Introduction to python

Array in Python

- Collection of same types unlike list
- Don’t have fix size in python can shrink and expand
- Flexible to work with
- Array in Python can be created by importing array module.
- array(data_type, value_list) is used to create an array with data type and value list specified in its arguments.
- From array import *
- Vals=array (‘i’,[1,2,3])
- Print (vals)
- Print (vals.buffer_info())
- ‘buffer_info () returns address and size of array

Array in Python

- Collection of same types unlike list
- Don’t have fix size in python can shrink and expand
- Flexible to work with

```
from array import *
```

```
arrayIdentifierName = array(typecode, [Initializers]
```

```
my_array = array('i',[1,2,3,4])
```

- ‘b’ -> Represents signed integer of size 1 byte
- ‘B’ -> Represents unsigned integer of size 1 byte
- ‘c’ -> Represents character of size 1 byte
- ‘u’ -> Represents unicode character of size 2 bytes
- ‘h’ -> Represents signed integer of size 2 bytes
- ‘H’ -> Represents unsigned integer of size 2 bytes
- ‘i’ -> Represents signed integer of size 2 bytes
- ‘I’ -> Represents unsigned integer of size 2 bytes
- ‘w’ -> Represents unicode character of size 4 bytes
- ‘l’ -> Represents signed integer of size 4 bytes
- ‘L’ -> Represents unsigned integer of size 4 bytes
- ‘f’ -> Represents floating point of size 4 bytes
- ‘d’ -> Represents floating point of size 8 bytes

```
from array import *
temps =  array ('i',[1,2,3])
#for i in temps:
#                print (temps)

#for i in range (3):
#                print (temps [i])

for i in range (len (temps)):
     print (temps [i])

##temps.append (4)
##print (temps)
##temps.insert (0,0)
##print (temps)
##temps.remove (0)
##print (temps)
##temps.pop ()
#print (temps)
print (temps.index (1))
temps.reverse ()
print (temps.count (2))
print (temps.buffer_info())
```

Slicing Python Arrays

```
import array as arr

numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
numbers_array = arr.array('i', numbers_list)

print(numbers_array[2:5]) # 3rd to 5th
print(numbers_array[:-5]) # beginning to 4th
print(numbers_array[5:])  # 6th to end
print(numbers_array[:])   # beginning to end
```

Output

```
array('i', [62, 5, 42])
array('i', [2, 5, 62])
array('i', [52, 48, 5])
array('i', [2, 5, 62, 5, 42, 52, 48, 5])
```

Python Arrays Concatenation

We can also concatenate two arrays using `+` operator.

```
import array as arr

odd = arr.array('i', [1, 3, 5])
even = arr.array('i', [2, 4, 6])

numbers = arr.array('i')   # creating empty array of integer
numbers = odd + even

print(numbers)
```

Output

```
array('i', [1, 3, 5, 2, 4, 6])
```

Python Lists Vs Arrays

In Python, we can treat lists as arrays. However, we cannot constrain the type of elements stored in a list. For example:

```
# elements of different types
a = [1, 3.5, "Hello"]
```

If you create arrays using the `array` module, all elements of the array must be of the same numeric type.

```
import array as arr
# Error
a = arr.array('d', [1, 3.5, "Hello"])
```

Output

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
    a = arr.array('d', [1, 3.5, "Hello"])
TypeError: must be real number, not str
```

When to use arrays?

Lists are much more flexible than arrays. They can store elements of different data types including strings. And, if you need to do mathematical computation on arrays and matrices, you are much better off using something like NumPy.

Introduction to Numpy

Numpy

- Numpy is the core library for scientific computing in Python
- It provides a high-performance multidimensional array object, and tools for working with these arrays.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- At the core, numpy provides the excellent *n* dimensional array objects.
- Numpy stands for Numerical Python

Why do we use Numpy?

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called *n*darray, it provides a lot of supporting functions that make working with *n*darray very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.

Why Numpy is faster than Lists?

- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- This behavior is called locality of reference in computer science.
- This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.
- The source code for NumPy is located at this github repository https://github.com/numpy/numpy

Creating Numpy Array

```
import numpy as np

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

print(arr)
```

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

```
print(np.__version__)
```

```
1.21.2
```

```
print(type(arr))
```

```
<class 'numpy.ndarray'>
```

NumPy is used to work with arrays. The array object in NumPy is called `ndarray`.
We can create a NumPy `ndarray` object by using the `array()` function.

Creating Numpy Array

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])

print(arr)
```

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

```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```

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.

Example

Create a 0-D array with value 42

```
import numpy as np

arr = np.array(42)

print(arr)
```

Dimensions in Arrays

1-D Arrays

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

These are the most common and basic arrays.

Example

Create a 1-D array containing the values 1,2,3,4,5:

```
import numpy as np

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

print(arr)
```

Dimensions in Arrays

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.

NumPy has a whole sub module dedicated towards matrix operations called `numpy.mat`

Example

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

```
import numpy as np

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

print(arr)
```

Dimensions in Arrays

3-D arrays

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

These are often used to represent a 3rd order tensor.

Example

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

```
import numpy as np

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

print(arr)
```

Dimensions in Arrays

Check Number of Dimensions?

NumPy Arrays provides the `ndim` attribute that returns an integer that tells us how many dimensions the array have.

Example

Check how many dimensions the arrays have:

```
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)
```

Dimensions in Arrays

Higher Dimensional Arrays

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using the `ndmin` argument.

Example

Create an array with 5 dimensions and verify that it has 5 dimensions:

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('number of dimensions :', arr.ndim)
```

NUMPY Array Indexing: Accessing the array elements

Get the first element from the following array:

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[0])
```

Access the third element of the second array of the first array:

```
import numpy as np

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

print(arr[0, 1, 2])
```

Access the 2nd element on 1st dim:

```
import numpy as np

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

print('2nd element on 1st dim: ', arr[0, 1])
```

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])
```

NUMPY Array Slicing

- Slicing in python means taking 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 its considered length of array in that dimension
- If we don't pass step its considered 1

NUMPY Array Slicing

```
#Slicing Elements from index 1 to index less than 5 (means index 4)
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```

```
[2 3 4 5]
```

Slice elements from index 4 to the end of the array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[4:])
```

Slice elements from the beginning to index 4 (not included):

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[:4])
```

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[-3:-1])
```

```
[5 6]
```

NUMPY Array Slicing

```
#Use the step value to determine the step of the slicing:
#Return every other element from index 1 to index 5: (5  not included)
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5:2])
```

```
[2 4]
```

Return every other element from the entire array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[::2])
```

NUMPY Array Slicing

Create an array with data type string:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr)
print(arr.dtype)
```

Create an array with data type 4 bytes integer:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')

print(arr)
print(arr.dtype)
```

NUMPY Array Slicing

What if a Value Can Not Be Converted?

If a type is given in which elements can't be casted then NumPy will raise a ValueError.

ValueError: In Python ValueError is raised when the type of passed argument to a function is unexpected/incorrect.

Example

A non integer string like 'a' can not be converted to integer (will raise an error):

```
import numpy as np

arr = np.array(['a', '2', '3'], dtype='i')
```

NUMPY Array Slicing

Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the `astype()` method.

The `astype()` function creates a copy of the array, and allows you to specify the data type as a parameter.

The data type can be specified using a string, like `'f'` for float, `'i'` for integer etc. or you can use the data type directly like `float` for float and `int` for integer.

Example

Change data type from float to integer by using `'i'` as parameter value:

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

```
i = integer
b = boolean
u = unsigned integer
f = float
c = complex float
m = timedelta
M = datatime
O = object
S = string
```

NUMPY Array Shape

Get the Shape of an Array

NumPy arrays have an attribute called `shape` that returns a tuple with each index having the number of corresponding elements.

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(arr.shape)
```

```
(2, 4)
```

NUMPY Array Reshaping

Reshaping arrays

Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

```
import numpy as np

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

newarr = arr.reshape(4, 3)

print(newarr)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

NUMPY Array Iterating

```
import numpy as np

arr = np.array([1, 2, 3])

for x in arr:
  print(x)
```

```
import numpy as np

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

for x in arr:
  print(x)
```

```
import numpy as np

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

for x in arr:
  print(x)
```

NUMPY Array Iterating

Iterate on each scalar element of the 2-D array:

```
import numpy as np

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

for x in arr:
  for y in x:
    print(y)
```

```
import numpy as np

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

for x in arr:
  for y in x:
    for z in y:
      print(z)
```

Iterating Arrays Using nditer()

The function `nditer()` is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, lets go through it with examples.

Iterating on Each Scalar Element

In basic `for` loops, iterating through each scalar of an array we need to use *n* `for` loops which can be difficult to write for arrays with very high dimensionality.

Example

Iterate through the following 3-D array:

```
import numpy as np

arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

for x in np.nditer(arr):
  print(x)
```

NUMPY Array Splitting

Splitting is reverse operation of Joining.

Joining merges multiple arrays into one and Splitting breaks one array into multiple.

We use `array_split()` for splitting arrays, we pass it the array we want to split and the number of splits.

```
import numpy as np

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

newarr = np.array(np.array_split(arr, 3))
```

```
print (newarr)
```

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

```
print (newarr.shape)
```

```
(3, 2)
```

NUMPY Array Splitting

```
#spliting 2D array into 3 2D arrays
import numpy as np

arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
print (arr.shape)
newarr = np.array(np.array_split(arr, 3))

print(newarr)
print (newarr.shape)
```

```
(6, 2)
[[[ 1  2]
  [ 3  4]]

 [[ 5  6]
  [ 7  8]]

 [[ 9 10]
  [11 12]]]
(3, 2, 2)
```

Train and Test Data Split

```
# split train and test data
from numpy import array
# define array
data = array([
[11, 22, 33],
[44, 55, 66],
[77, 88, 99]])
# separate data
split = 2
train,test = data[:split,:],data[split:,:]
print(train)
print(test)
```

data [initial row:last row - 1,all columns in all row] , data [last row: all columns]
#train,test = data[0:2,0:3],data[2:3,0:3]

- # split train and test data
- from numpy import array
- # define array
- data = array([
- [11, 22, 33],
- [44, 55, 66],
- [77, 88, 99]])
- # separate data
- split = 2
- train,test = data[:split,:],data[split:,:]
- print(train)
- print(test)

Introduction to Pandas

Agenda

- What is Pandas?
- Data Frame
- Create DataFrame
- Import CSV and Excel Data
- Selecting and Slicing Data
- Filtering Data
- Dealing With Missing and Duplicated Data
- Joining Data
- Aggregating Data

What is Pandas

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.
- In order to “get” Pandas you would need to install it. Python 2.7 and above as a pre-requirement for installation.
- Pip install pandas
- Import pandas as pd

Why do we use Pandas?

- 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.
- Pandas gives you answers about the data. Like:
  - Is there a correlation between two or more columns?
  - What is average value?
  - Max value?
  - Min value?
- Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.
- The source code for Pandas is located at this github repository https://github.com/pandas-dev/pandas

Data Frame

- Panda is built on the Numpy package and its key data structure is called the DataFrame.

- DataFrames allow you to store and manipulate tabular data in rows of observations and columns of variables.


```
import pandas as pd

mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}

myvar = pd.DataFrame(mydataset)

print(myvar)
```

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

```
print(myvar.shape)
```

```
(3, 2)
```

```
print(pd.__version__)
```

```
1.3.3
```

Create Data Frame

```
dict = {
        "country": ["Brazil", "Russia", "India"],
        "capital": ["Brasilia", "Moscow", "New Dehli"],
        "area": [8.516, 17.10, 3.286],
        "population": [200.4, 143.5, 1252]
        }
df = pd.DataFrame(dict)
print(df)
```

```
  country    capital    area  population
0  Brazil   Brasilia   8.516       200.4
1  Russia     Moscow  17.100       143.5
2   India  New Dehli   3.286      1252.0
```

```
#Storing values such as Date, Time Worked, and Money Earned in a DataFrame
# Creating a data frame df.
df = pd.DataFrame({
'Date':['11/05/19', '12/05/19', '19/05/19'],
'Time Worked': [3, 3, 4],
'Money Earned': [33.94, 33.94, 46.0]
})
df.head()
```

	Date	Time Worked	Money Earned
0	11/05/19	3	33.94
1	12/05/19	3	33.94
2	19/05/19	4	46.00

Pandas Series, Labels, and Dataframes

- Pandas Series: It 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)
```

```
0    1
1    7
2    2
dtype: int64
```

Pandas Series, Labels, and Dataframes

- Pandas Label: If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc. (as shown in previous example). This label can be used to access a specified value.

- You can also create your own labels

```
#Create a simple Pandas Series and access specific element:
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a)
print(myvar[1])
```

```
7
```

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a, index = ["x", "y", "z"])

print(myvar)
```

```
x    1
y    7
z    2
dtype: int64
```

```
print(myvar["y"])
```

```
7
```

Pandas Series, Labels, and Dataframes

- **Key/Value Objects as Series**
- You can also use a key/value object, like a dictionary, when creating a Series.

- 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 Series.

```
import pandas as pd

calories = {"day1": 420, "day2": 380, "day3": 390}

myvar = pd.Series(calories)

print(myvar)
```

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

```
#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)
```

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

Pandas Series, Labels, and Dataframes

- **Data Frames**
- Data sets in Pandas are usually multi-dimensional tables, 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)
```

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

Pandas Series, Labels, and Dataframes

- **Locate Row of a DataFrame**
- As you can see from the result above, the DataFrame is like a table with rows and columns.
- Pandas use the *loc* attribute to return one or more specified row(s)
- Note: When using [], the result is a Pandas DataFrame.

```
#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)
```

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

```
#Return row 0:
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

#load data into a DataFrame object:
df = pd.DataFrame(data)

print(df.loc[0])
```

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

Example

Return row 0 and 1:

```
#use a list of indexes:
print(df.loc[[0, 1]])
```

Pandas DataFrame: Working with Named Index

- With the *index* argument, you can name your own indexes.

- Use the *named index* in the *loc* attribute to return the specified row(s).

```
#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)
```

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

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

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

```
#Load an excel file (xlsx file) into a DataFrame:
import pandas as pd
df = pd.read_excel('data.xlsx')
print(df)
```

Read CSV Files

- 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.
- In our examples we will be using a CSV file called **'data.csv'**.

```
import pandas as pd
df = pd.read_csv('data.csv')
#By default, when you print a DataFrame, you will only get the first 5 rows, and the last 5 rows:
#print(df)
#use to_string() to print the entire DataFrame.
#print(df.to_string())
#df.head() read first five rows. You can pass any number in head () function to read rows.
print (df.head())
```

```
   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
```

Panadas - Analyzing DataFrames

- **Viewing the Data**
- One of the most used method for getting a quick overview of the DataFrame, is the head() method.
- The head() method returns the headers and a specified number of rows, starting from the top.

```
import pandas as pd
df = pd.read_csv('data.csv')
#df.head() read first five (05) rows. You can pass any number in head () function to read rows.
#print (df.head())
print (df.head(10))
```

```
   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
5        60    102       127     300.0
6        60    110       136     374.0
7        45    104       134     253.3
8        30    109       133     195.1
9        60     98       124     269.0
```

Panadas - Analyzing DataFrames

There is also a `tail()` method for viewing the *last* rows of the DataFrame.

The `tail()` method returns the headers and a specified number of rows, starting from the bottom.

Example

Print the last 5 rows of the DataFrame:

```
print(df.tail())
```

Panadas - Analyzing DataFrames

- **Info about the Data**
- The DataFrames object has a method called info(), that gives you more information about the data set.

