Numpy

- Numpy stands for Numerical Python.
- It was created in 2005 by Travis Oliphant. And it is an open source project, so we can use it freely.
- Numpy is a Python Library and used to work with arrays. It has functions also which can use to work with linear algebra, Fourier transforms and matrices.
- In Python, we have lists that serve the purpose of arrays, but they are slow in process.
- Numpy provides an array object, that works faster than traditional Python lists.
- The array object in Numpy is called ndarray, and it provides lot of supporting functions.
- Numpy arrays are stored at one continuous place in memory, so it will work faster than lists.
- Numpy library is written partially in Python, but most of the modules which are required for fast computation are written in C or C++.
- We can install Numpy using PIP, if Python and PIP are already installed on a system.
- We can import Numpy after installation using: import numpy

```
#!/usr/bin/env python
# coding: utf-8

# In[2]:
import numpy
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
```
[1 2 3 4 5]

```
# In[3]:
#Numpy can be imported under the name np
import numpy as np
arr = np.array([1, 2, 3, 4, 5,6,7,8,9,10])
print(arr)
```
[1 2 3 4 5 6 7 8 9 10]

```
# In[4]:
#to check the numpy version
import numpy as np
print(np.__version__)
```
1.21.5
```
# In[5]:
#Numpy is used to work with arrays, and the array object in Numpy
is called ndarray.
```

```
#So we can create ndarray object in Numpy using the array function.

import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```
[1 2 3 4 5]
<class 'numpy.ndarray'>

```
# In[10]:
#To create an ndarray, we can pass a list, tuple or any array-like
object into the array() method,
#and it will be converted into an ndarray:
import numpy as np
arr = np.array((1, 2, 3, 4, 5)) #tuple as an argument
print(arr)
arr1=np.array([6,7,8,9,10]) #list as an argument
print(arr1)
arr2=np.array(arr) # array-like object as an argument
print(arr2)
```
[1 2 3 4 5]
[6 7 8 9 10]
[1 2 3 4 5]

```
# In[19]:
#Dimensions in arrays
#0-D Arrays - 0-D arrays, or Scalars, are the elements in an array.
Each value in an array is a 0-D array.

import numpy as np
arr = np.array(42)
print(arr)
print() #for new line space

#1-D Arrays - An array that has 0-D arrays as its elements is
called uni-dimensional or 1-D array.

import numpy as np
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)
print() #for new line space

#2-D Arrays - An array that has 1-D arrays as its elements is
called a 2-D array.
#These are often used to represent matrix or 2nd order tensors.
```

```
#Create a 2-D array containing two arrays with the values 1,2,3 and
4,5,6:

import numpy as np
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
print() #for new line space

#3-D arrays - An array that has 2-D arrays (matrices) as its
elements is called 3-D array.

import numpy as np
arr3 = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11,
12]]])
print(arr3)
print() #for new line space

#NumPy Arrays provides the ndim attribute that returns an integer
that tells how many dimensions the array has.

import numpy as np
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
42

[1 2 3 4 5]

[[1 2 3]
 [4 5 6]]

[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]

0
1
2
```

3

```
# In[20]:
#We can create an array with any number of dimensions with ndmin
argument

import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

[[[[[1 2 3 4]]]]]
number of dimensions : 5

```
# In[23]:
#Indexing - it is same as accessing an array elements.
#we can access an array element by referring its index number.
#The indexes in NumPy arrays start with 0, meaning that the first
element has index 0, and the second has index 1 etc.

#To get elements from the array
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0]) # to get the first element from the array
print(arr[1]) # to get the second element from the array
print(arr[0] + arr[2]) # to get first and third element and to
print their addition
```

1
2
4

```
# In[25]:
# Accessing the elements from 2-D arrays:
#To access elements from 2-D arrays we can use comma separated
integers representing
#the dimension and the index of the element.

# 2-D arrays are like a table with rows and columns,
#where the row represents the dimension and the index represents
the column.

#To Access the element on the first row, second column:
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])
```

```
print('5th element on 2nd row: ', arr[1, 4]) #Access the element on
the 2nd row, 5th column:
```

