP already installed on a system, then installation of Matplotlib is very easy.

```
pip install matplotlib
```

But mostly distribution like Anaconda, Spyder have pre-installed matplotlib.

3.2 Pyplot

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias:

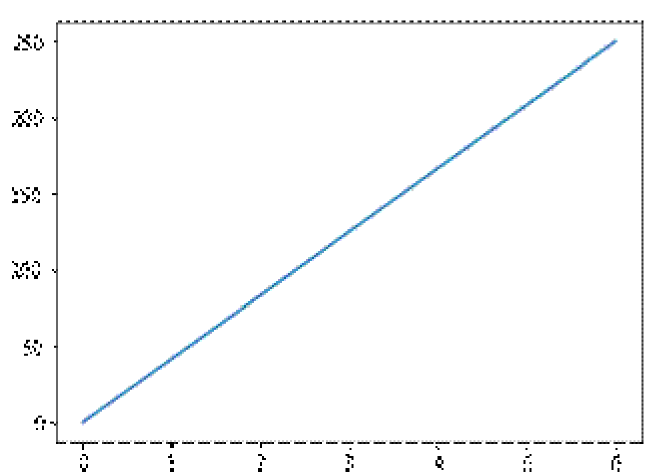



```
import matplotlib.pyplot as plt
```

Now the Pyplot package can be referred to as plt.

Example

Draw a line in a diagram from position (0,0) to position (6,250):

Code	Output
import matplotlib.pyplot as plt import numpy as np xpoints=np.array([0, 6]) ypoints=np.array([0, 250]) plt.plot(xpoints, ypoints) plt.show()	

3.3 Plotting x and y points

The plot() function is used to draw points (markers) in a diagram.

By default, the plot() function draws a line from point to point.

The function takes parameters for specifying points in the diagram.

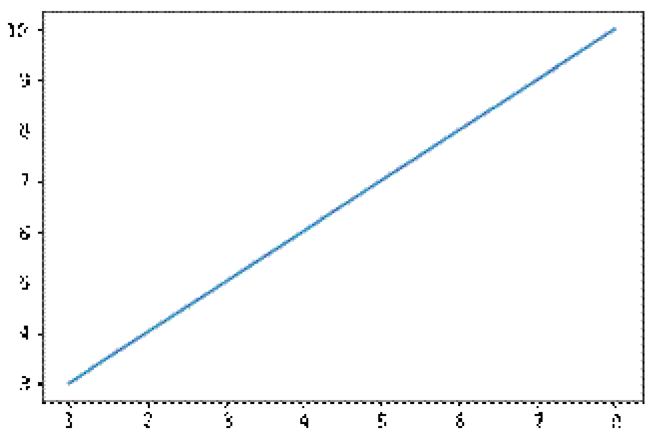
Parameter 1 is an array containing the points on the **x-axis**.

Parameter 2 is an array containing the points on the **y-axis**.

If we need to plot a line from (1, 3) to (8, 10), we have to pass two arrays [1, 8] and [3, 10] to the plot function.

Example

Draw a line in a diagram from position (1, 3) to position (8, 10):

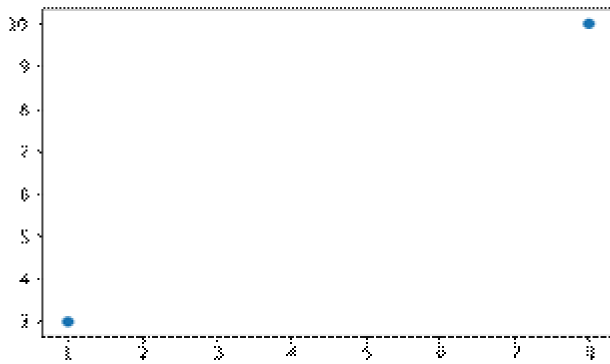
Code	Output
import matplotlib.pyplot as plt import numpy as np xpoints = np.array([1, 8]) ypoints = np.array([3, 10]) plt.plot(xpoints, ypoints) plt.show()	

There are many types of single lines/multiple lines that can be drawn, explore other types at: https://www.w3schools.com/python/matplotlib_line.asp

3.4 Plotting Without Line

To plot only the markers, you can use *shortcut string notation* parameter 'o', which means 'rings'.