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 169 entries, 0 to 168
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  169 non-null    int64
 1   Pulse     169 non-null    int64
 2   Maxpulse  169 non-null    int64
 3   Calories  164 non-null    float64
dtypes: float64(1), int64(3)
memory usage: 5.3 KB
None
```

Result Explained

The result tells us there are 169 rows and 4 colu

```
RangeIndex: 169 entries, 0 to 168
Data columns (total 4 columns):
```

And the name of each column, with the data type

```
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  169 non-null    int64
 1   Pulse     169 non-null    int64
 2   Maxpulse  169 non-null    int64
 3   Calories  164 non-null    float64
```

Panadas - Analyzing DataFrames

Null Values

The `info()` method also tells us how many Non-Null values there are present in each column, and in our data set it seems like there are 164 of 169 Non-Null values in the "Calories" column.

Which means that there are 5 rows with no value at all, in the "Calories" column, for whatever reason.

Empty values, or Null values, can be bad when analyzing data, and you should consider removing rows with empty values. This is a step towards what is called *cleaning data*, and you will learn more about that in the next chapters.

Panadas – Data Cleaning

- Data cleaning means fixing bad data in your data set.
- Bad data could be:
  - Empty cells
  - Data in wrong format
  - Wrong data
  - Duplicates
- The data set contains some empty cells ("Date" in row 22, and "Calories" in row 18 and 28).
- The data set contains wrong format ("Date" in row 26).
- The data set contains wrong data ("Duration" in row 7).
- The data set contains duplicates (row 11 and 12).

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3
14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0
16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
18	45	'2020/12/18'	90	112	NaN
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
22	45	NaN	100	119	282.0
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	2020/12/26	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
28	60	'2020/12/28'	103	132	NaN
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

Panadas – Data Cleaning

- Reading data
- We can see many empty cells
- We can notice wrong format data
- We can notice wrong data
- We can notice duplicate rows

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

```
    Duration          Date  Pulse  Maxpulse  Calories
0         60  '2020/12/01'    110       130     409.1
1         60  '2020/12/02'    117       145     479.0
2         60  '2020/12/03'    103       135     340.0
3         45  '2020/12/04'    109       175     282.4
4         45  '2020/12/05'    117       148     406.0
5         60  '2020/12/06'    102       127     300.0
6         60  '2020/12/07'    110       136     374.0
7        450  '2020/12/08'    104       134     253.3
8         30  '2020/12/09'    109       133     195.1
9         60  '2020/12/10'     98       124     269.0
10        60  '2020/12/11'    103       147     329.3
11        60  '2020/12/12'    100       120     250.7
12        60  '2020/12/12'    100       120     250.7
13        60  '2020/12/13'    106       128     345.3
14        60  '2020/12/14'    104       132     379.3
15        60  '2020/12/15'     98       123     275.0
16        60  '2020/12/16'     98       120     215.2
17        60  '2020/12/17'    100       120     300.0
18        45  '2020/12/18'     90       112       NaN
19        60  '2020/12/19'    103       123     323.0
20        45  '2020/12/20'     97       125     243.0
21        60  '2020/12/21'    108       131     364.2
22        45           NaN    100       119     282.0
23        60  '2020/12/23'    130       101     300.0
24        45  '2020/12/24'    105       132     246.0
25        60  '2020/12/25'    102       126     334.5
26        60      20201226    100       120     250.0
27        60  '2020/12/27'     92       118     241.0
28        60  '2020/12/28'    103       132       NaN
29        60  '2020/12/29'    100       132     280.0
30        60  '2020/12/30'    102       129     380.3
31        60  '2020/12/31'     92       115     243.0
```

Panadas – Data Cleaning

- Empty Cells
- Empty cells can potentially give you a wrong result when you analyze data.
- Remove Rows
- One way to deal with empty cells is to remove rows that contain empty cells.
- This is usually OK, since data sets can be very big, and removing a few rows will not have a big impact on the result.

```
import pandas as pd
df = pd.read_csv('dirtydata.csv')
new_df = df.dropna()
print(new_df.to_string())
```

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3
14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0
16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	20201226	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

Panadas – Data Cleaning

- We use *dropna()* method to remove the empty rows.
- By default, the *dropna()* method returns a new DataFrame, and will not change the original.
- If you want to change the original DataFrame, use the *inplace = True* argument.
- Now, the *dropna(inplace = True)* will NOT return a new DataFrame, but it will remove all rows contain NULL values from the original DataFrame.

```
import pandas as pd
df = pd.read_csv('dirtydata.csv')
df.dropna(inplace = True)
print(df.to_string())
```

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3
14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0
16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	20201226	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

Missing Values

- First of all, we need to check whether we have null values in our dataset or not.
- df.isnull()
- # Returns a boolean matrix, if the value is NaN then True otherwise False
- df.isnull().sum()
- #remove all missing data
- data_frame.dropna()
- Arguments for dropna():
  - axis — We can specify axis=0 if we want to remove the rows and axis=1 if we want to remove the columns.
  - how — If we specify how = 'all' then the rows and columns will only be dropped if all the values are NaN.By default how is set to 'any'.
  - thresh — It determines the threshold value so if we specify thresh=5 then the rows having less than 5 real values will be dropped.
  - subset —If we have 4 columns A,B,C and D then if we specify subset=['C'] then only the rows that have their C value as NaN will be removed.
  - inplace — By default no changes will be made to your dataframe. So if you want these changes to reflect onto your dataframe then you need to use inplace = True.

Panadas – Data Cleaning

- Replace Empty Value
- Another way of dealing with empty cells is to insert a new value instead.
- This way you do not have to delete entire rows just because of some empty cells.
- The *fillna()* method allows us to replace empty cells with a value:

```
: #Replace NULL values with the number 130:
import pandas as pd
df = pd.read_csv('dirtydata.csv')
df.fillna(130, inplace = True)
#print(df.info())
print(df.to_string())
```

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3
14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0
16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
18	45	'2020/12/18'	90	112	130.0
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
22	45	130	100	119	282.0
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	20201226	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
28	60	'2020/12/28'	103	132	130.0
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

Panadas – Data Cleaning

- Replace only specified column
- The previous example replaces all empty cells in the whole Data Frame.
- To only replace empty values for one column, specify the column name for the DataFrame:

```
import pandas as pd

df = pd.read_csv('data.csv')

df["Calories"].fillna(130, inplace = True)
```

Panadas – Data Cleaning

- Replace using mean, median and mode

- A common way to replace empty cells, is to calculate the mean, median or mode value of the column.

- Pandas uses the *mean()*, *median()*, and *mode()* methods to calculate the respective values for a specified column:

- *Mean* = the average value (the sum of all values divided by number of values).

- *Median* = the value in the middle, after you have sorted all values ascending.

- *Mode* = the value that appears most frequently.

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].mean()

df["Calories"].fillna(x, inplace = True)
```

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].median()

df["Calories"].fillna(x, inplace = True)
```

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].mode()[0]

df["Calories"].fillna(x, inplace = True)
```

Panadas – Data Cleaning

- Cleaning data of wrong format
- Cells with data of wrong format can make it difficult, or even impossible, to analyze data.
- To fix it, you have two options: remove the rows, or convert all cells in the columns into the same format.
- In our Data Frame, we have two cells with the wrong format. Check out row 22 and 26, the 'Date' column should be a string that represents a date:
- To convert all cells in the 'Date' column into dates, Pandas has a to_*datetime()* method.
- As you can see from the result, the date in row 26 was fixed, but the empty date in row 22 got a NaT (Not a Time) value, in other words an empty value.
- One way to deal with empty values is simply removing the entire row.

- `df.dropna(subset=['Date'], inplace = True)`

```
import pandas as pd
df = pd.read_csv('dirtydata.csv')
df['Date'] = pd.to_datetime(df['Date'])
print(df.to_string())
```

	Duration	Date	Pulse	Maxpulse	Calories
0	60	2020-12-01	110	130	409.1
1	60	2020-12-02	117	145	479.0
2	60	2020-12-03	103	135	340.0
3	45	2020-12-04	109	175	282.4
4	45	2020-12-05	117	148	406.0
5	60	2020-12-06	102	127	300.0
6	60	2020-12-07	110	136	374.0
7	450	2020-12-08	104	134	253.3
8	30	2020-12-09	109	133	195.1
9	60	2020-12-10	98	124	269.0
10	60	2020-12-11	103	147	329.3
11	60	2020-12-12	100	120	250.7
12	60	2020-12-12	100	120	250.7
13	60	2020-12-13	106	128	345.3
14	60	2020-12-14	104	132	379.3
15	60	2020-12-15	98	123	275.0
16	60	2020-12-16	98	120	215.2
17	60	2020-12-17	100	120	300.0
18	45	2020-12-18	90	112	NaN
19	60	2020-12-19	103	123	323.0
20	45	2020-12-20	97	125	243.0
21	60	2020-12-21	108	131	364.2
22	45	NaT	100	119	282.0
23	60	2020-12-23	130	101	300.0
24	45	2020-12-24	105	132	246.0
25	60	2020-12-25	102	126	334.5
26	60	2020-12-26	100	120	250.0
27	60	2020-12-27	92	118	241.0
28	60	2020-12-28	103	132	NaN
29	60	2020-12-29	100	132	280.0
30	60	2020-12-30	102	129	380.3
31	60	2020-12-31	92	115	243.0

Panadas – Data Cleaning

• Wrong Data

- "Wrong data" does not have to be "empty cells" or "wrong format", it can just be wrong, like if someone registered "199" instead of "1.99".
- Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be.
- If you take a look at our data set, you can see that in row 7, the duration is 450, but for all the other rows the duration is between 30 and 60.
- It doesn't have to be wrong, but taking in consideration that this is the data set of someone's workout sessions, we conclude with the fact that this person did not work out in 450 minutes.
- How can we fix wrong values, like the one for "Duration" in row 7?

```
import pandas as pd
df = pd.read_csv('dirtydata.csv')
df['Date'] = pd.to_datetime(df['Date'])
print(df.to_string())
```

	Duration	Date	Pulse	Maxpulse	Calories
0	60	2020-12-01	110	130	409.1
1	60	2020-12-02	117	145	479.0
2	60	2020-12-03	103	135	340.0
3	45	2020-12-04	109	175	282.4
4	45	2020-12-05	117	148	406.0
5	60	2020-12-06	102	127	300.0
6	60	2020-12-07	110	136	374.0
7	450	2020-12-08	104	134	253.3
8	30	2020-12-09	109	133	195.1
9	60	2020-12-10	98	124	269.0
10	60	2020-12-11	103	147	329.3
11	60	2020-12-12	100	120	250.7
12	60	2020-12-12	100	120	250.7
13	60	2020-12-13	106	128	345.3
14	60	2020-12-14	104	132	379.3
15	60	2020-12-15	98	123	275.0
16	60	2020-12-16	98	120	215.2
17	60	2020-12-17	100	120	300.0
18	45	2020-12-18	90	112	NaN
19	60	2020-12-19	103	123	323.0
20	45	2020-12-20	97	125	243.0
21	60	2020-12-21	108	131	364.2
22	45	NaT	100	119	282.0
23	60	2020-12-23	130	101	300.0
24	45	2020-12-24	105	132	246.0
25	60	2020-12-25	102	126	334.5
26	60	2020-12-26	100	120	250.0
27	60	2020-12-27	92	118	241.0
28	60	2020-12-28	103	132	NaN
29	60	2020-12-29	100	132	280.0
30	60	2020-12-30	102	129	380.3
31	60	2020-12-31	92	115	243.0

Panadas – Data Cleaning

- Correcting the Wrong Data by replacing the value
- One way to fix wrong values is to replace them with something else.
- In our example, it is most likely a typo, and the value should be "45" instead of "450", and we could just insert "45" in row 7:
- Hence, to set "Duration" = 45 in row 7, we need to write:
- `df.loc[7, 'Duration'] = 45`

```
df.loc[7,'Duration']=45
df.head (10)
```

	Duration	Date	Pulse	Maxpulse	Calories
0	60	2020-12-01	110	130	409.1
1	60	2020-12-02	117	145	479.0
2	60	2020-12-03	103	135	340.0
3	45	2020-12-04	109	175	282.4
4	45	2020-12-05	117	148	406.0
5	60	2020-12-06	102	127	300.0
6	60	2020-12-07	110	136	374.0
7	45	2020-12-08	104	134	253.3
8	30	2020-12-09	109	133	195.1
9	60	2020-12-10	98	124	269.0

Panadas – Data Cleaning

- Correcting the Wrong Data by replacing the value
- For small data sets you might be able to replace the wrong data one by one, but not for big data sets.

- To replace wrong data for larger data sets you can create some rules, e.g. set some boundaries for legal values, and replace any values that are outside of the boundaries.

Example

Loop through all values in the "Duration" column.

If the value is higher than 120, set it to 120:

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.loc[x, "Duration"] = 120
```

Panadas – Data Cleaning

- Correcting the Wrong Data by removing the rows
- Another way of handling wrong data is to remove the rows that contains wrong data.
- This way you do not have to find out what to replace them with, and there is a good chance you do not need them to do your analyses.

Example

Delete rows where "Duration" is higher than 120:

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.drop(x, inplace = True)
```

Panadas – Data Cleaning

- Removing Duplicate rows
- Duplicate rows are rows that have been registered more than one time.
- For instance, see row 11, and 12.
- To discover duplicates, we can use the *duplicated()* method.
- The *duplicated()* method returns a Boolean values for each row:

```
print (df.duplicated())
```

```
0     False
1     False
2     False
3     False
4     False
5     False
6     False
7     False
8     False
9     False
10    False
11    False
12     True
13    False
14    False
15    False
16    False
17    False
18    False
19    False
20    False
21    False
23    False
24    False
25    False
26    False
27    False
28    False
29    False
30    False
31    False
dtype: bool
```

```
import pandas as pd
df = pd.read_csv('dirtydata.csv')
df['Date'] = pd.to_datetime(df['Date'])
print(df.to_string())
```

	Duration	Date	Pulse	Maxpulse	Calories
0	60	2020-12-01	110	130	409.1
1	60	2020-12-02	117	145	479.0
2	60	2020-12-03	103	135	340.0
3	45	2020-12-04	109	175	282.4
4	45	2020-12-05	117	148	406.0
5	60	2020-12-06	102	127	300.0
6	60	2020-12-07	110	136	374.0
7	450	2020-12-08	104	134	253.3
8	30	2020-12-09	109	133	195.1
9	60	2020-12-10	98	124	269.0
10	60	2020-12-11	103	147	329.3
11	60	2020-12-12	100	120	250.7
12	60	2020-12-12	100	120	250.7
13	60	2020-12-13	106	128	345.3
14	60	2020-12-14	104	132	379.3
15	60	2020-12-15	98	123	275.0
16	60	2020-12-16	98	120	215.2
17	60	2020-12-17	100	120	300.0
18	45	2020-12-18	90	112	NaN
19	60	2020-12-19	103	123	323.0
20	45	2020-12-20	97	125	243.0
21	60	2020-12-21	108	131	364.2
22	45	NaT	100	119	282.0
23	60	2020-12-23	130	101	300.0
24	45	2020-12-24	105	132	246.0
25	60	2020-12-25	102	126	334.5
26	60	2020-12-26	100	120	250.0
27	60	2020-12-27	92	118	241.0
28	60	2020-12-28	103	132	NaN
29	60	2020-12-29	100	132	280.0
30	60	2020-12-30	102	129	380.3
31	60	2020-12-31	92	115	243.0

Example

Returns `True` for every row that is a duplicate, othwerwise `False` :

```
print(df.duplicated())
```

Panadas – Data Cleaning

- Removing Duplicate rows
- To remove duplicates, use the *drop_duplicates()* method.

- Remember: The *(inplace = True)* will make sure that the method does NOT return a new DataFrame, but it will remove all duplicates from the original DataFrame.

```
df.drop_duplicates(inplace = True)
df.head(15)
```

	Duration	Date	Pulse	Maxpulse	Calories
0	60	2020-12-01	110	130	409.1
1	60	2020-12-02	117	145	479.0
2	60	2020-12-03	103	135	340.0
3	45	2020-12-04	109	175	282.4
4	45	2020-12-05	117	148	406.0
5	60	2020-12-06	102	127	300.0
6	60	2020-12-07	110	136	374.0
7	45	2020-12-08	104	134	253.3
8	30	2020-12-09	109	133	195.1
9	60	2020-12-10	98	124	269.0
10	60	2020-12-11	103	147	329.3
11	60	2020-12-12	100	120	250.7
13	60	2020-12-13	106	128	345.3
14	60	2020-12-14	104	132	379.3
15	60	2020-12-15	98	123	275.0

Panadas – Data Correlation

- 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.
- The examples in this page uses a CSV file called: 'data.csv'.
- Note: The *corr()* method ignores "not numeric" columns.

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

	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

Panadas – Data Correlation

	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

Result Explained

The Result of the `corr()` method is a table with a lot of numbers that represents how well the relationship is between two columns.

The number varies from -1 to 1.

1 means that there is a 1 to 1 relationship (a perfect correlation), and for this data set, each time a value went up in the first column, the other one went up as well.

0.9 is also a good relationship, and if you increase one value, the other will probably increase as well.

-0.9 would be just as good relationship as 0.9, but if you increase one value, the other will probably go down.

0.2 means NOT a good relationship, meaning that if one value goes up does not mean that the other will.

Panadas – Data Correlation

What is a good correlation? It depends on the use, but I think it is safe to say you have to have at least `0.6` (or `-0.6`) to call it a good correlation.

Perfect Correlation:

We can see that "Duration" and "Duration" got the number `1.000000`, which makes sense, each column always has a perfect relationship with itself.

Good Correlation:

"Duration" and "Calories" got a `0.922721` correlation, which is a very good correlation, and we can predict that the longer you work out, the more calories you burn, and the other way around: if you burned a lot of calories, you probably had a long work out.

Bad Correlation:

"Duration" and "Maxpulse" got a `0.009403` correlation, which is a very bad correlation, meaning that we can not predict the max pulse by just looking at the duration of the work out, and vice versa.

Panadas – Selecting Data: Selecting specific columns

```
import pandas as pd
df = pd.read_csv('data.csv')
#dduration = df['Duration']
duration=df.Duration
duration.head()
print(type(duration))
duration.shape
```

```
<class 'pandas.core.series.Series'>

(169,)
```

```
dur_cal = df[["Duration","Calories"]]
dur_cal.head()
```

	Duration	Calories
0	60	409.1
1	60	479.0
2	60	340.0
3	45	282.4
4	45	406.0

```
print(type(dur_cal))
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
dur_cal.shape
```

```
(169, 2)
```

Panadas – Selecting Data: Selecting specific columns

- What will happen when you flip the order or columns?