2nd element on 1st row: 2
5th element on 2nd row: 10

```
# In[29]:
#Negative Indexing - Use negative indexing to access an array from
the end.

#Print the last element from the 2nd dim:
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])

print('Last element from 2nd dim: ', arr[1, -2]) # to print the
last but one element from the 2nd dim:
```

Last element from 2nd dim: 10
Last but one element from 2nd dim: 9

Slicing:

```
#Slicing arrays -Slicing in Python means take elements from one
given index to another given index.
#We pass slice instead of index like this: [start:end]. We can also
define the step, like this: [start:end:step].
#If we don't pass start its considered 0, If we don't pass end it
considered length of array in that dimension
#If we don't pass step its considered 1

import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5]) #result includes the start index but exclude end
index.
#Output: [2 3 4 5]

arr1 = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr1[4:]) #it gives elements from 4th index to end of the
array.
#Output:[5 6 7]

arr2 = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr2[:4]) #result from beginning to index 4
#Output:[1 2 3 4]

#negative slicing: we use minus operator to refer to an index from
the end.
arr3 = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr3[-3:-1]) # for negative slicing -1 refers last element
#Output:[5 6]

#step value is used to determine the slicing
arr4 = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr4[1:5:2])
#Output:[2 4]

#to return every other or alternate element from the entire array
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[::2])
#Output:[1 3 5 7]

#Slicing 2-D Arrays
#From the second element or dimension, slice elements from index 1
to index 4 (not included):
arr5 = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr5[1, 1:4])
#Output:[7 8 9]
```

```
#From both elements or dimension, return index 1:
arr6 = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr6[0:2, 1])
#Output:[2 7]

#From both elements, slice index 1 to index 4 (not included), this
will return a 2-D array:
arr7 = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr7[0:2, 1:4])
#Output: [[2 3 4]
 [7 8 9]]


#Data Types in Numpy
#By default Python have these data types:

#strings - used to represent text data, the text is given under
quote marks. e.g. "ABCD"
#integer - used to represent integer numbers. e.g. -1, -2, -3
#float - used to represent real numbers. e.g. 1.2, 42.42
#boolean - used to represent True or False.
#complex - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5
+ 2.5j

#Data Types in NumPy
#NumPy has some extra data types, and refer to data types with one
character, like i for integers, u for unsigned integers etc.

#Below is a list of all data types in NumPy and the characters used
to represent them.

#i - integer
#b - boolean
#u - unsigned integer
#f - float
#c - complex float
#m - timedelta
#M - datetime
#O - object
#S - string
#U - unicode string
#V - fixed chunk of memory for other type ( void )

import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr.dtype) #data type of an array object
```

```
#output int32

arr = np.array(['apple', 'banana', 'cherry']) #data type of an
array containing strings
print(arr.dtype)
#output <U6
```

Numpy Random

```
from numpy import random
x = random.rand() #It generates the random number between 0 to 1
print(x)
#0.527467729519793

x1 = random.randint(100) #It generates the random number between 0
to 100
print(x1)
#21

x2=random.randint(100, size=(5)) #It generates 1-D array containing
5 randomly generated elements
print(x2)
#[41 50 72 26 59]

x3 = random.randint(100, size=(3, 5)) #It generates 2-D array with
3 rows and five elements in each row
print(x3)
#Output [[40 75 59 88 91]
 [36 14 49 14 52]
 [39 74 54 30 38]]

#Generate random number from the array
x4 = random.choice([3, 5, 7, 9])
print(x4)
#output 3

#Generate a 2-D array that consists of the values in the array
parameter (3, 5, 7, and 9):
x5 = random.choice([3, 5, 7, 9], size=(3, 5))
print(x5)
#Output
[[7 3 7 5 3]
 [7 5 3 5 3]
 [9 5 9 5 9]]
```