Draw two points in the diagram, one at position (1, 3) and one in position (8, 10):

Code	Output
import matplotlib.pyplot as plt import numpy as np xpoints = np.array([1, 8]) ypoints = np.array([3, 10]) plt.plot(xpoints, ypoints, 'o') plt.show()	

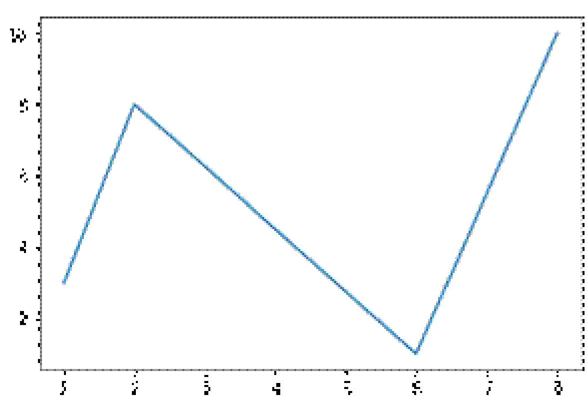
There can be different type of markers, you can explore at:
https://www.w3schools.com/python/matplotlib_markers.asp

3.5 Multiple Points

You can plot as many points as you like, just make sure you have the same number of points in both axis.

Example

Draw a line in a diagram from position (1, 3) to (2, 8) then to (6, 1) and finally to position (8, 10):

Code	Output
import matplotlib.pyplot as plt import numpy as np xpoints = np.array([1, 2, 6, 8]) ypoints = np.array([3, 8, 1, 10]) plt.plot(xpoints, ypoints) plt.show()	

3.6 Default X-Points

If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, (etc. depending on the length of the y-points.

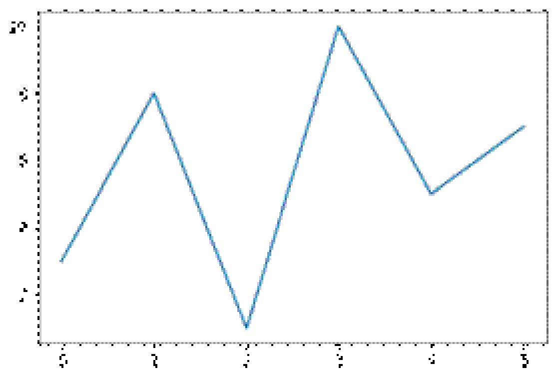
So, if we take the same example as above, and leave out the x-points, the diagram will look like this:

Example

Plotting without x-points:

<table>
<tr><th>Code</th><th>Output</th></tr>
<tr><td>
<pre>
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10, 5, 7])

plt.plot(ypoints)
plt.show()
</pre>
</td><td></td></tr>
</table>

The x-points in the example above are [0, 1, 2, 3, 4, 5] by default.

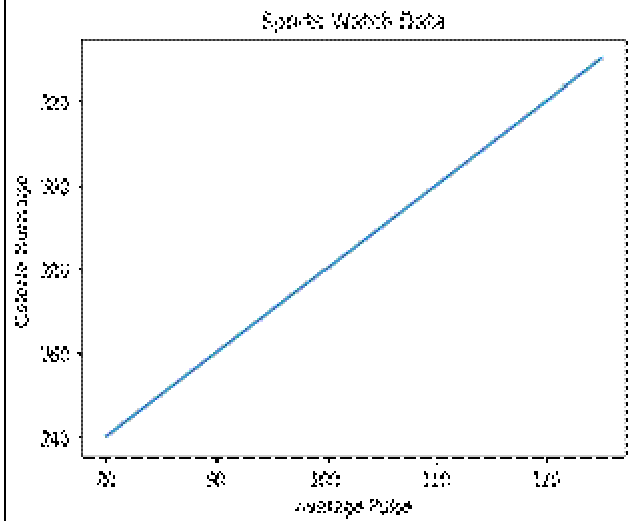
3.7 Create Labels and title for a Plot

With Pyplot, you can use the xlabel() and ylabel() functions to set a label for the x- and y-axis.