- What will happen when you entered the wrong column name?

```
print(df[["Calories" , "Duration"]])
```

```
print(df[["Calories" , "Duratin"]])
```

Panadas – Selecting Data: Selecting specific Rows

- Select first 5 rows

- How will you select last row?

```
#df[0:5]
df [:5]
```

	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

Panadas – Selecting Data: Selecting specific Rows

- Select first 5 rows

- How will you select last row?

```
df[-1:]
```

	Duration	Pulse	Maxpulse	Calories
168	75	125	150	330.4

```
#df[0:5]
df [:5]
```

	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

Panadas – Selecting Data: Selecting specific Rows

- Select first 3 rows but with column 2, 3, and 4.

```
# iloc[row slicing, column slicing]
df.iloc[0:3, 1:4]
```

	Pulse	Maxpulse	Calories
0	110	130	409.1
1	117	145	479.0
2	103	135	340.0

Panadas – Selecting Data: Selecting specific Rows

```
df.iloc[0:10,:]
```

	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
5	60	102	127	300.0
6	60	110	136	374.0
7	45	104	134	253.3
8	30	109	133	195.1
9	60	98	124	269.0

```
df.loc[0:10,:]
```

	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
5	60	102	127	300.0
6	60	110	136	374.0
7	45	104	134	253.3
8	30	109	133	195.1
9	60	98	124	269.0
10	60	103	147	329.3

```
# To get Friday's temperature
>>> df.loc['Fri', 'Temperature']

10.51
```

The equivalent `iloc` statement should take the row number `4` and the column number `1` .

```
# The equivalent 'iloc' statement
>>> df.iloc[4, 1]

10.51
```

Panadas – Filtering Data

```
# df_filtered = df[(index with expression) & (index with expression)]
df_filtered = df[(df.Duration>=60) & (df.Calories >=400)]
print(df_filtered.shape)
print(df_filtered.head())
```

```
(36, 4)
    Duration  Pulse  Maxpulse  Calories
0         60    110       130     409.1
1         60    117       145     479.0
35        60    114       140     415.0
43        60    111       138     400.0
51        80    123       146     643.1
```

Panadas – Filtering Data

```
#A data frames columns can be queried with a boolean expression.
#Every frame has the module query() as one of its objects members.
df_filtered = df.query('Duration >= 60 & Calories >= 400')
print(df_filtered.shape)
print(df_filtered.head())
```

```
(36, 4)
    Duration  Pulse  Maxpulse  Calories
0         60    110       130     409.1
1         60    117       145     479.0
35        60    114       140     415.0
43        60    111       138     400.0
51        80    123       146     643.1
```

Working with Data Visualization

Why Build Visuals

- For exploratory data analysis
- Communicate data clearly
- Share unbiased representation of data
- Use them to support recommendations to different stakeholders
- Several data visualization libraries in Python
  - Matplotlib
  - Pandas Visualization
  - Seaborn
  - Ggplot
  - Plotly

About Matplotlib

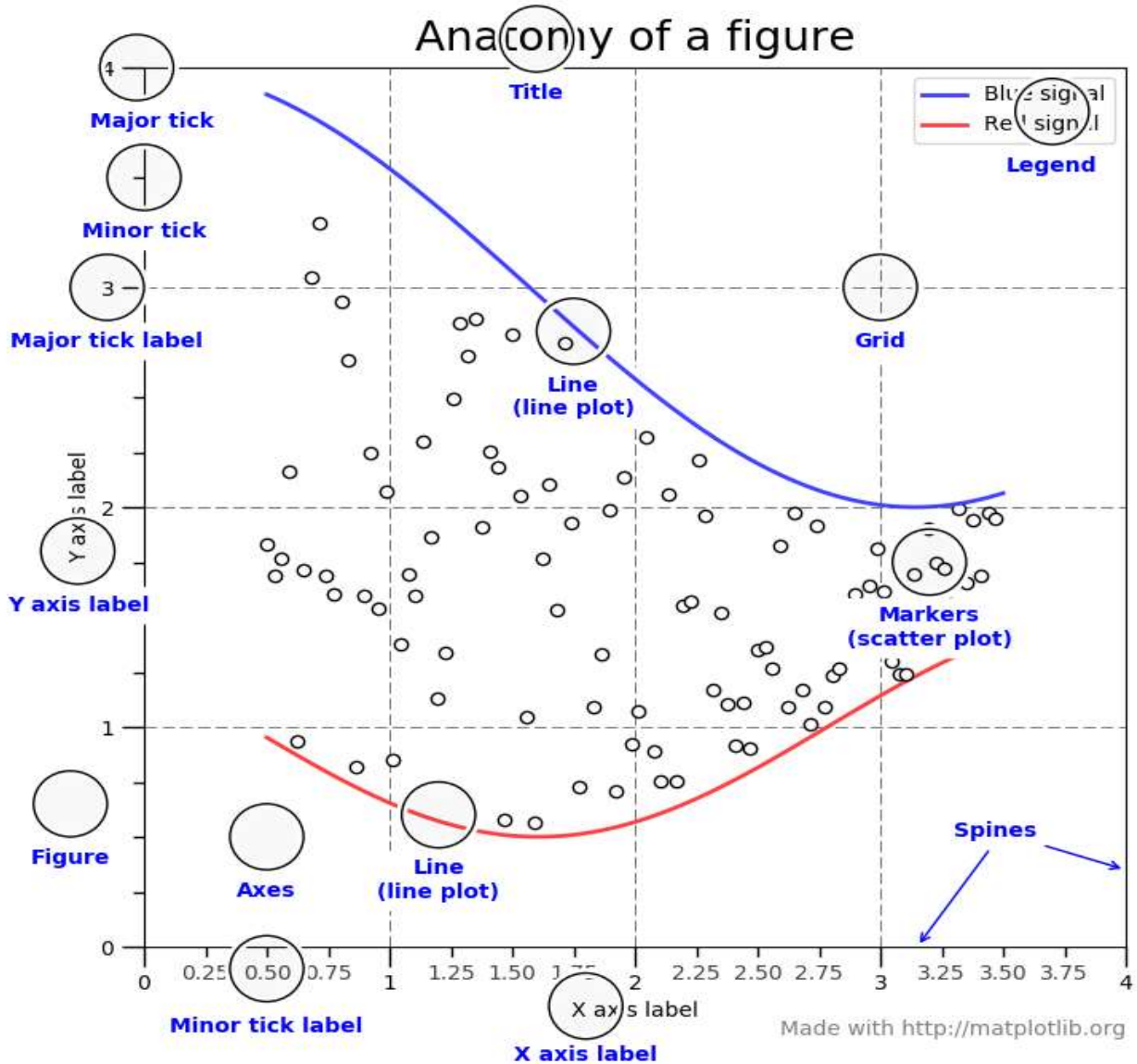
- Matplotlib is a low-level graph plotting library in python that serves as a visualization utility.
- Matplotlib was created by John D. Hunter.
- Matplotlib is opensource and we can use it freely.
- Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Java script for Platform compatibility.
- The source code for Matplotlib is located at this github repository https://github.com/matplotlib/matplotlib

About Matplotlib

```
C:\Users\Your Name>pip install matplotlib
```

Example

```
import matplotlib

print(matplotlib.__version__)
```

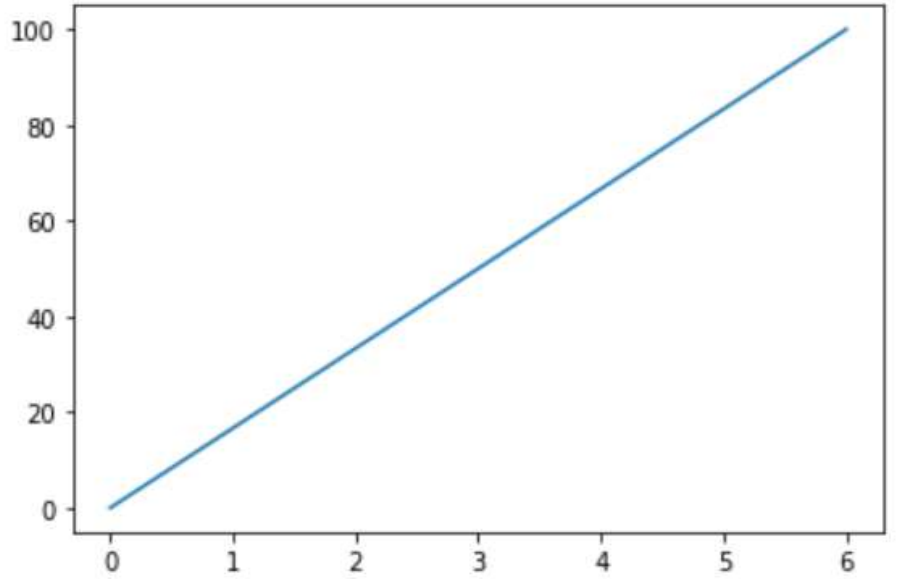

Anatomy of a figure
Title
Major tick
Minor tick
Major tick label
Grid
Legend
Line
(line plot)
Y axis label
Markers
(scatter plot)
Figure
Axes
Line
(line plot)
Spines
Minor tick label
X axis label
X axis label
Made with http://matplotlib.org

About Sub Module PyPlot of Matplotlib Module

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

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

```
#Example: Draw a line in a diagram from position (0,0) to position (6,250):
import matplotlib.pyplot as plt
import numpy as np
x_points = np.array([0,6])
y_points = np.array([0,100])
plt.plot (x_points,y_points)
plt.show ()
```



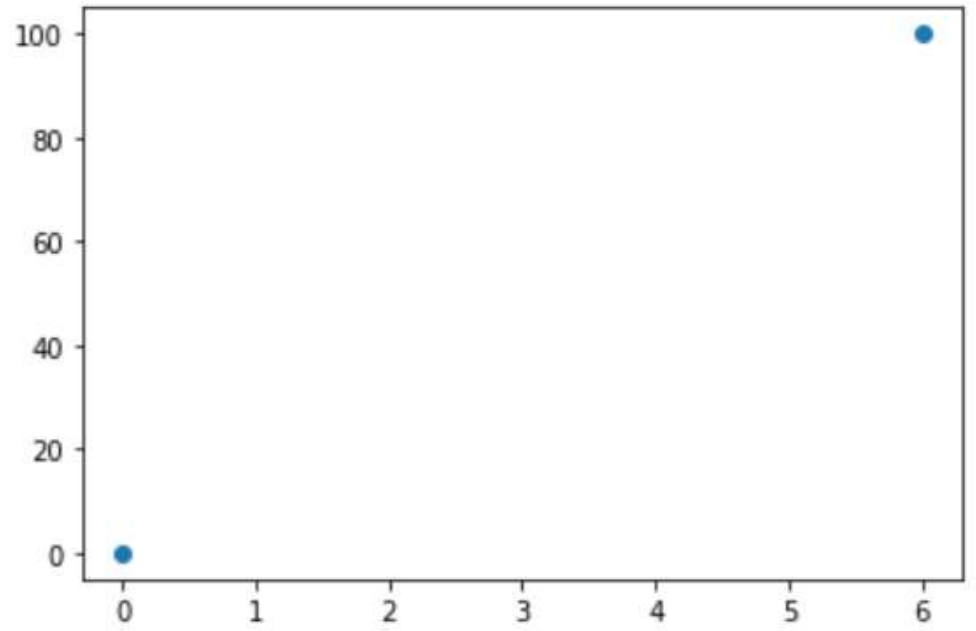
Plot () Function

- 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.

Plotting without a 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):
import matplotlib.pyplot as plt
import numpy as np
x_points = np.array([0,6])
y_points = np.array([0,100])
plt.plot (x_points,y_points,'o')
plt.show ()
```

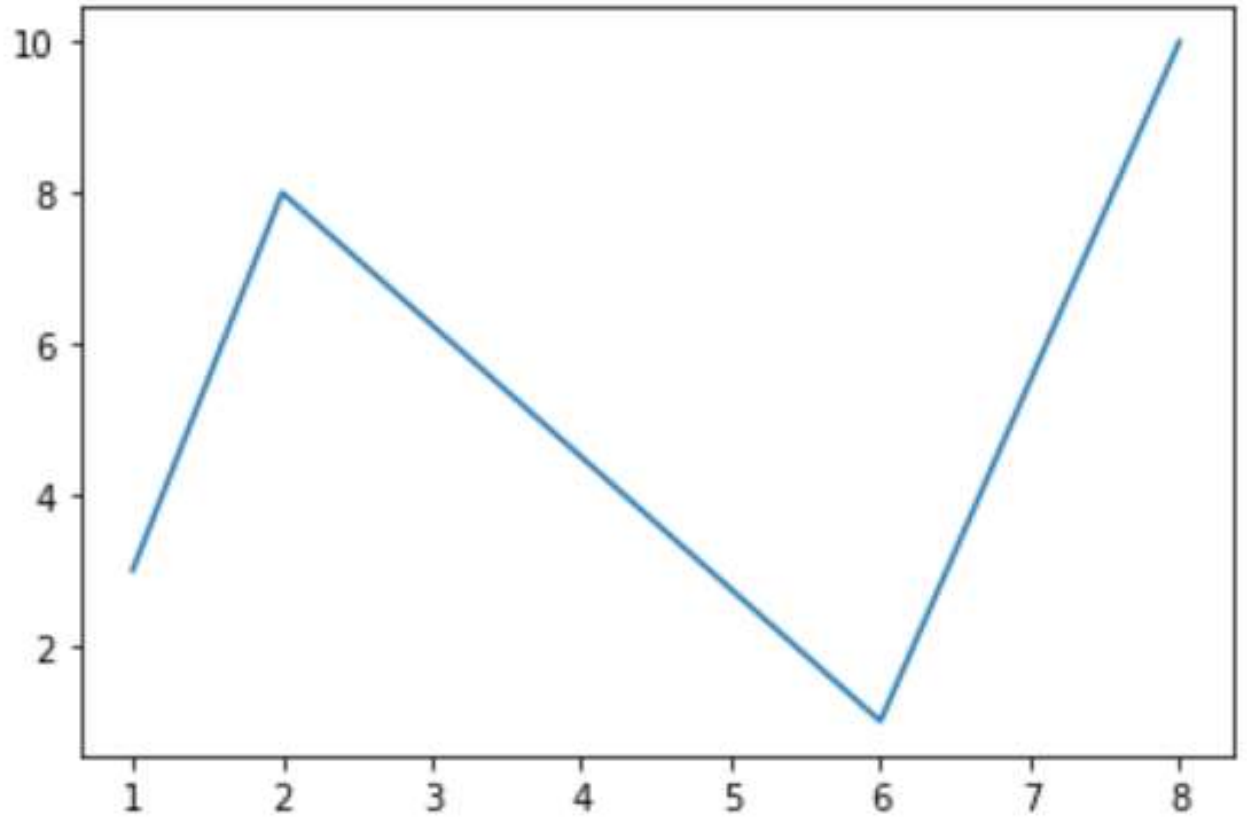


Draw Multiple Points

- 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):

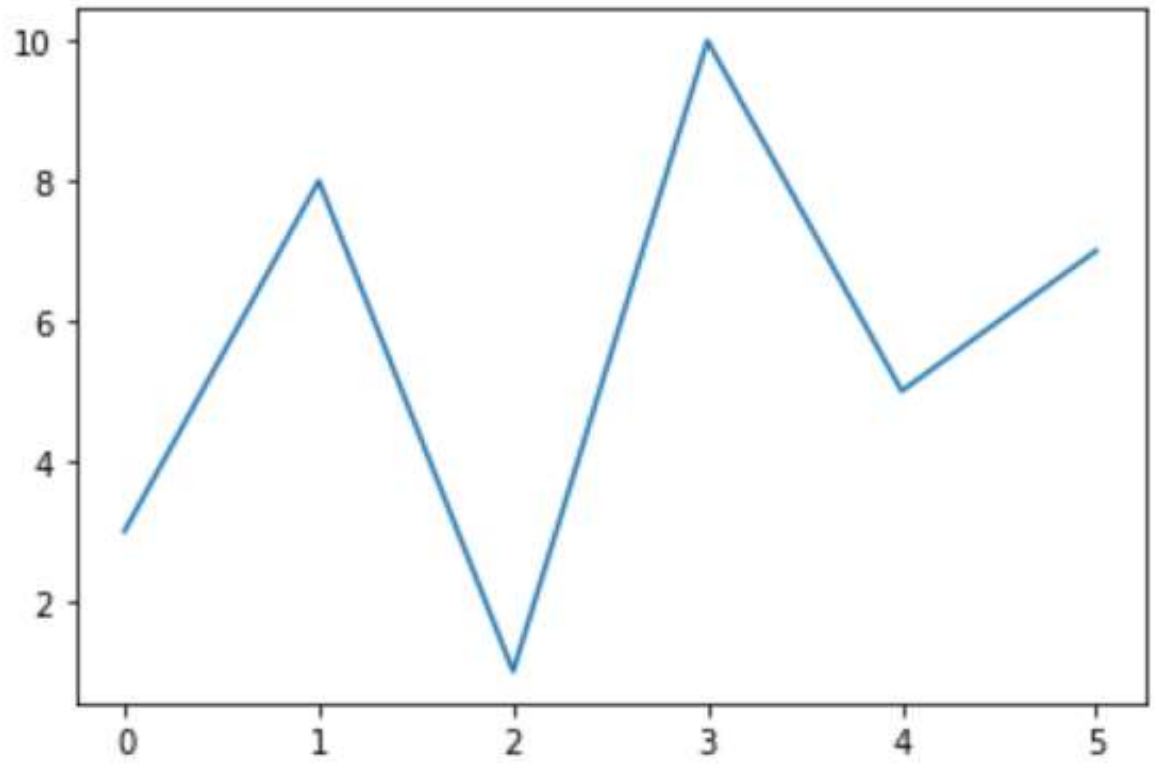
```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])

plt.plot(xpoints, ypoints)
plt.show()
```



Default X Values

- 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.