#Data Distribution - List of all possible values
#Random Distribution - is a set of random numbers that follow a certain probability density function i.e,
#probability of all values within the given range.
#We use choice() method of the random module to select random numbers based on the defined or given probabilities.
#The choice() method allows us to specify the probability for each value.

```
from numpy import random
x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(100))
#The sum of all probabilites should be equal to 1.
print(x)
```

#Output
#[3 7 5 5 7 7 7 5 7 7 7 7 7 3 3 7 7 5 5 3 5 7 7 7 7 3 5 7 7 7 7 7 3 5 7 7 7
#5 5 7 7 5 7 5 7 5 7 5 7 7 5 5 5 7 5 5 5 3 7 5 7 5 7 5 5 7 7 7 7 7 3 7 5 7
#7 5 7 7 7 7 5 5 7 3 7 7 7 5 7 7 5 5 7 7 7 7 7 7 7 3]

#Random Permutation - arrangement of elements and it can be done in two ways: shuffle() and permutation()
#Shuffling Arrays - It means changing arrangement of elements in-place. i.e. in the array itself.
#Permutation gives a re-arranged array

#Randomly shuffle elements of following array:

```
from numpy import random
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
random.shuffle(arr)
print(arr)
```

#Output [2 1 4 3 5]

```
print(random.permutation(arr)) #Generate a random permutation of elements of following array:
```

#Output [2 1 4 3 5]

Normal Distribution

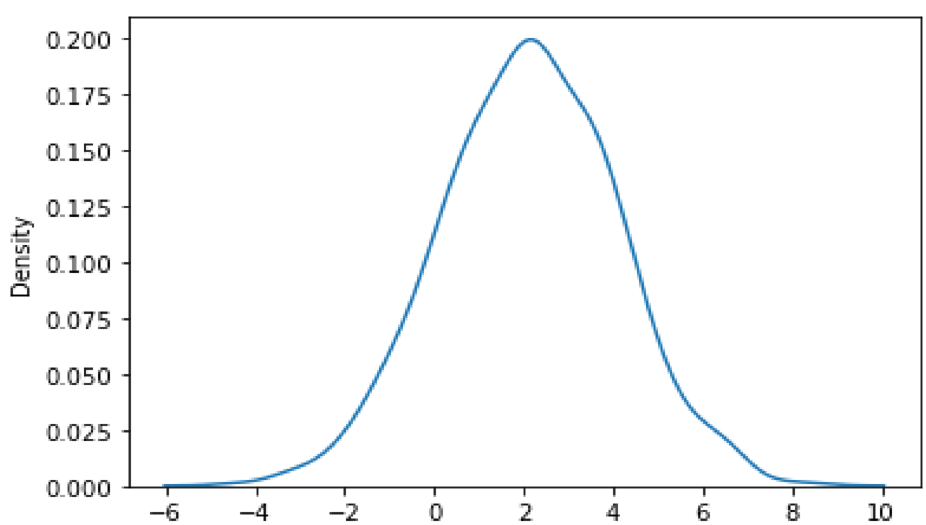
#The Normal Distribution is one of the most important distributions. It is also called the Gaussian Distribution.
It fits the probability distribution of many events, eg. IQ Scores, Heartbeat etc.
#Use the random.normal() method to get a Normal Data Distribution.
#It has three parameters:
#loc - (Mean) where the peak of the bell exists.
#scale - (Standard Deviation) how flat the graph distribution should be.
#size - The shape of the returned array.
#The curve of a Normal Distribution is also known as the Bell Curve because of the bell-shaped curve.

```
from numpy import random
x = random.normal(size=(2, 3))
print(x)
Output: [[-1.55065941 -0.16036268 -1.56157732]
        [ 0.08903827 -0.06865819  1.60877254]]
```