Example

Add labels to the x- and y-axis:

Code	Output

import numpy as np import matplotlib.pyplot as plt x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125]) y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330]) plt.plot(x, y) plt.title("Sports Watch Data") plt.xlabel("Average Pulse") plt.ylabel("Calorie Burnage") plt.show()	

3.8 Add Grid Lines to a Plot

With Pyplot, you can use the grid() function to add grid lines to the plot.

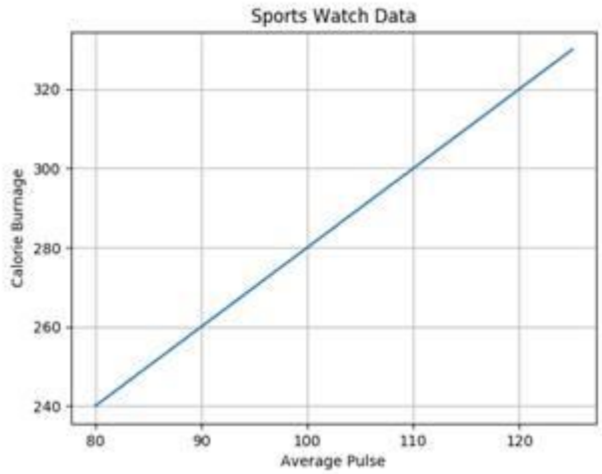
Example

Add grid lines to the plot:

Code	**Output**

<table>
<tr>
<td>

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100,

105, 110, 115, 120, 125])


y = np.array([240, 250, 260, 270,

280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)

plt.grid()