```
import matplotlib.pyplot as plt
import numpy as np
ypoints = np.array([3, 8, 1, 10, 5, 7])
plt.plot(ypoints)
plt.show()
```

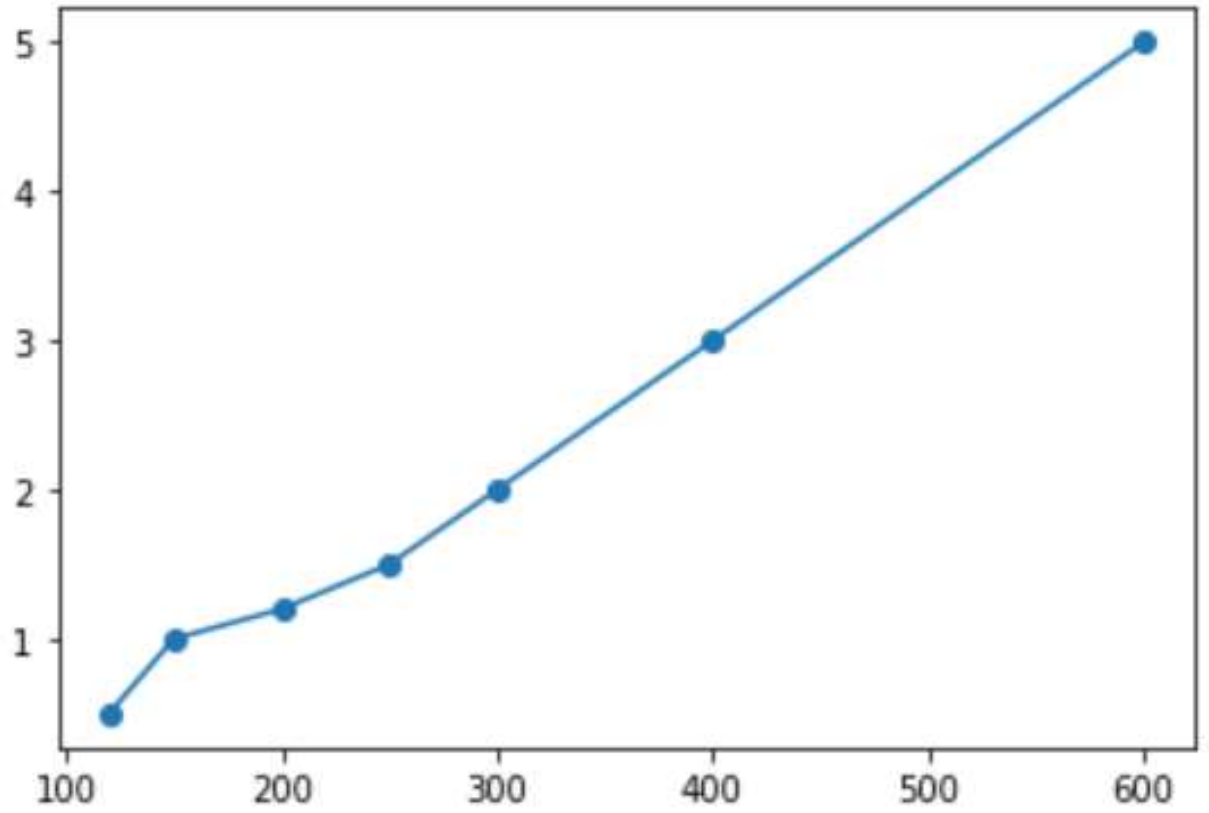


Matplot Markers

- You can use the keyword argument *marker* to emphasize each point with a specified marker:
- Notice the difference between following three lines of code.

```
#this line will draw just line
plt.plot (home_sizes,home_prices)

#this line will draw just markers
plt.plot (home_sizes,home_prices,'o')

#this line will draw line and markers both
plt.plot (home_sizes,home_prices,marker = 'o')
```

```
import matplotlib.pyplot as plt
import numpy as np
home_sizes = np.array ([120,150,200,250,300,400,600])
home_prices = np.array ([0.5,1.0,1.2,1.5,2.0,3.0,5.0])
plt.plot (home_sizes,home_prices,marker = 'o')
plt.show ()
```



Marker Reference

You can choose any of these markers:

','	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	Vline
'_'	Hline

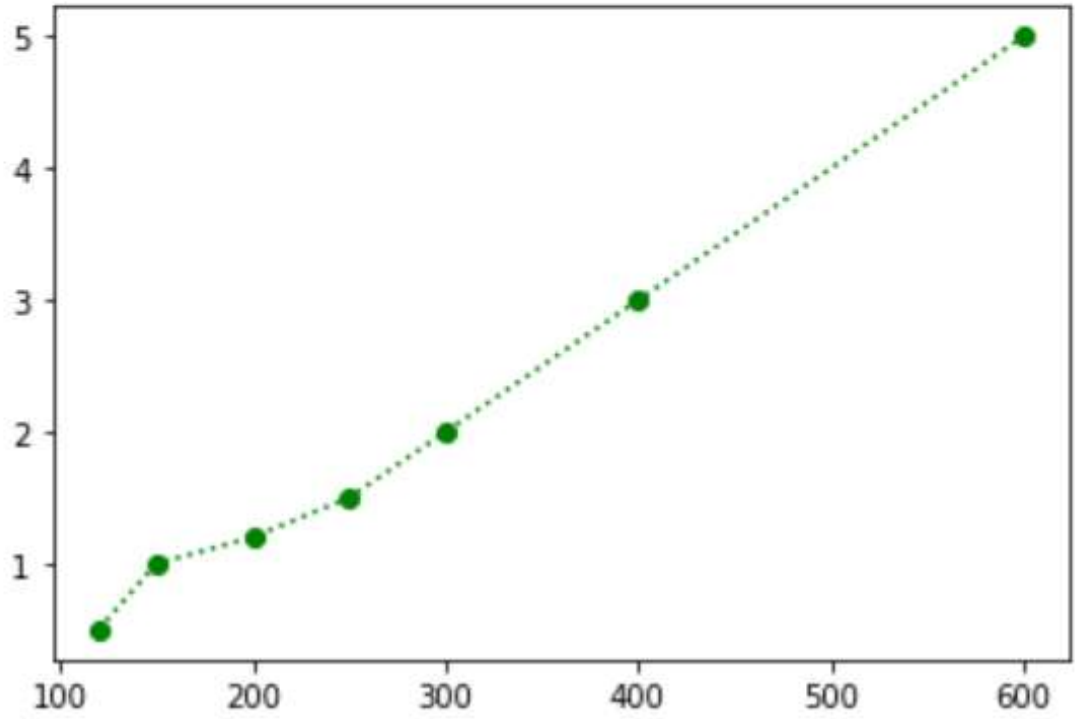
Matplot Format Strings fmt

- You can use also use the *shortcut string notation* parameter to specify the marker.

- This parameter is also called *fmt*, and is written with this syntax:

- marker|line|color

```
import matplotlib.pyplot as plt
import numpy as np
home_sizes = np.array ([120,150,200,250,300,400,600])
home_prices = np.array ([0.5,1.0,1.2,1.5,2.0,3.0,5.0])
#her o is marker, : is used for dotted line and g is used for green color
plt.plot (home_sizes,home_prices,'o:g')
plt.show ()
```



Marker Reference

','	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	Vline
'_'	Hline

Line Reference

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

Color Reference

Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

Marker Size

You can use the keyword argument *markersize* or the shorter version, *ms* to set the size of the markers:

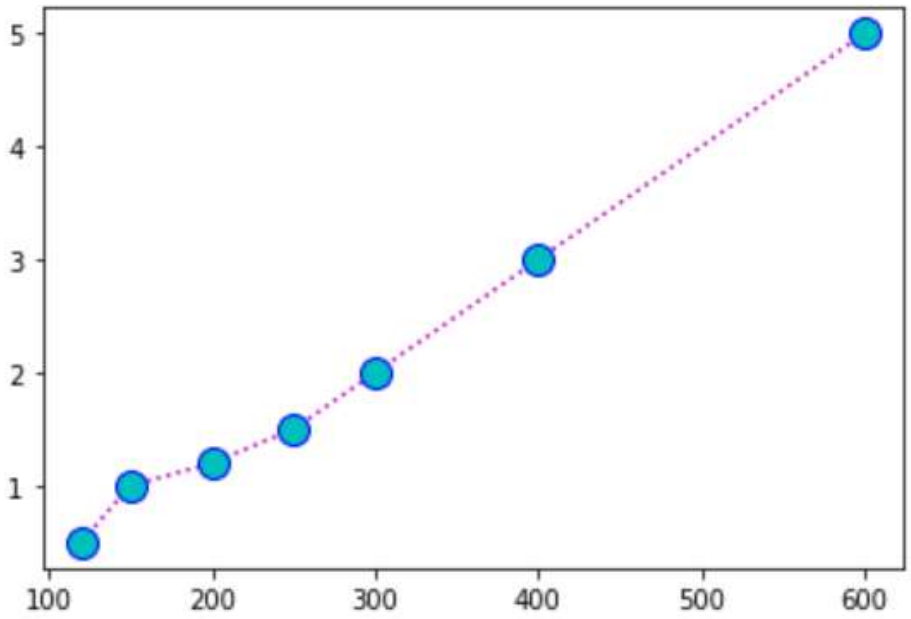
Marker Edge Color

You can use the keyword argument *markeredgecolor* or the shorter *mec* to set the color of the edge of the markers:

Marker Face Color

You can use the keyword argument *markerfacecolor* or the shorter *mfc* to set the color inside the edge of the markers:

```
plt.plot (home_sizes,home_prices,'o:m',ms=12,mec='b',mfc='c')
plt.show ()
```



You can also use hexadecimal color values
You can use 140 supporting HTML color names

Matplotlib Line

- You can use the keyword argument *linestyle*, or shorter *ls*, to change the style of the plotted line:

- You can use the keyword argument *color* or the shorter *c* to set the color of the line:

- You can use the keyword argument *linewidth* or the shorter *lw* to change the width of the line. The value is a floating number, in points:

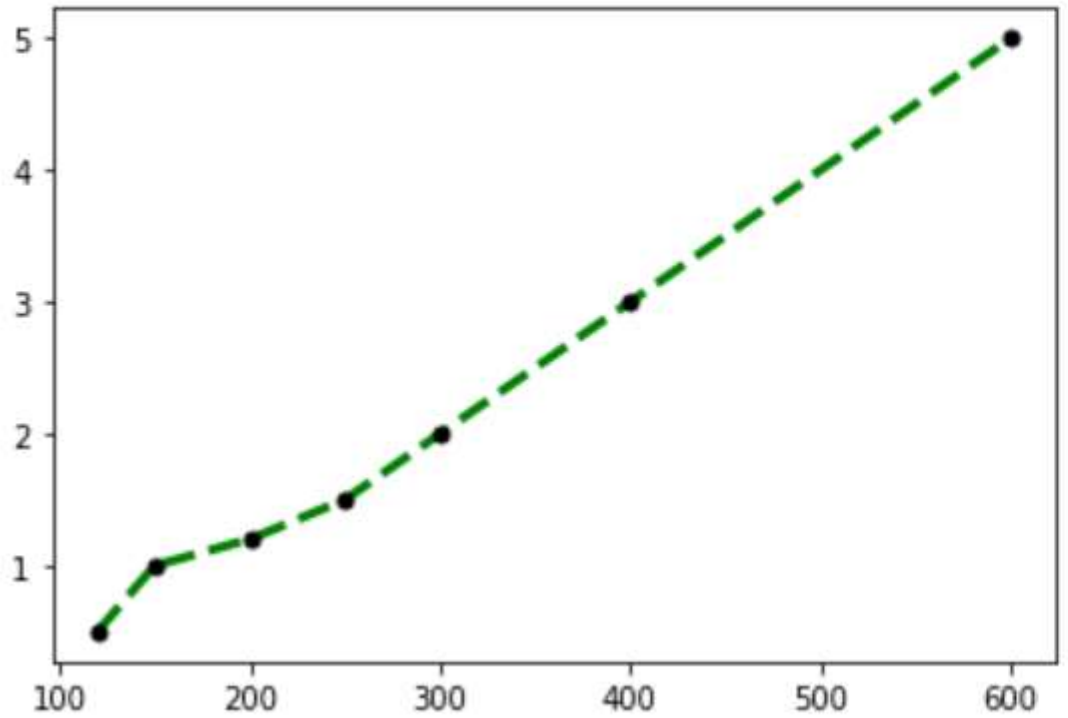
Line Styles

You can choose any of these styles:

Style	Or
'solid' (default)	'-'
'dotted'	':'
'dashed'	'--'
'dashdot'	'-.'
'None'	'' or ' '

Matplotlib Line

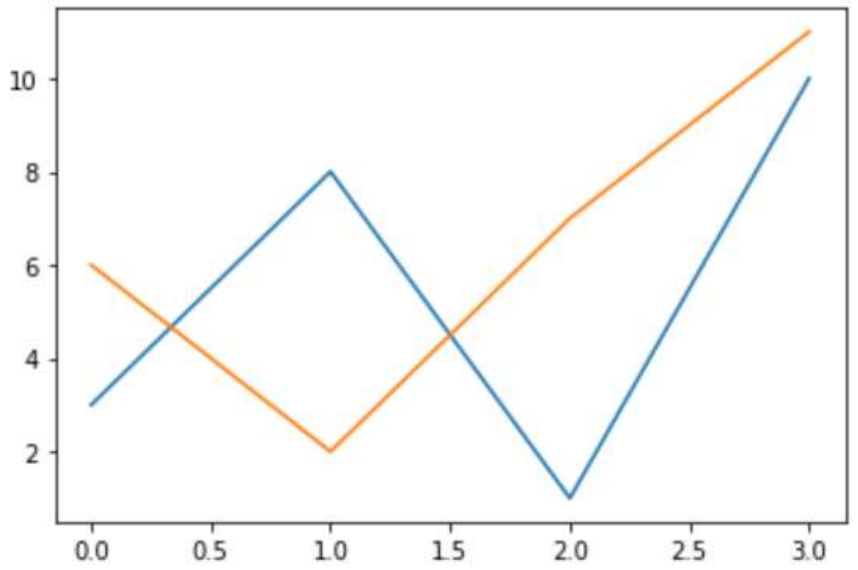
```
plt.plot (home_sizes,home_prices, marker='o' , ms=5, mec='k', mfc='k', ls = '--', c='g', lw='3.0')
plt.show ()
```



Matplotlib Drawing Multiple Lines

- You can plot as many lines as you like by simply adding more *plt.plot()* functions:

```
#Example: Draw two lines by specifying a plt.plot() function for each line:
import matplotlib.pyplot as plt
import numpy as np