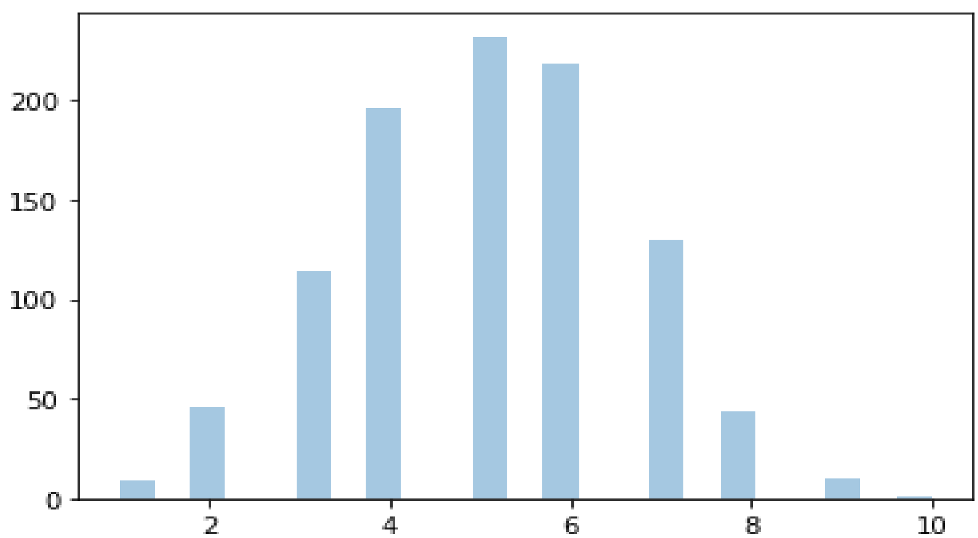
```
from numpy import random
x = random.normal(loc=1, scale=2, size=(2, 3))
print(x)
Output: [[-1.54754665  0.72619477  3.93296342]
        [ 1.48538526  3.53613269  2.50657834]]
```

#To Visualize the Normal Distribution

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot(random.normal(loc=2, scale=2,size=1000), hist=False)
plt.show()
```

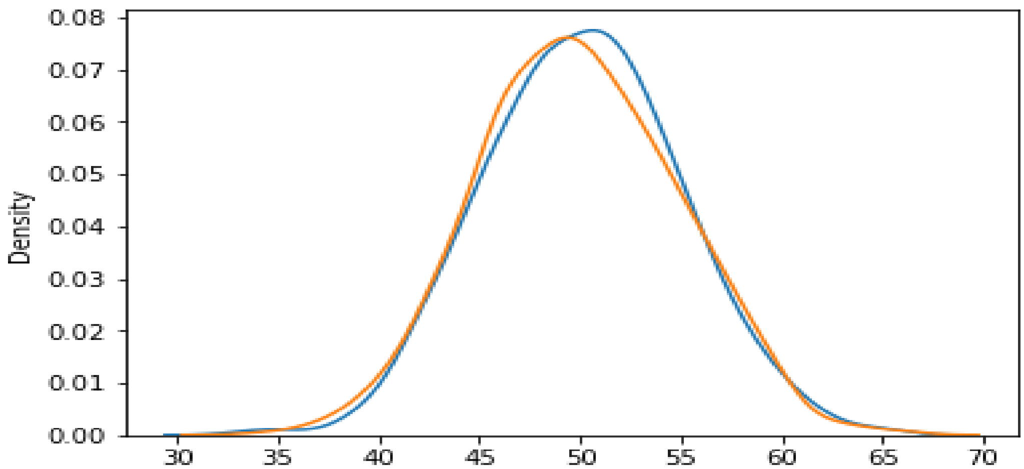


```
#Binomial Distribution is a Discrete Distribution.
#It describes the outcome of binary scenarios, e.g. toss of a coin,
it will either be head or tails.
#It has three parameters:
#n - number of trials.p - probability of occurence of each trial
(e.g. for toss of a coin 0.5 each).
#size - The shape of the returned array.
#Discrete Distribution:The distribution is defined at separate set
of events, e.g. a coin toss's
#result is discrete as it can be only head or tails whereas height
of people is continuous as it can be 170, 170.1, 170.11
#and so on.