plt.show()
```

</td>
<td>


</td>
</tr>
</table>

Different type of grid can be generated, for more details see:
https://www.w3schools.com/python/matplotlib_grid.asp

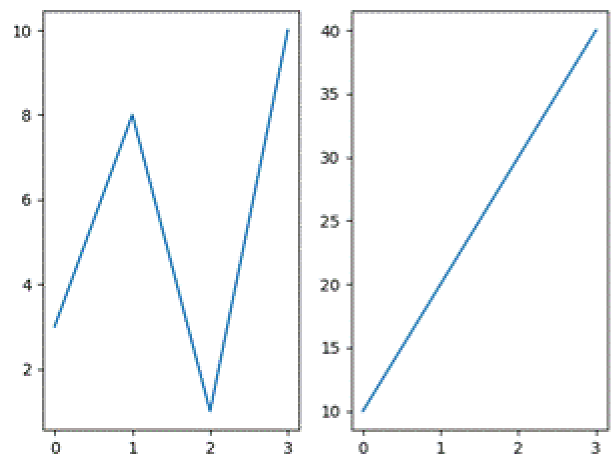
3.9 Display Multiple Plots

With the subplots() function you can draw multiple plots in one figure:

Example

Draw 2 plots:

Code	Output

<pre>import matplotlib.pyplot as plt
import numpy as np
 	
#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
 	
plt.subplot(1, 2, 1)
plt.plot(x,y)
 	
#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
 	
plt.subplot(1, 2, 2)
plt.plot(x,y)
 	
plt.show()</pre>
	

There different ways to plot multiple plots:
https://www.w3schools.com/python/matplotlib_subplots.asp

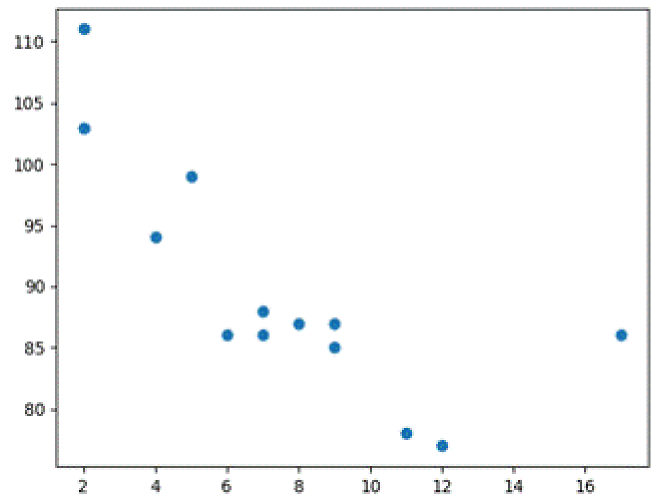
3.10 Creating Scatter Plots

With Pyplot, you can use the scatter() function to draw a scatter plot.

The scatter() function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis:

Example:

Code	Output

import matplotlib.pyplot as plt import numpy as np x=np.array([5,7,8,7,2,17,2,9, 4,11,12,9,6]) y=np.array([99,86,87,88,111, 86,103,87,94,78,77,85,86]) plt.scatter(x,y) plt.show()	

Explanation of above plot:

The observation in the example above is the result of 13 cars passing by. The X-axis shows how old the car is. The Y-axis shows the speed of the car when it passes. Are there any relationships between the observations? It seems that the newer the car, the faster it drives, but that could be a coincidence, after all we only registered 13 cars.

There are different type of scatter graphs that can be created (kindly see the link given, as all examples will make the manual lengthy):
https://www.w3schools.com/python/matplotlib_scatter.asp

3.11 Creating Bars

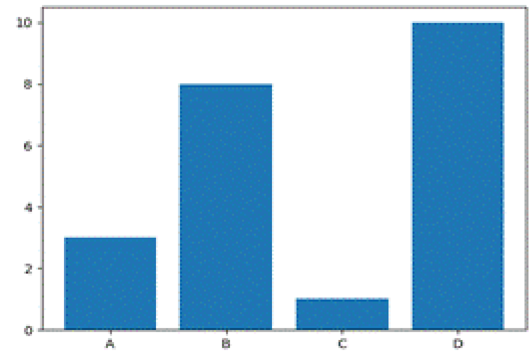
With Pyplot, you can use the bar() function to draw bar graphs:

Example

Draw 4 bars:

Code	Output

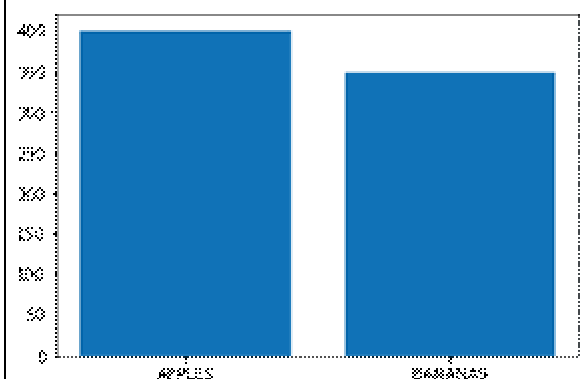
```python
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x,y)
plt.show()
```



The bar() function takes arguments that describes the layout of the bars.

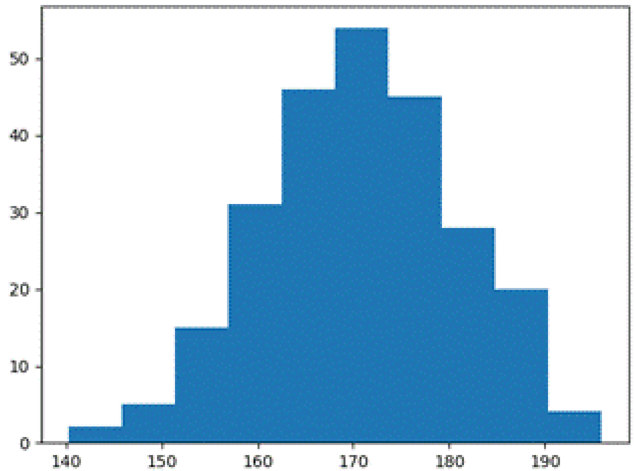
The categories and their values represented by the *first* and *second* argument as arrays.