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1)
plt.plot(y2)

plt.show()
```



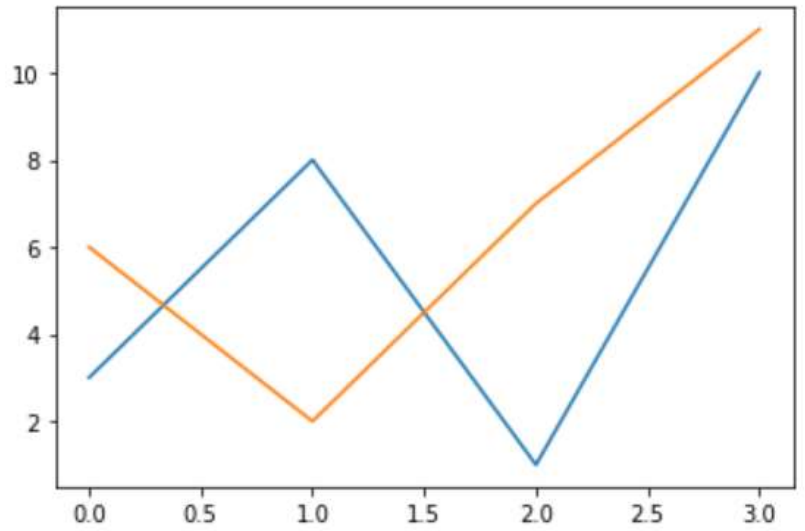
Matplotlib Drawing Multiple Lines

- You can also plot many lines by adding the points for the x- and y-axis for each line in the same plt.plot() function.

- (In the examples above we only specified the points on the y-axis, meaning that the points on the x-axis got the the default values (0, 1, 2, 3).)

- The x- and y- values come in pairs:

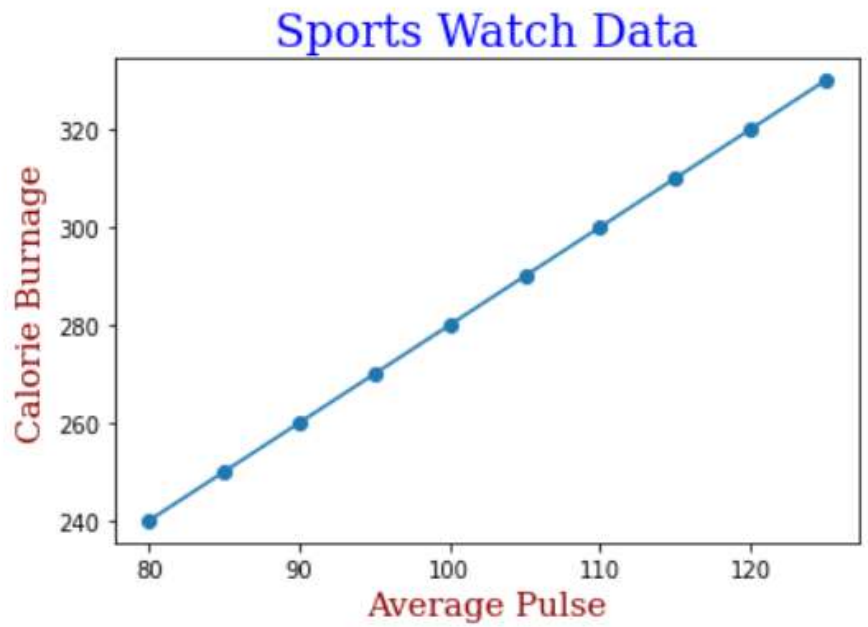
```
#Example: Draw two lines by specifiyng the x- and y-point values for both lines:
import matplotlib.pyplot as plt
import numpy as np

x1 = np.array([0, 1, 2, 3])
y1 = np.array([3, 8, 1, 10])
x2 = np.array([0, 1, 2, 3])
y2 = np.array([6, 2, 7, 11])

plt.plot(x1, y1, x2, y2)
plt.show()
```



Matplotlib Labels and Titles

- With Pyplot, you can use the *xlabel()* and *ylabel()* functions to set a label for the x- and y-axis.
- With Pyplot, you can use the *title()* function to set a title for the plot.
- You can use the *fontdict* parameter in *xlabel()*, *ylabel()*, and *title()* to set font properties for the title and labels.
- You can use the loc parameter in *title()* to position the title.
- Legal values are: *'left'*, *'right'*, and *'center'*. Default value is *'center'*.

```python
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])

font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}

plt.title("Sports Watch Data", fontdict = font1, loc = 'center')
plt.xlabel("Average Pulse", fontdict = font2)
plt.ylabel("Calorie Burnage", fontdict = font2)

plt.plot(x, y , marker = 'o')
plt.show()
```



Matplotlib: Adding the Grid Lines

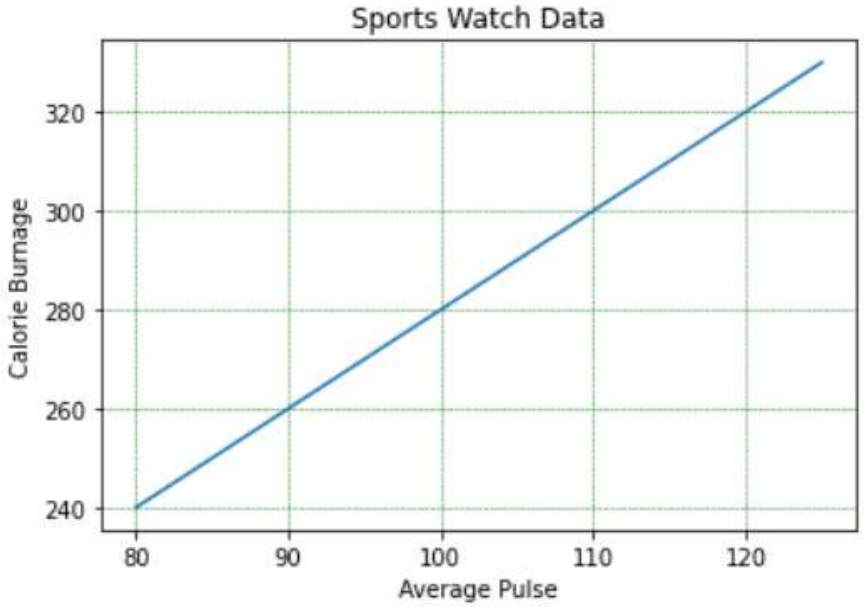
- With Pyplot, you can use the *grid()* function to add grid lines to the plot.

- You can use the *axis* parameter in the *grid()* function to specify which grid lines to display.

- Legal values are: *'x'*, *'y'*, and *'both'*. Default value is *'both'*.

- You can also set the *line* properties of the grid, like this: grid(color = 'color', linestyle = 'linestyle', linewidth = number).

```python
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(axis = 'both' , color = 'green', linestyle = '--', linewidth = 0.5)

plt.show()
```



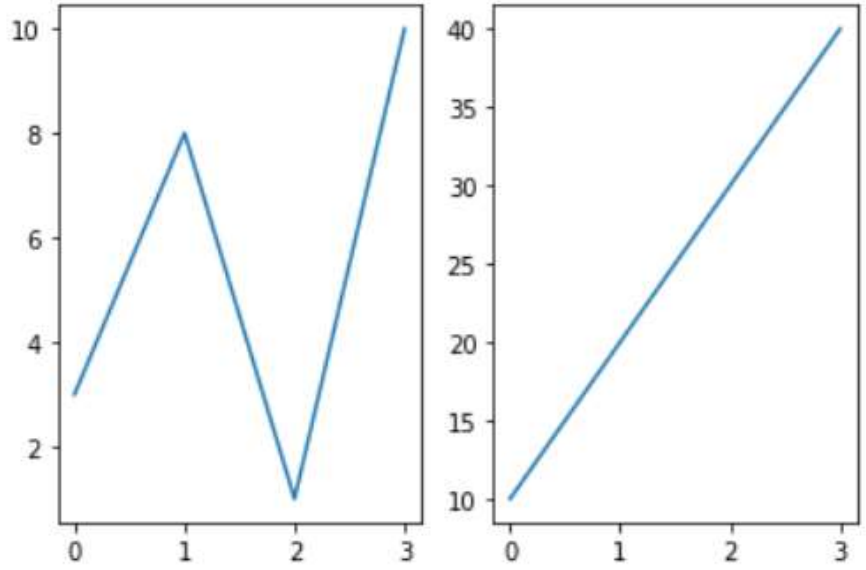
Matplotlib: SubPlots

- With the *subplots()* function you can draw multiple plots in one figure.
- The *subplots()* function takes three arguments that describes the layout of the figure.
- The layout is organized in rows and columns, which are represented by the first and second argument.
- The third argument represents the index of the current plot.
- You can add a title to each plot with the *title()* function:
- You can add a title to the entire figure with the *suptitle()* function:

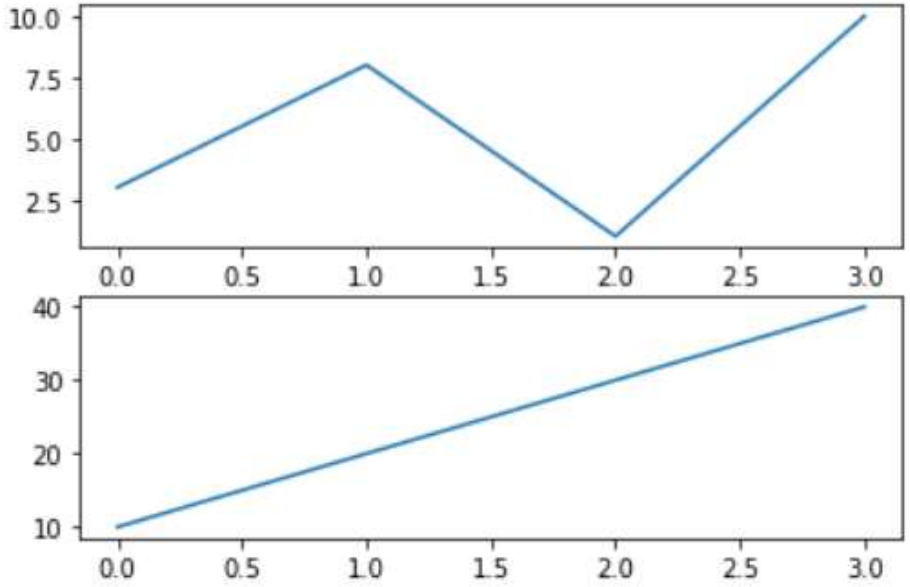
```
plt.subplot(1, 2, 1)
#the figure has 1 row, 2 columns, and this plot is the first plot.
```

```
plt.subplot(1, 2, 2)
#the figure has 1 row, 2 columns, and this plot is the second plot.
```

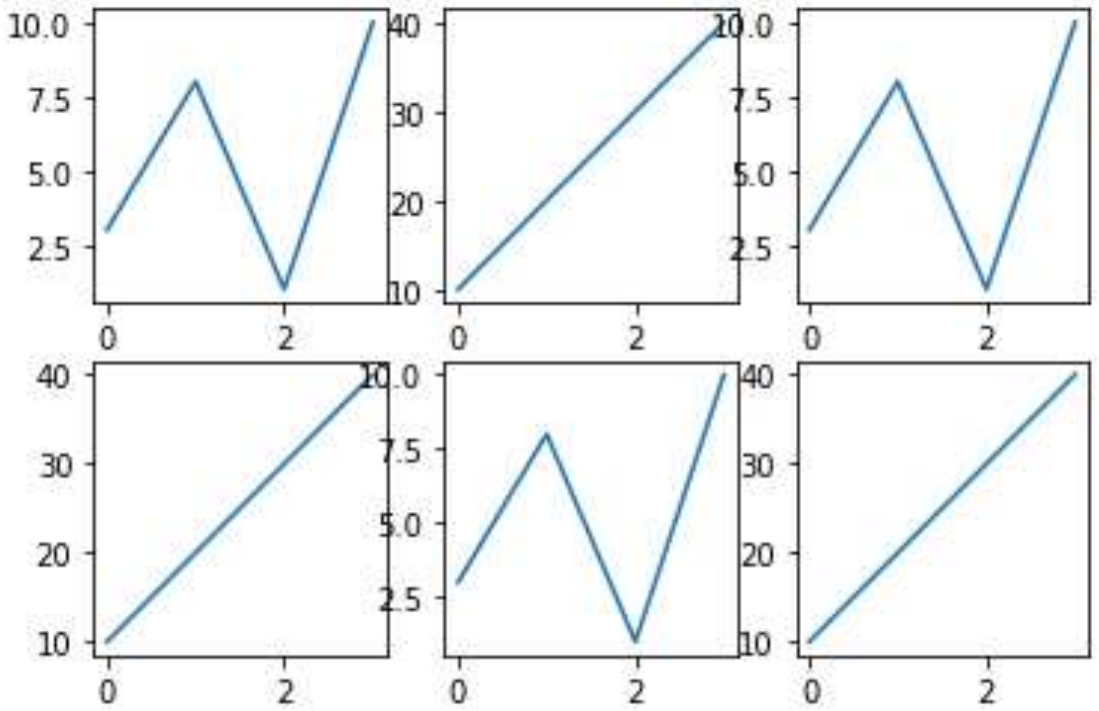
```python
import matplotlib.pyplot as plt
import numpy as np
#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(1, 2, 1)
plt.plot(x,y)
#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.show()
```




```python
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(2, 1, 1)
plt.plot(x,y)
#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 1, 2)
plt.plot(x,y)
plt.show()
```

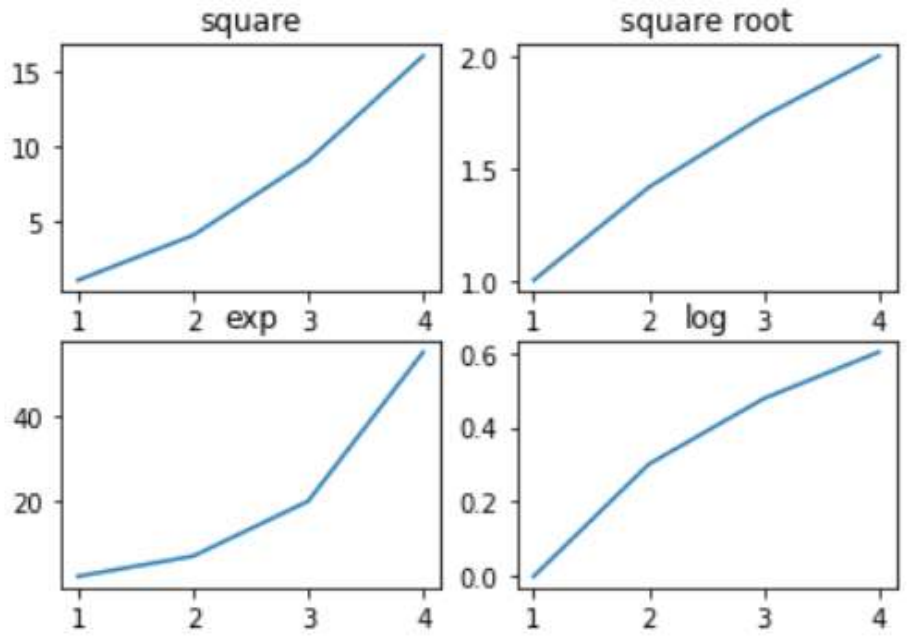


```python
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(2, 3, 1)
plt.plot(x,y)
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 3, 2)
plt.plot(x,y)
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(2, 3, 3)
plt.plot(x,y)
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 3, 4)
plt.plot(x,y)
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(2, 3, 5)
plt.plot(x,y)
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 3, 6)
plt.plot(x,y)
plt.show()
```

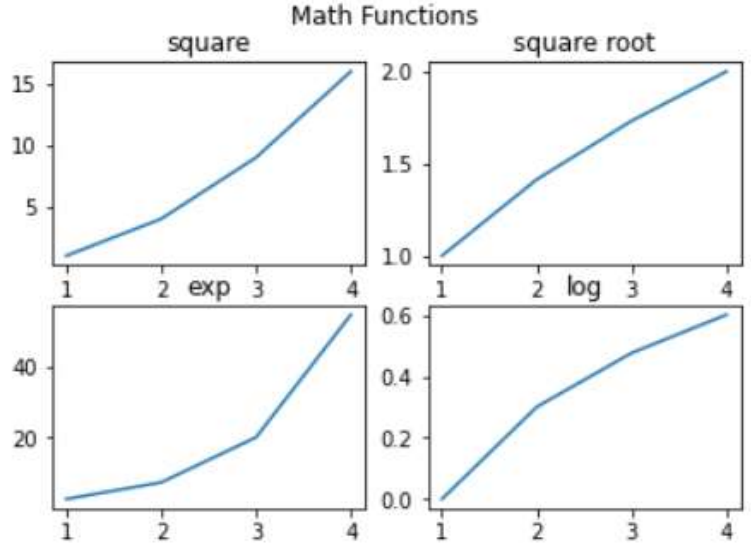


```python
import matplotlib.pyplot as plt
import numpy as np

#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)
plt.title("SALES")

#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.title("INCOME")

plt.suptitle("MY SHOP")
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np
fig,a =  plt.subplots(2,2)
x = np.arange(1,5) # x value is from 1 to 4
a[0][0].plot(x,x*x)
a[0][0].set_title('square')
a[0][1].plot(x,np.sqrt(x))
a[0][1].set_title('square root')
a[1][0].plot(x,np.exp(x))
a[1][0].set_title('exp')
a[1][1].plot(x,np.log10(x))
a[1][1].set_title('log')
plt.show()
```