#Given 10 trials for coin toss generate 10 data points:
from numpy import random
x = random.binomial(n=10, p=0.5, size=10)
print(x)
Output: [3 7 6 4 6 5 7 4 3 8]


#Visualization of Binomial Distribution
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot(random.binomial(n=10, p=0.5, size=1000), hist=True,
kde=False)
plt.show()
```




```
#Difference Between Normal and Binomial Distribution: The main
difference is that normal distribution
#is continous whereas binomial is discrete, but if there are enough
data points it will be quite similar
#to normal distribution with certain loc and scale.
```

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot(random.normal(loc=50, scale=5, size=1000), hist=False,
label='normal')
sns.distplot(random.binomial(n=100, p=0.5, size=1000), hist=False,
label='binomial')

plt.show()
```



Pandas

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.

We Use Pandas for :

- Pandas allows us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.
- Relevant data is very important in data science.

Using Pandas we can find for

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
- to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called *cleaning* the data.

We use the following command to install Pandas:

```
pip install pandas
Once it is installed we can import using import command.
```

#Pandas
import pandas as pd
mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}
myvar = pd.DataFrame(mydataset)
print(myvar)

Output:

```
       cars  passings
0     BMW         3
1   Volvo         7
2    Ford         2
```

```
import pandas as pd
mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}
myvar = pd.DataFrame(mydataset)
print(myvar)
```

```
Output
       cars  passings
0     BMW         3
1   Volvo         7
2    Ford         2
```

#To check Pandas Version

import pandas as pd
print(pd.__version__)
Output

```
1.4.2
```

#A Pandas Series is like a column in a table.
#It is a one-dimensional array holding data of any type.
#Create a simple Pandas Series from a list:
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a)
print(myvar)

Output

```
0    1
1    7
2    2
```

Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.
This label can be used to access a specified value.

```
print(myvar[0]) #Return the first value of the Series:
1
```

```
#Create Labels - With the index argument, you can name your own lab
els.
#Create you own labels:
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a, index = ["x", "y", "z"])
print(myvar)

Output:
x    1
y    7
z    2
dtype: int64

#When you have created labels, you can access an item by referring
to the label.
print(myvar["y"])

Output:
7

#Key/Value Objects as Series
#You can also use a key/value object, like a dictionary, when creat
ing a Series.
#Create a simple Pandas Series from a dictionary:

import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories)
print(myvar)

#To select only some of the items in the dictionary, use the index
argument and specify only the items you want to include in the Seri
es.
#Create a Series using only data from "day1" and "day2":

import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
```

```
myvar = pd.Series(calories, index = ["day1", "day2"])
print(myvar)
```

Output

```
day1    420
day2    380
day3    390
dtype: int64

day1    420
day2    380
dtype: int64
```

DataFrames

```
#DataFrames - Data sets in Pandas are usually multi-dimensional tab
les, called DataFrames.
#Series is like a column, a DataFrame is the whole table.
#Create a DataFrame from two Series:

import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
myvar = pd.DataFrame(data)
print(myvar)
```

Output

```
calories  duration
0       420        50
1       380        40
2       390        45
```

```
#Pandas use the loc attribute to return one or more specified row(s
)
#refer to the row index:
print(myvar.loc[0])
```

Output

```
calories    420
duration     50
Name: 0, dtype: int64
```

```
#use a list of indexes:
print(myvar.loc[[0, 1]]) #Return row 0 and 1:
```

Output

```
   calories  duration
0       420        50
1       380        40
```

#Named Indexes - With the index argument, you can name your own indexes.
#Add a list of names to give each row a name:

```
#import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
print(df)
```

Output

```
calories  duration
day1       420        50
day2       380        40
day3       390        45
```

#Write an example to add and display totally 10 days and also give some values for calories and duration.