```python
import matplotlib.pyplot as plt

x = ["APPLES", "BANANAS"]
y = [400, 350]
plt.bar(x, y)
```



3.12 Histogram

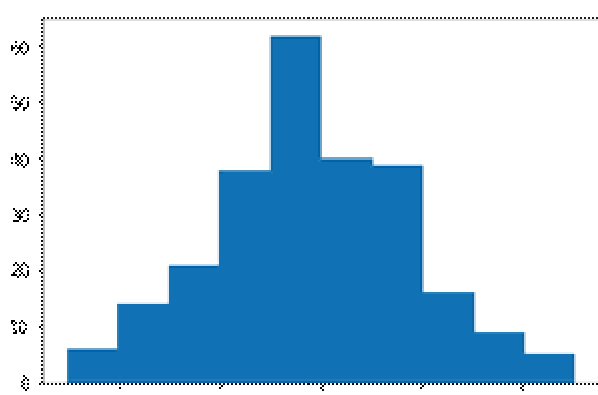
A histogram is a graph showing *frequency* distributions. It is a graph showing the number of observations within each given interval. Example: Say you ask for the height of 250 people; you might end up with a histogram like this:

You can read from the histogram that there are approximately: 2 people from 140 to 145cm 5 people from 145 to 150cm 15 people from 151 to 156cm 31 people from 157 to 162cm 46 people from 163 to 168cm 53 people from 168 to 173cm 45 people from 173 to 178cm 28 people from 179 to 184cm 21 people from 185 to 190cm 4 people from 190 to 195cm	

3.12.1 Create Histogram

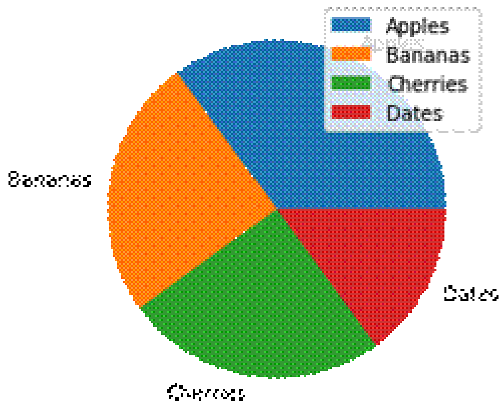
In Matplotlib, we use the hist() function to create histograms.

The hist() function will use an array of numbers to create a histogram, the array is sent into the function as an argument. For simplicity we use NumPy to randomly generate an array with 250 values, where the values will concentrate around 170, and the standard deviation is 10.

Code	**Output**
import matplotlib.pyplot as plt import numpy as np x = np.random.normal(170, 10, 250) plt.hist(x) plt.show()	

3.13 Creating Pie Charts

With Pyplot, you can use the pie() function to draw pie charts:

Code	Output
import matplotlib.pyplot as plt import numpy as np y = np.array([35, 25, 25, 15]) mylabels = ["Apples","Bananas","Cherries","Dates"] plt.pie(y, labels = mylabels) plt.legend() plt.show()	

As you can see the pie chart draws one piece (called a wedge) for each value in the array (in this case [35, 25, 25, 15]).

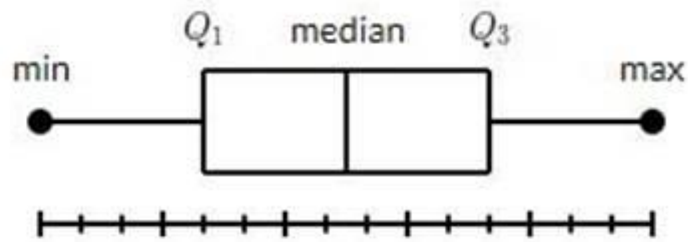
By default, the plotting of the first wedge starts from the x-axis and move *counterclockwise*:

Note: The size of each wedge is determined by comparing the value with all the other values, by using this formula:

The value divided by the sum of all values: x/sum(x)

3.14 Box Plot

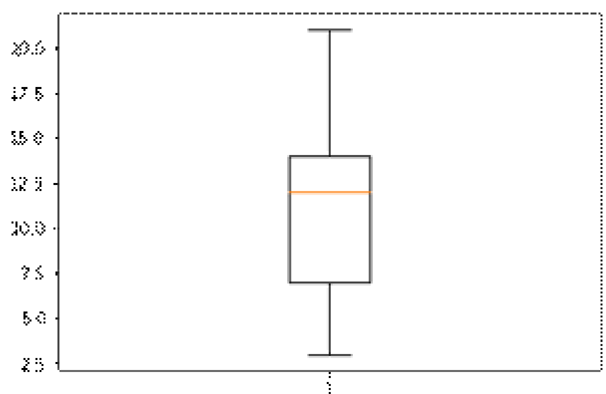
A box plot which is also known as a whisker plot displays a summary of a set of data containing the minimum, first quartile, median, third quartile, and maximum. In a box plot, we draw a box from the first quartile to the third quartile. A vertical line goes through the box at the median. The whiskers go from each quartile to the minimum or maximum.



The image is taken from: https://www.tutorialspoint.com/matplotlib/matplotlib_box_plot.htm

Example 1: Draw a box-and-whisker plot for the data set {3, 7, 8, 5, 12, 14, 21, 13, 18}.

Minimum: 3, Q_1: 6, Median: 12, Q_3: 16, and Maximum: 21.

Code	Output
import matplotlib.pyplot as plt data = [3, 7, 8, 5, 12, 14, 21, 13, 18] plt.boxplot(data) plt.show()	

AI Lab – 03

Python Libraries for Data Science & Visualization

Course: AL-2002 / CL-461 – Artificial Intelligence Lab
Lab Focus: NumPy, Pandas, Dataset Handling, Matplotlib
Total Marks: 20
Tools: Python, Jupyter Notebook

Task 1: NumPy Array Creation & Attributes (3 Marks)

1. Create the following NumPy arrays:
    - A 1D array with values `[10, 20, 30, 40, 50]`
    - A 2×3 array filled with ones
    - A 3×3 identity matrix
2. Print the following properties for each array:
    - `shape`
    - `ndim`
    - `dtype`
    - `size`

📌 *Objective:* Understanding NumPy array structure and attributes.

Task 2: NumPy Indexing & Slicing (3 Marks)

Given the matrix:

```
[[ 2,  4,  6,  8],
 [10, 12, 14, 16],
 [18, 20, 22, 24]]
```

Perform the following:

1. Extract the second row.
2. Extract the last two columns.
3. Modify the value `14` to `99` using indexing.
4. Print the updated array.

📌 *Objective:* Practice slicing and indexing in multidimensional arrays.

Task 3: NumPy Array Operations (3 Marks)

1. Create two NumPy arrays of size 2×2.
2. Perform:
    - Element-wise addition
    - Element-wise subtraction
    - Element-wise multiplication
    - Element-wise division
3. Compute the square root of the first array.

📌 *Objective:* Apply mathematical operations using NumPy.

Task 4: Pandas Series & DataFrame (3 Marks)

1. Create a Pandas **Series** representing marks of 5 students.
2. Create a **DataFrame** with the following columns:
   - `Student_Name`
   - `Marks`
   - `Grade`
3. Display:
   - First two rows
   - Shape of the DataFrame
   - Column names

📌 *Objective:* Understanding Series and DataFrame structures.

Task 5: Dataset Handling using Iris Dataset (4 Marks)

1. Load the **Iris dataset** using Pandas.
2. Display:
   - First 5 rows
   - Dataset shape
   - Column names
3. Compute:
   - Mean of each numerical attribute
   - Maximum sepal length
4. Identify if the dataset contains missing values.

📌 *Objective:* Real-world dataset handling and exploration.

Task 6: Statistical Measures (2 Marks)

Using a numeric column from the Iris dataset:

1. Calculate:
   - Mean
   - Median
   - Mode
   - Range

📌 *Objective:* Apply statistical concepts discussed in the lab manual.

Task 7: Data Visualization using Matplotlib (4 Marks)

Using the Iris dataset:

1. Plot a **line graph** between Sepal Length and Sepal Width.
2. Create a **scatter plot** between Petal Length and Petal Width.
3. Draw a **histogram** of Sepal Length.
4. Add:
    - Title
    - X-axis label
    - Y-axis label
    - Grid

📌 *Objective:* Visualize data patterns using Matplotlib.

Task 8: Box plot

Create a **box plot** for Petal Length and identify outliers visually.