```
import matplotlib.pyplot as plt
import numpy as np
#fig,a =  plt.subplots(2,2)
x = np.arange(1,5) # x value is from 1 to 4
plt.subplot (2,2,1)
plt.plot(x,x*x)
plt.title('square')
plt.subplot(2,2,2)
plt.plot(x,np.sqrt(x))
plt.title('square root')
plt.subplot(2,2,3)
plt.plot(x,np.exp(x))
plt.title('exp')
plt.subplot(2,2,4)
plt.plot(x,np.log10(x))
plt.title('log')
plt.suptitle('Math Functions')
plt.show()
```



Matplotlib.pyplot.subplot2grid()

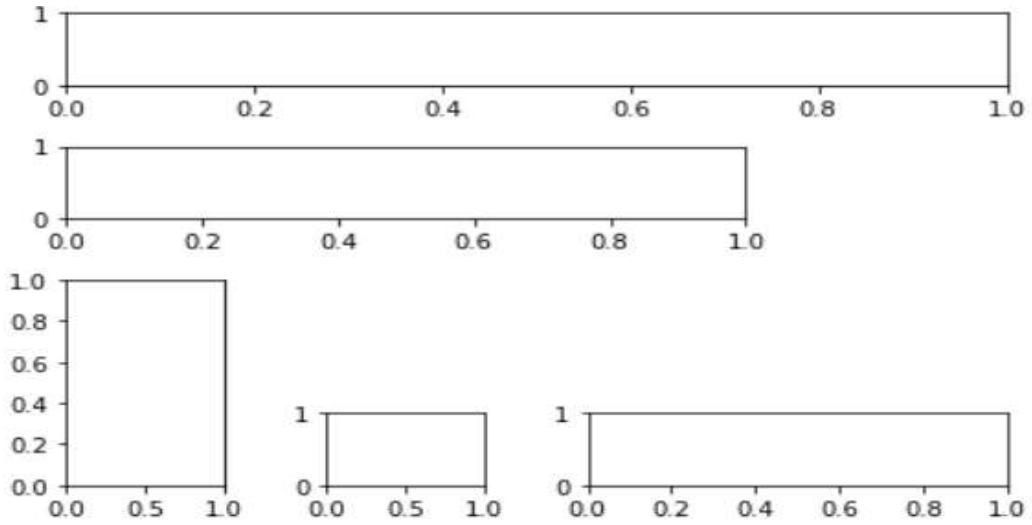
- The *Matplotlib.pyplot.subplot2grid()* function gives additional flexibility in creating axes object at a specified location inside a grid.
- It also helps in spanning the axes object across multiple rows or columns.
- Syntax : Plt.subplot2grid(shape, location, rowspan, colspan)

Parameters :

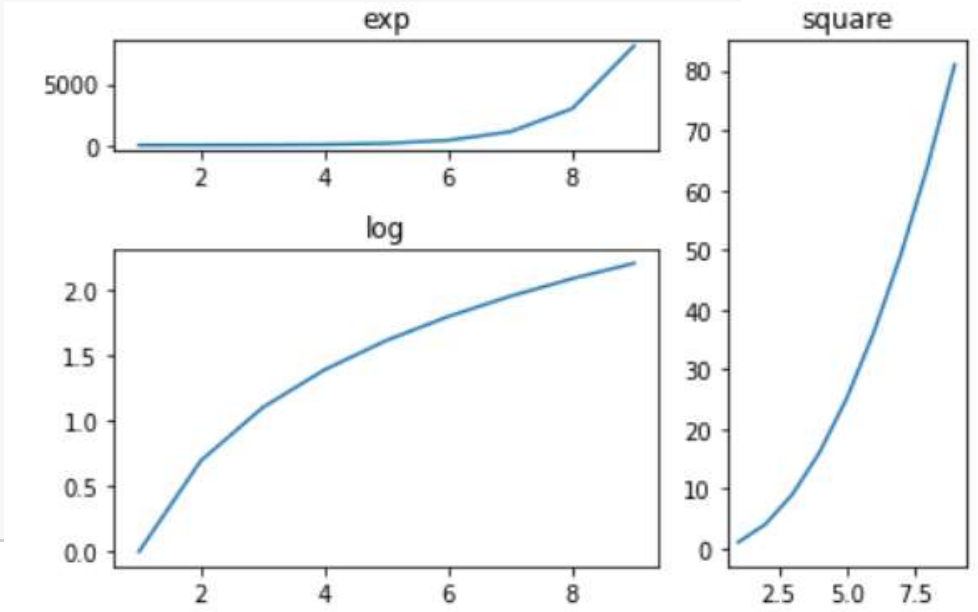
- ***shape:*** *As the name suggests it is used to define the shape of the grid to be plotted within the graph. It is a required argument and is generally passed in as a list or tup le of two numbers which are responsible for the layout of the grid with the first num ber being the number of rows and the second number as the number of columns.*
- ***location (loc):*** *This is the second mandatory argument that this function takes. Sim ilar to the shape argument it is also a required argument and is generally passed in as a list or tuple of two numbers. It is used for specifying the row and column numb er to place the sub-plot. It is also important to note that the indexes start from 0. So (0, 0) would be the cell in the first row and the first column of the grid.*
- ***rowspan:*** *Once the grid layout is set and the starting index is decided using location (loc) one can expand the selection to take up more rows with this argument. This is an optional parameter and has a default value of 1.*
- ***colspan:*** *Similar to rowspan it is used to expand the selection to take up more colu mns. It is also an optional parameter with default value of 1.*

```
import matplotlib.pyplot as plt
# draw 4 x 4 figure, on 1st row 1st columns merge four columns
axes1 = plt.subplot2grid((4, 4), (0, 0), colspan = 4)
# draw 4 x 4 figure, on 2nd row 1st column merge 3 columns
axes2 = plt.subplot2grid((4, 4), (1, 0),colspan = 3)
# draw 4 x 4 figure, on 2nd row 1st columns merge 2 row
axes3 = plt.subplot2grid((4, 4), (2, 0),rowspan = 2)
# draw 4 x 4 figure, on 4th row 2nd column
axes4 = plt.subplot2grid((4, 4), (3, 1))
# draw 4 x 4 figure, on 4th row 3rd column merge 2 columns
axes5 = plt.subplot2grid((4, 4), (3, 2),colspan=2)
plt.tight_layout()
```

tight_layout **automatically adjusts subplot params** so that the subplot(s) fits in to the figure area.



```
import matplotlib.pyplot as plt
import numpy as np
#Plt.subplot2grid(shape, location, rowspan, colspan)
a1 = plt.subplot2grid((3,3),(0,0),colspan = 2)
a2 = plt.subplot2grid((3,3),(0,2), rowspan = 3)
a3 = plt.subplot2grid((3,3),(1,0),rowspan = 2, colspan = 2)
x = np.arange(1,10)
a2.plot(x, x*x)
a2.set_title('square')
a1.plot(x, np.exp(x))
a1.set_title('exp')
a3.plot(x, np.log(x))
a3.set_title('log')
plt.tight_layout()
plt.show()
```



How To Decide Which Type of Chart to Use?

Do you want to compare values?

- If you want to know more information about how a data set performed during a specific time period.
  - Column
  - Bar
  - Pie
  - Line
  - Scatter Plot

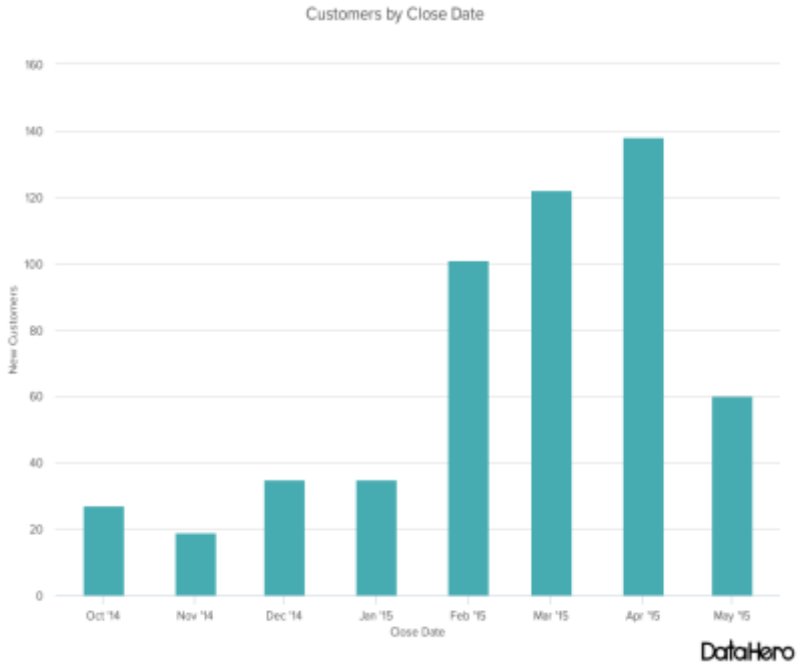
Do you want to show the composition of something?

- Use this type of chart to show how individual parts make up the whole of something.
  - Pie
  - Stacked Bar
  - Stacked Column
  - Area
  - Waterfall

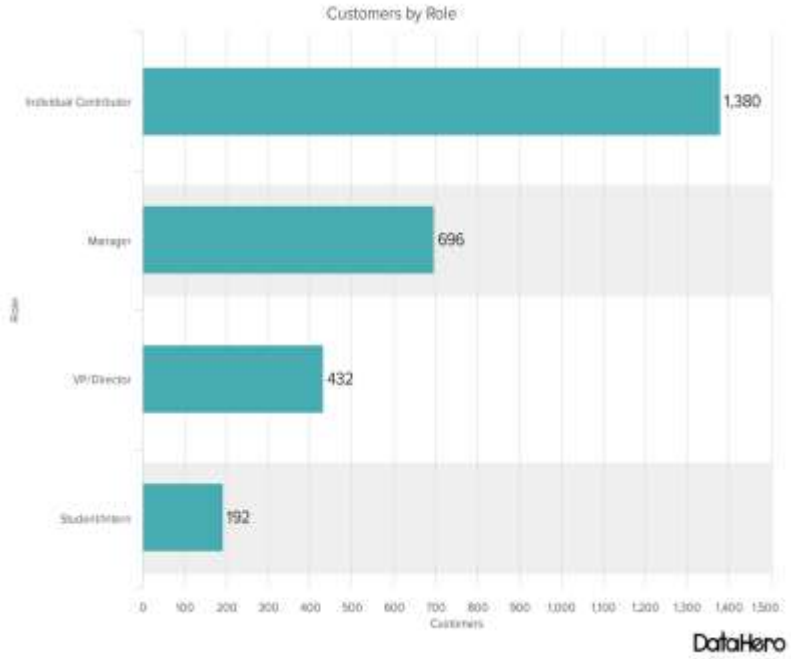
Do you want to understand the distribution of your data?

- Distribution charts help you to understand outliers, the normal tendency, and the range of information in your values.
  - Scatter Plot
  - Line
  - Column
  - Bar

Types of Charts

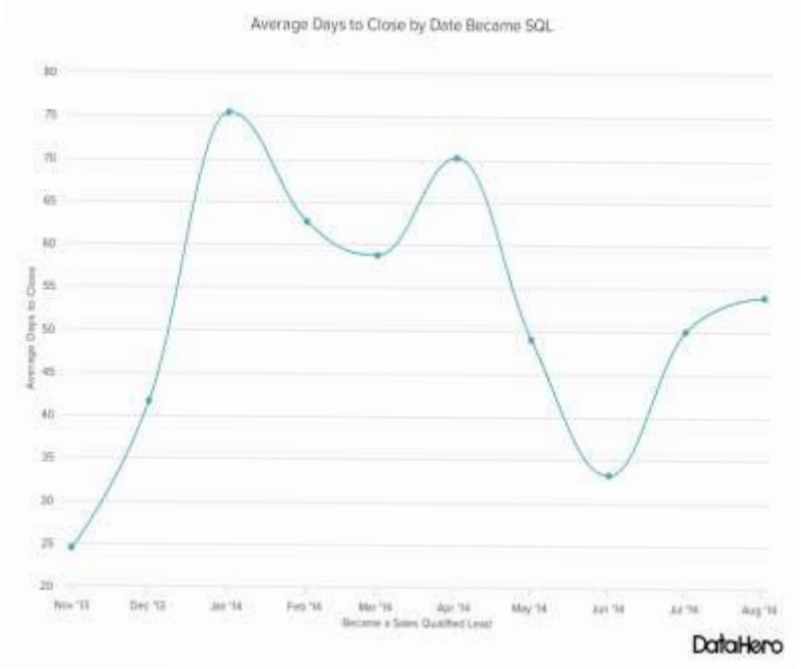


- Column Charts

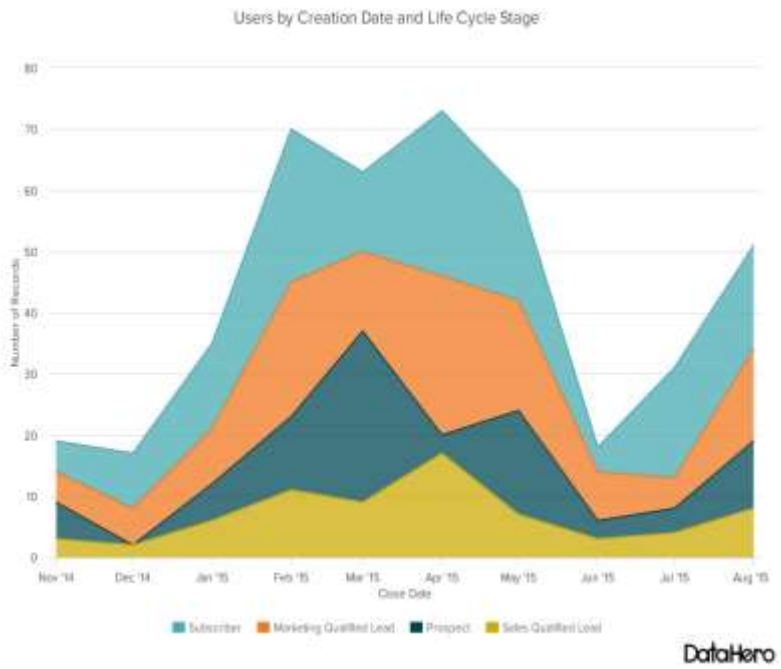


- Bar Charts

Types of Charts

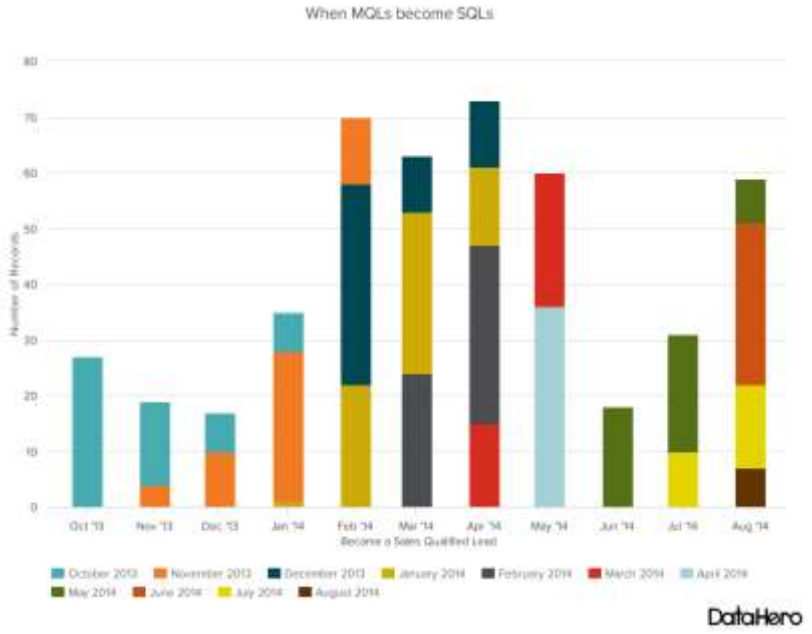


- Line Charts

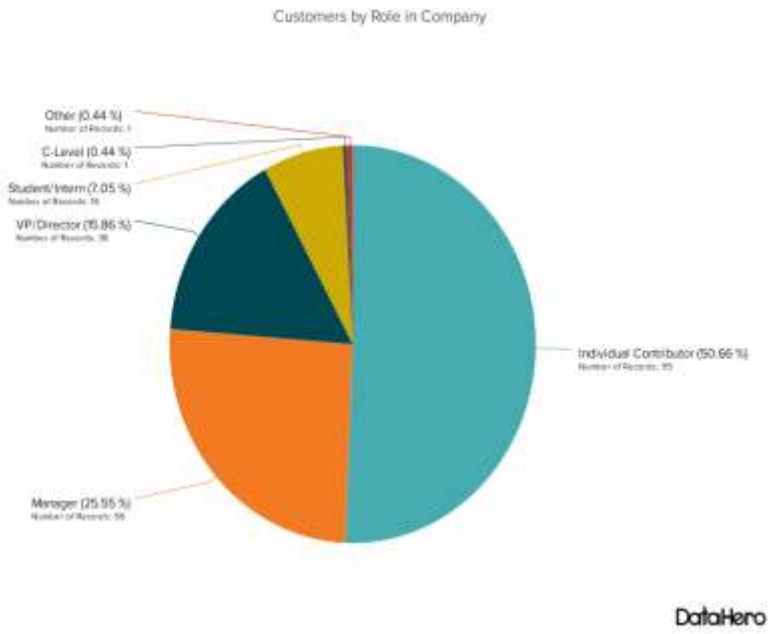


- Area Charts

Types of Charts



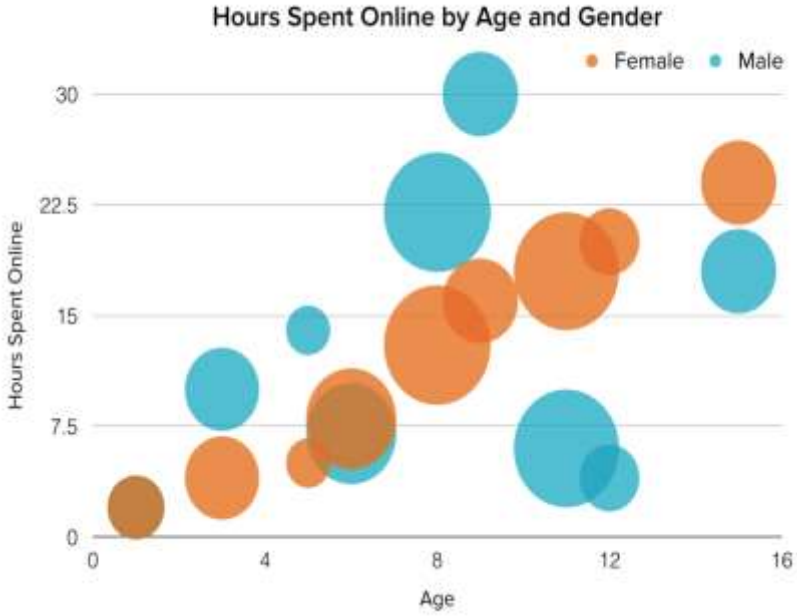
- Stacked Bar Charts



- Pie Charts

Types of Charts

- Scatter Charts



- Bubble Charts

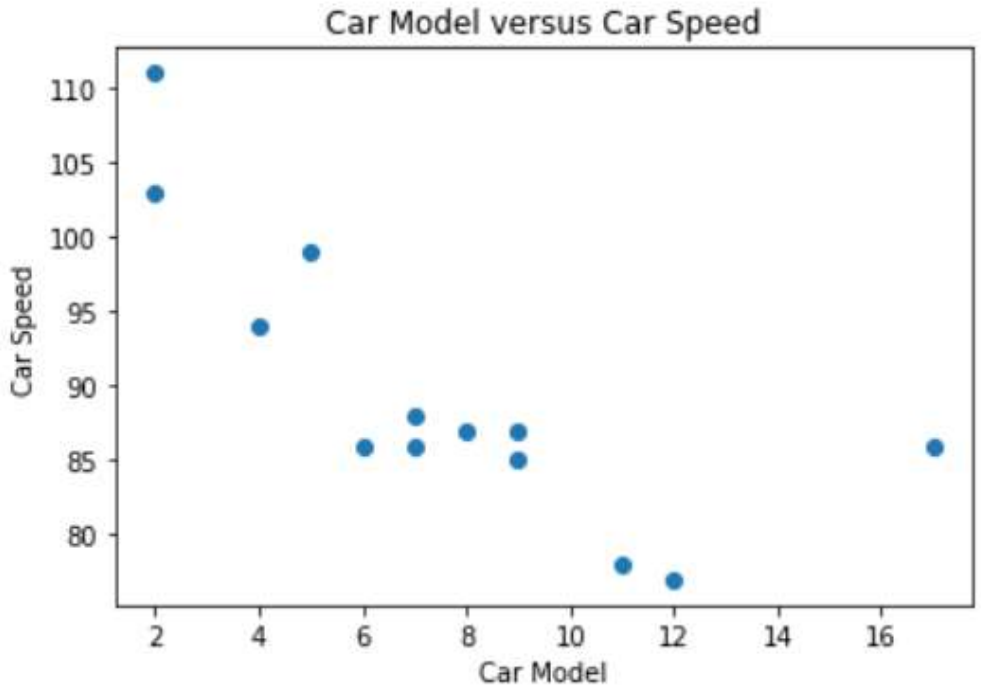
Matplotlib Scatter Plots

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

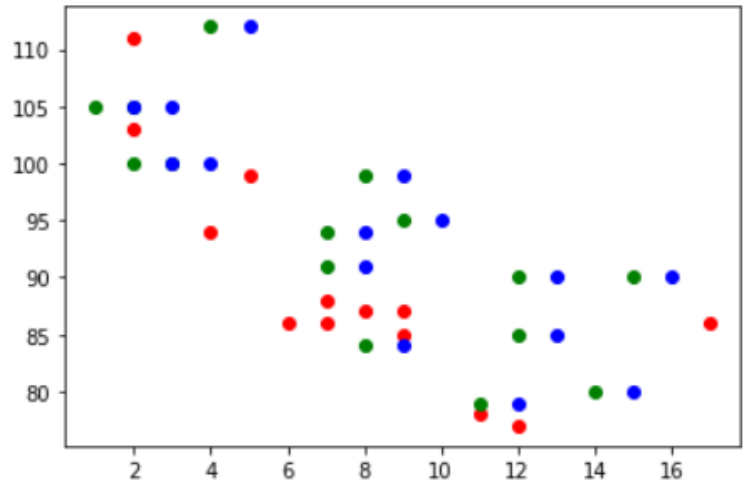
- The *scatter()* function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis:

- In this example, there seems to be a relationship between speed and model, but what if we plot the observations from another day as well? Will the scatter plot tell us something else?