Output

#Locate Named Indexes -Use the named index in the loc attribute to return the specified row(s).

```
#refer to the named index:
print(df.loc["day2"]) #Return "day2":
```

Output

```
calories    380
duration     40
Name: day2, dtype: int64
```

#Load Files Into a DataFrame
#If your data sets are stored in a file, Pandas can load them into a DataFrame.
#Load a comma separated file (CSV file) into a DataFrame:
#Create a DataFrame with the .CSV format and give name Book1.csv

```
Duration  Pulse  MaxPulse  Calories
  60    110       130     409.1
  60    110       150     400.5
  60    117       125     396.2
  45    114       132     205.6
  45    115       136     356.4
  25    112       134     395.4
  72    102       139     425.3
  58    106       138     412.9
  65    105       137     320.6
  65    108       136     321.3
  57    112       141     356.1
  96    113       140     389.4
  67    114       142     389.4
  69    115       139     417.5
  63    115       150     457.9
  68    116       160     369.4
```

```
import pandas as pd
df = pd.read_csv('Book1.csv')
print(df)
```

Output

```
       Duration  Pulse  MaxPulse  Calories
0         60    110       130     409.1
1         60    110       150     400.5
2         60    117       125     396.2
3         45    114       132     205.6
4         45    115       136     356.4
5         25    112       134     395.4
6         72    102       139     425.3
7         58    106       138     412.9
8         65    105       137     320.6
9         65    108       136     321.3
10        57    112       141     356.1
11        96    113       140     389.4
12        67    114       142     389.4
13        69    115       139     417.5
14        63    115       150     457.9
15        68    116       160     369.4
```

#A simple way to store big data sets is to use CSV files (comma separated files).
#CSV files contains plain text and is a well know format that can be read by everyone including Pandas.

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df)
```

Output

```
Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2          60    103       135     340.0
3          45    109       175     282.4
4          45    117       148     406.0
..        ...    ...       ...       ...
164        60    105       140     290.8
165        60    110       145     300.0
166        60    115       145     310.2
167        75    120       150     320.4
168        75    125       150     330.4

[169 rows x 4 columns]
```

```
#Load the CSV into a DataFrame:
#use to_string() to print the entire DataFrame.
#If you have a large DataFrame with many rows, Pandas will only ret
urn the first 5 rows, and the last 5 rows:

import pandas as pd
df = pd.read_csv('data.csv')
print(df.to_string())
```

Output

```
#Print the DataFrame without the to_string() method:

import pandas as pd
df = pd.read_csv('data.csv')
print(df)
```

Output

```
Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2          60    103       135     340.0
3          45    109       175     282.4
4          45    117       148     406.0
..        ...    ...       ...       ...
164        60    105       140     290.8
165        60    110       145     300.0
166        60    115       145     310.2
```

```
167          75    120         150      320.4
168          75    125         150      330.4

[169 rows x 4 columns]
```

```
#max_rows - The number of rows returned is defined in Pandas option settings.
#You can check your system's maximum rows with the pd.options.display.max_rows statement.
#Check the number of maximum returned rows:

import pandas as pd
print(pd.options.display.max_rows)

#In my system the number is 60, which means that if the DataFrame contains more than 60 rows,
#the print(df) statement will return only the headers and the first and last 5 rows.
#You can change the maximum rows number with the same statement.
#Increase the maximum number of rows to display the entire DataFrame:

import pandas as pd
pd.options.display.max_rows = 9999
df = pd.read_csv('data.csv')
print(df)