```
import matplotlib.pyplot as plt
import numpy as np
car_model = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
car_speed = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(car_model, car_speed)
plt.xlabel("Car Model")
plt.ylabel("Car Speed")
plt.title("Car Model versus Car Speed")
plt.show()
```



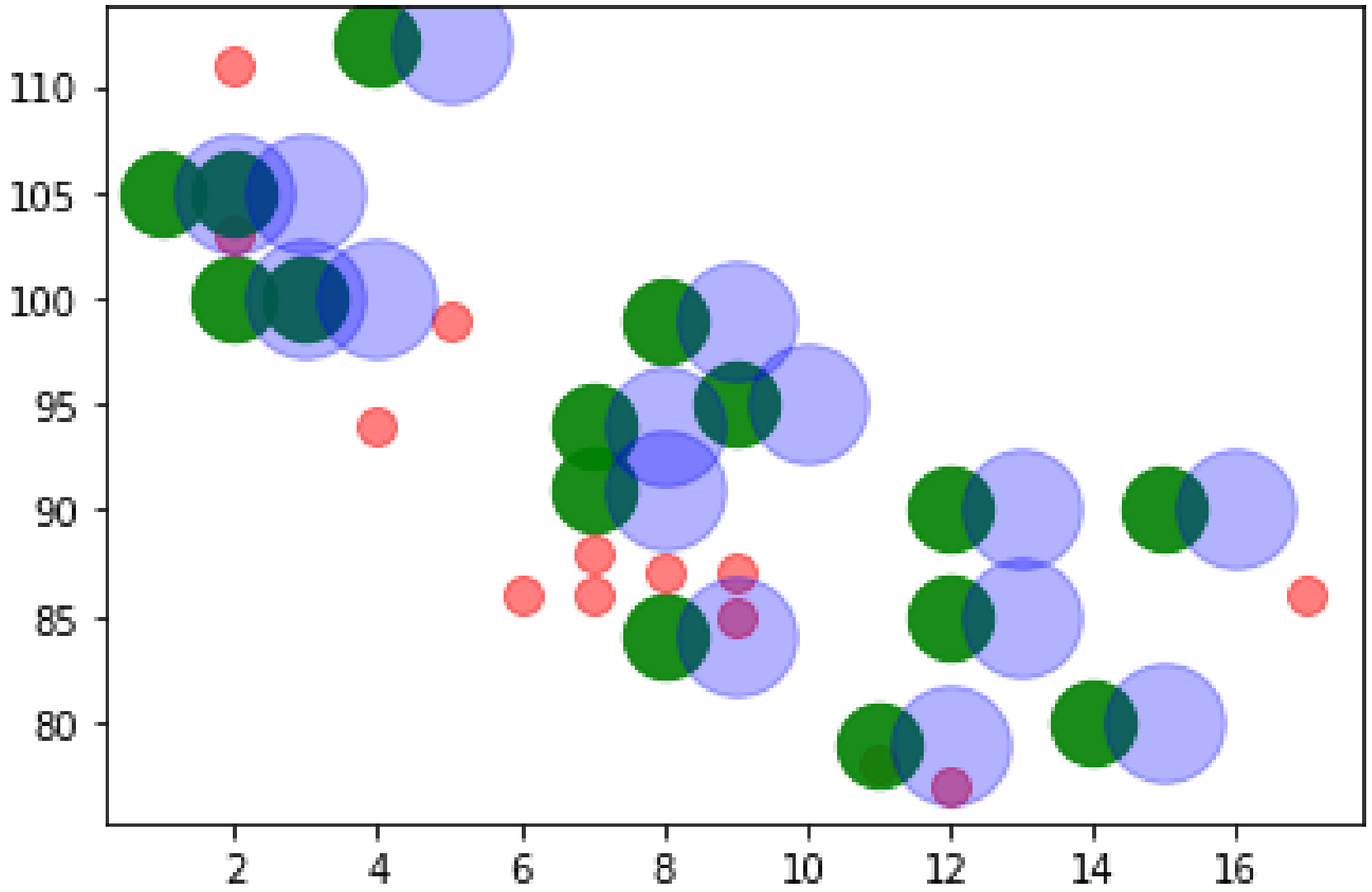
```python
import matplotlib.pyplot as plt
import numpy as np

#day one, the age and speed of 13 cars:
car_model = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
car_speed = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(car_model, car_speed,color = 'r')

#day two, the age and speed of 15 cars:
car_model = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
car_speed = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(car_model, car_speed , color = 'g')

#day three, the age and speed of 15 cars:
car_model = np.array([3,3,9,2,16,9,13,10,8,4,12,5,8,15,13])
car_speed = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(car_model, car_speed , color = 'b')

plt.show()
```



Matplotlib Scatter Plots

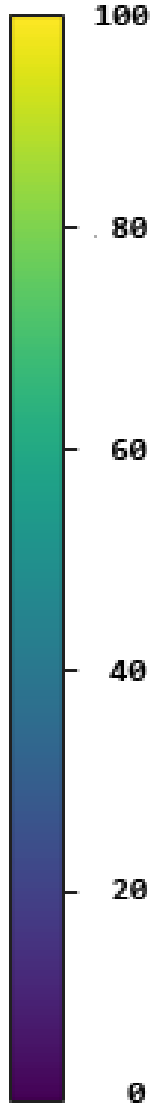
- You can change the *size* of the dots with the *s* argument.
- Just like colors, make sure the array for sizes has the same length as the arrays for the x- and y-axis:
- You can adjust the *transparency* of the dots with the *alpha* argument.
- Just like colors, make sure the array for sizes has the same length as the arrays for the x- and y-axis:

```
import matplotlib.pyplot as plt
import numpy as np
#day one, the age and speed of 13 cars:
car_model = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
car_speed = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(car_model, car_speed,color = 'r' , s = 100 , alpha=0.5)
#day two, the age and speed of 15 cars:
car_model = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
car_speed = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(car_model, car_speed , color = 'g' , s = 500 , alpha=0.9)
#day three, the age and speed of 15 cars:
car_model = np.array([3,3,9,2,16,9,13,10,8,4,12,5,8,15,13])
car_speed = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(car_model, car_speed , color = 'b' , s = 1000 , alpha=0.3)
plt.show()
```

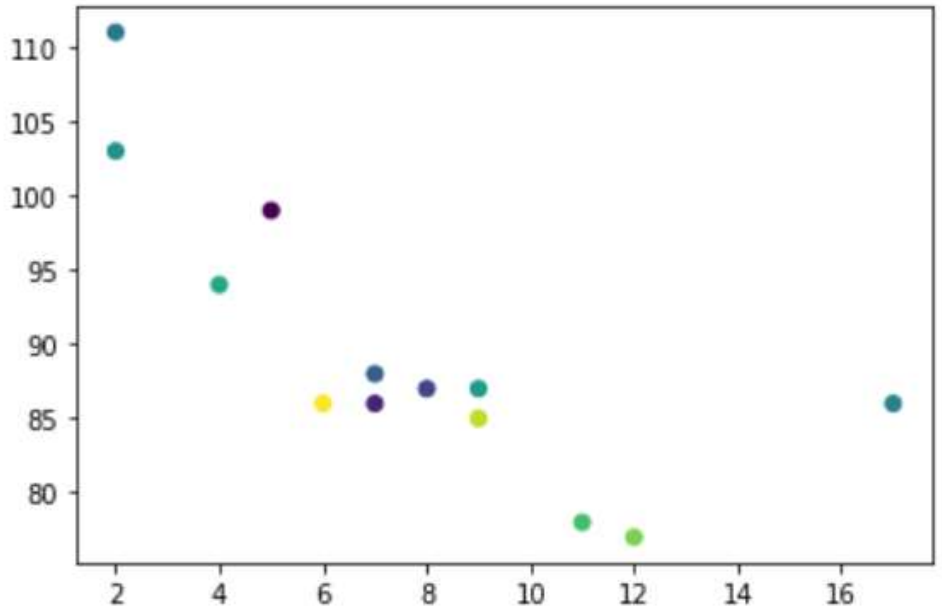

110
105
100
95
90
85
80
2
4
6
8
10
12
14
16

Matplotlib Scatter Plots

- The Matplotlib module has a number of available colormaps.
- A colormap is like a list of colors, where each color has a value that ranges from 0 to 100.
- Here is an example of a colormap:
- This colormap is called 'viridis' and as you can see it ranges from 0, which is a purple color, and up to 100, which is a yellow color.
- You can specify the colormap with the keyword argument cmap with the value of the colormap, in this case 'viridis' which is one of the built-in colormaps available in Matplotlib.
- In addition you have to create an array with values (from 0 to 100), one value for each of the point in the scatter plot:



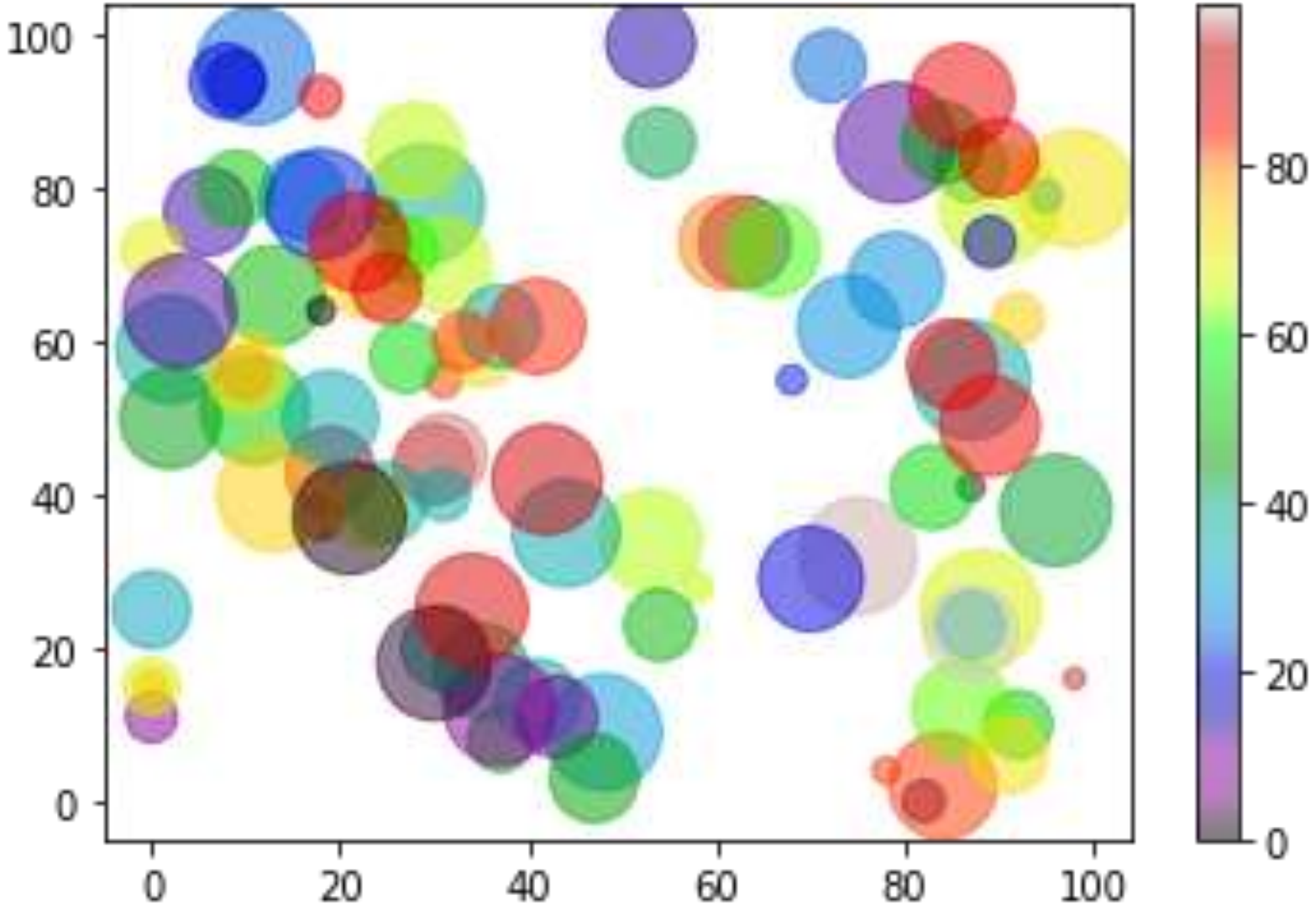
```
import matplotlib.pyplot as plt
import numpy as np