print(pd.options.display.max_rows) #It gives maximum number of rows .. now it will give 9999
```

#Pandas - Data Correlations

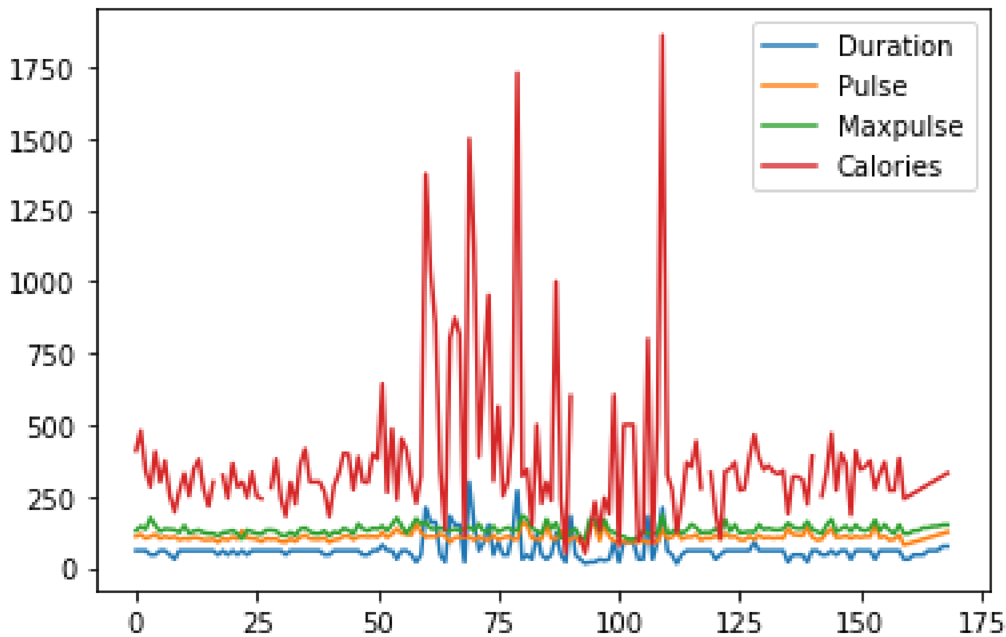
```
#Finding Relationships
#A great aspect of the Pandas module is the corr() method.
#The corr() method calculates the relationship between each column in your data set.

import pandas as pd
df = pd.read_csv('data.csv')
df.corr() #Show the relationship between the columns:
```

Output

	Duration	Pulse	Maxpulse	Calories
Duration	1.000000	-0.155408	0.009403	0.922717
Pulse	-0.155408	1.000000	0.786535	0.025121
Maxpulse	0.009403	0.786535	1.000000	0.203813
Calories	0.922717	0.025121	0.203813	1.000000

#Pandas - Plotting

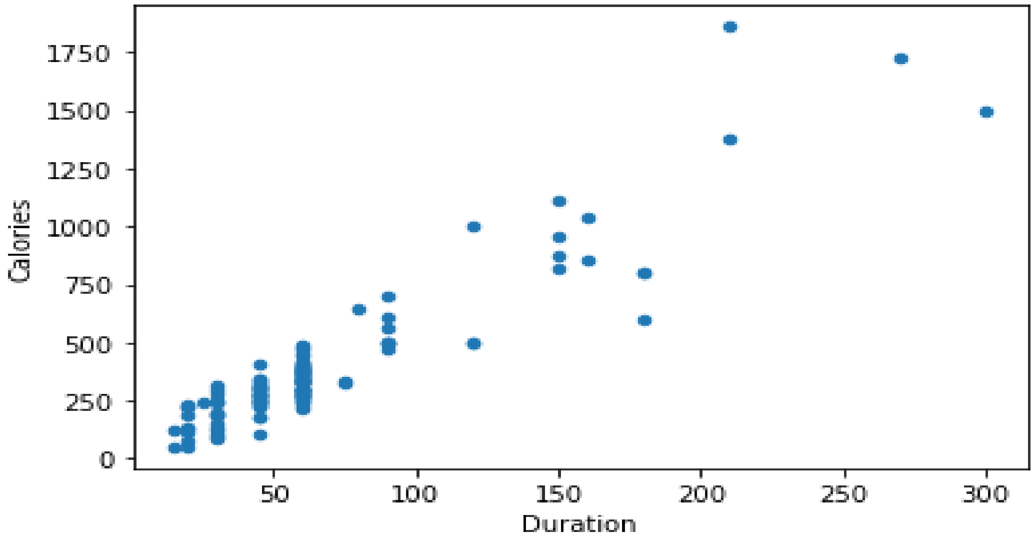
```
#Pandas uses the plot() method to create diagrams.
#We can use Pyplot, a submodule of the Matplotlib library to visual
ize the diagram on the screen.
#Import pyplot from Matplotlib and visualize our DataFrame:

import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('data.csv')
df.plot()
plt.show()
```



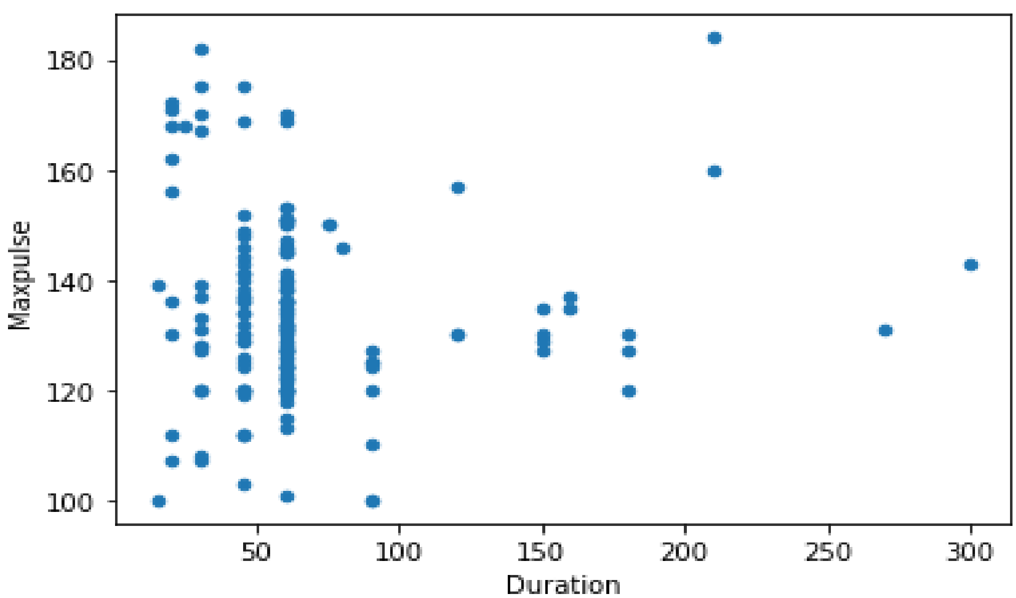
#Scatter Plot -

```
#Specify that you want a scatter plot with the kind argument:
#kind = 'scatter'
#A scatter plot needs an x- and a y-axis.
#In the example below we will use "Duration" for the x-axis and "Ca
lories" for the y-axis.
#Include the x and y arguments like this:
#x = 'Duration', y = 'Calories'
#In the previous example, we learned that the correlation between "
Duration" and "Calories" was 0.922721,
#and we conluded with the fact that higher duration means more calo
ries burned. So good relation between those two attributes.

import pandas as pd
import matplotlib.pyplot as plt
```

```
df = pd.read_csv('data.csv')
df.plot(kind = 'scatter', x = 'Duration', y = 'Calories')
plt.show()
```




```
#Let's create another scatterplot, where there is a bad relationshi
p between the columns,
#like "Duration" and "Maxpulse", with the correlation 0.009403:
#A scatterplot where there are no relationship between the columns:

import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('data.csv')
df.plot(kind = 'scatter', x = 'Duration', y = 'Maxpulse')
plt.show()
```



#Histogram - Use the kind argument to specify that you want a histogram:

```
#kind = 'hist'
#A histogram needs only one column. A histogram shows us the freque
ncy of each interval,
#e.g. how many workouts lasted between 50 and 60 minutes?
#In the example below we will use the "Duration" column to create t
he histogram:

df["Duration"].plot(kind = 'hist')
```