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])
colors = np.array([0, 10, 20, 30, 40, 45, 50, 55, 60, 70, 80, 90, 100])

plt.scatter(x, y, c=colors, cmap='viridis')

plt.show()
```



Combine Color Size and Alpha

You can combine a colormap with different sizes on the dots. This is best visualized if the dots are transparent:

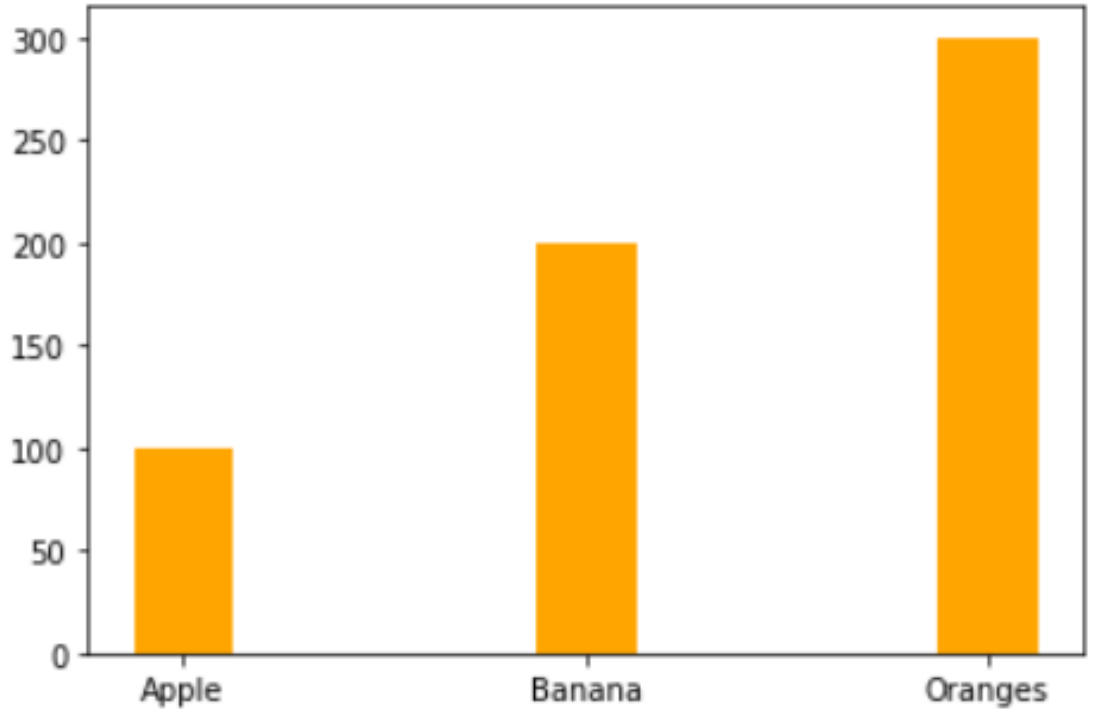
```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.randint(100, size=(100))
y = np.random.randint(100, size=(100))
colors = np.random.randint(100, size=(100))
sizes = 10 * np.random.randint(100, size=(100))

plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap='nipy_spectral')

plt.colorbar()

plt.show()
```


100
80
60
40
20
0
0
20
40
60
80
100
80
60
40
20
0

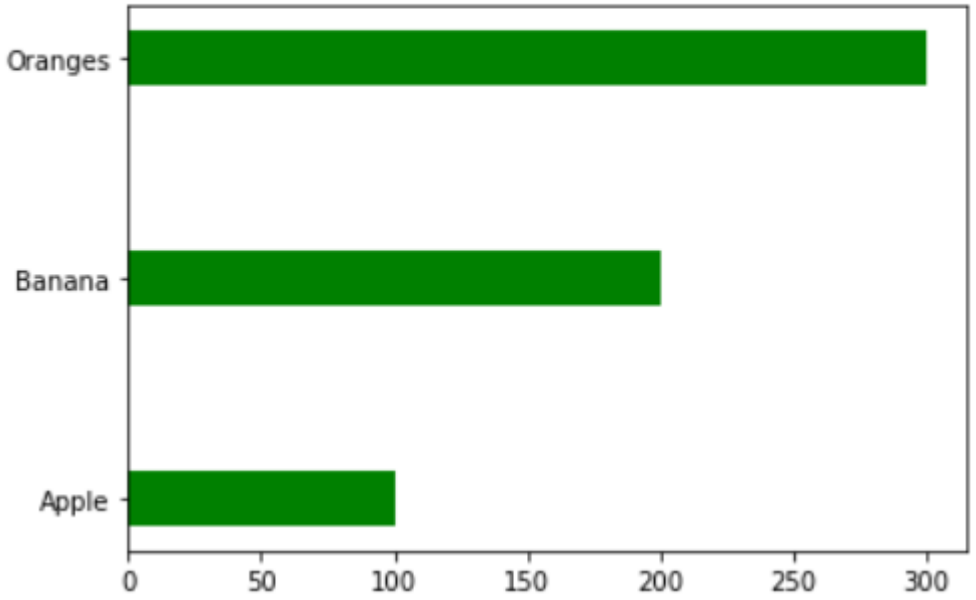
Matplotlib Bar Chart

- With Pyplot, you can use the *bar()* function to draw bar graphs.
- 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*.
- If you want the bars to be displayed *horizontally* instead of vertically, use the *barh()* function:
- The *bar()* and *barh()* takes the keyword argument *color* to set the color of the bars.
- You can use any of the *140 supported color* names you can use *Hexadecimal color* values.
- The *bar()* takes the keyword argument *width* to set the width of the bars.
- The *barh()* takes the keyword argument *height* to set the height of the bars.

```python
import matplotlib.pyplot as plt
import numpy as np
x = np.array (["Apple" , "Banana" , "Oranges"])
y = np.array ([100,200,300])
#plt.bar(x,y)
plt.bar(x,y,width=0.25,color = "orange")
plt.show()
```

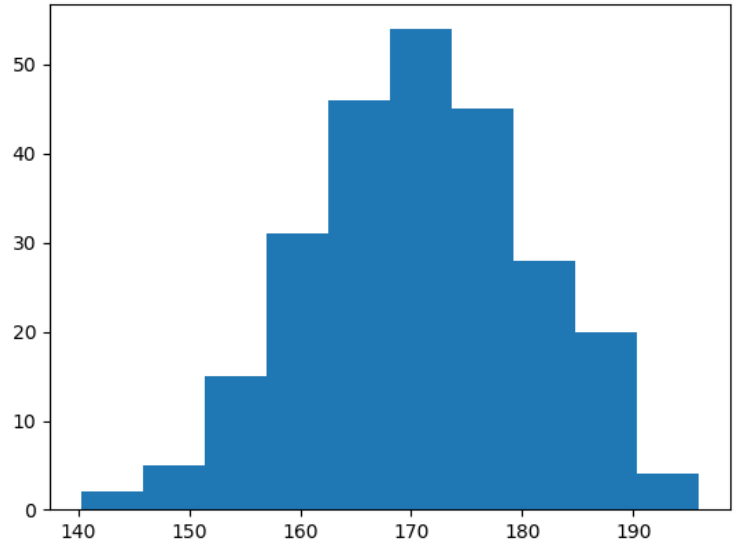



```python
plt.barh(x,y,color="green",height=0.25)
plt.show()
```



Matplotlib Histogram

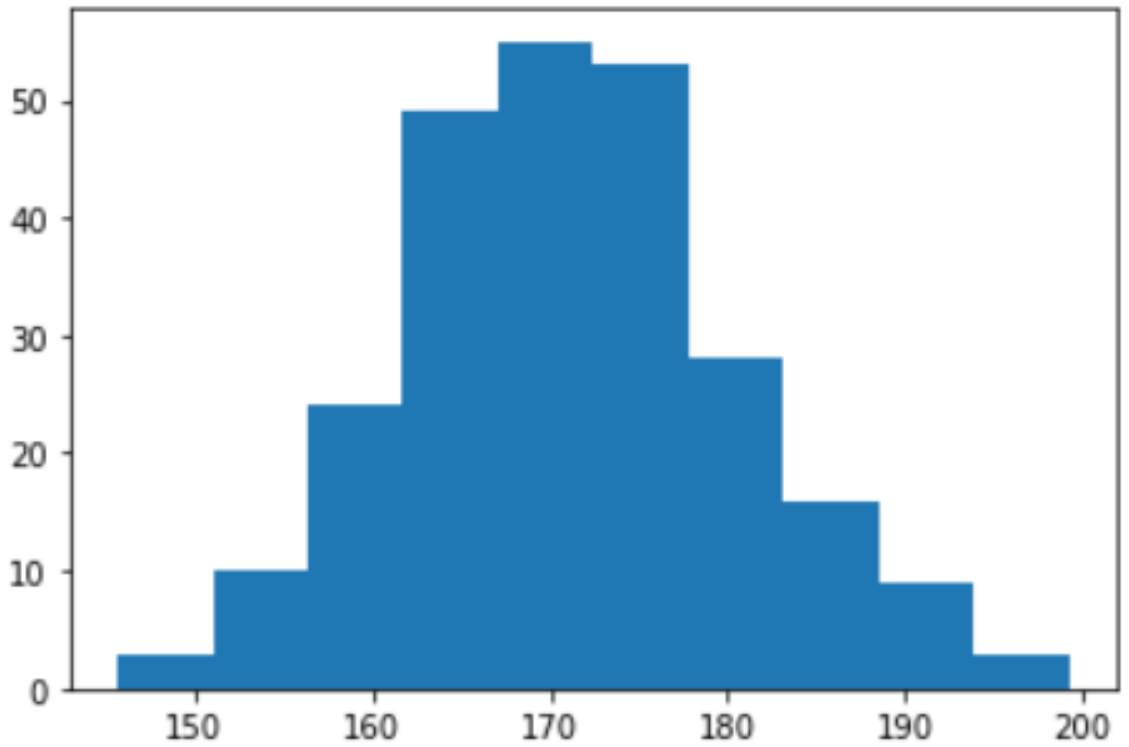
- A histogram is a graph showing frequency distributions.
- It is a graph showing the number of observations within each given interval.
- Example: Say you ask for the height of 250 people, you might end up with a histogram like this:
- You can read from the histogram that there are approximately:
  - 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



Matplotlib 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.
- The *hist()* function will read the array and produce a histogram:

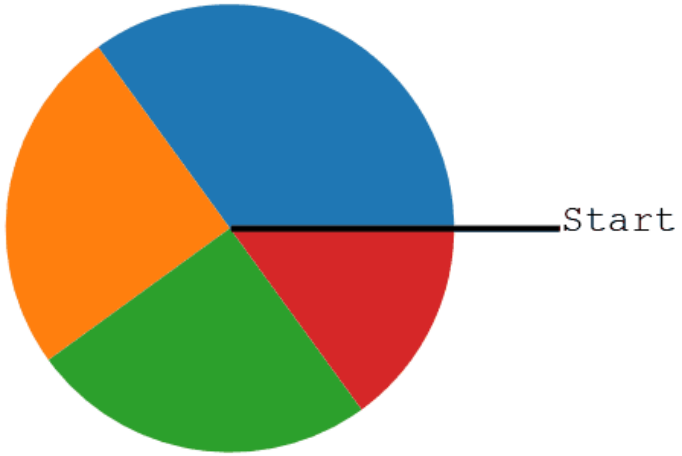
```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.normal(170, 10, 250)
plt.hist(x)
plt.show()
```



Matplotlib Pie Chart

- With Pyplot, you can use the *pie()* function to draw pie charts.
- 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.
- 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)*

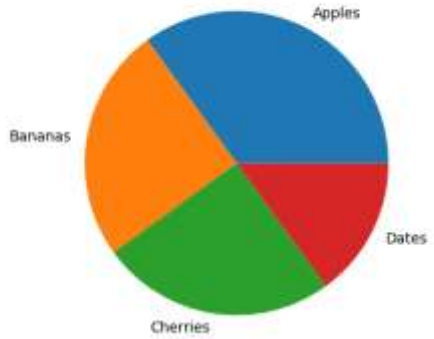
```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
plt.pie(y)
plt.show()
```



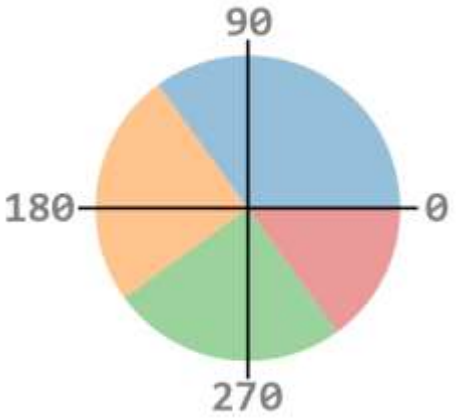
Matplotlib Pie Chart

- Add labels to the pie chart with the *labels* parameter.
- The label parameter must be an array with one label for each wedge:
- As mentioned the default start angle is at the x-axis, but you can change the start angle by specifying a *startangle* parameter.
- The *startangle* parameter is defined with an angle in degrees, default angle is 0.
- Maybe you want one of the wedges to stand out? The *explode* parameter allows you to do that.
- The *explode* parameter, if specified, and not None, must be an array with one value for each wedge.
- Each value represents how far from the center each wedge is displayed:
- Add a shadow to the pie chart by setting the *shadow* parameter to *True*
- You can set the color of each wedge with the *colors* parameter.
- The colors parameter, if specified, must be an array with one value for each wedge
- To add a distribution of numbers in a wedges, use the *autopct* parameter.
- To add a list of explanation for each wedge, use the *legend()* function.
- To add a header to the legend, add the *title* parameter to the *legend function*.

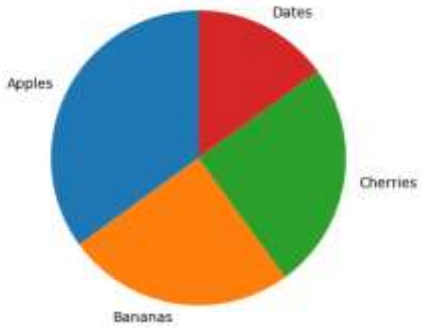
Default angle (0°)



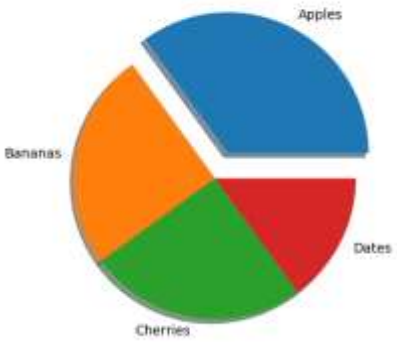
Angle Explanation



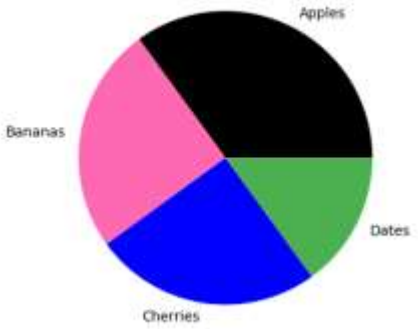
Angle (90°)

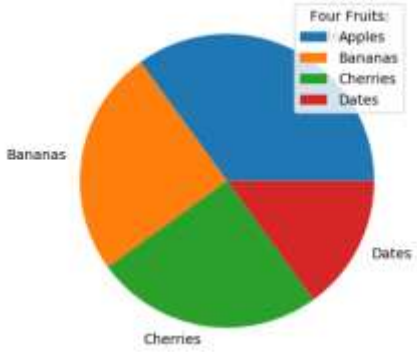


Explode and Shadow



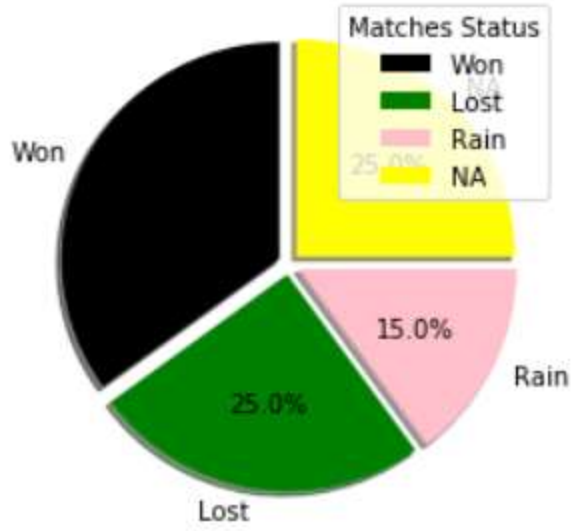
Colors





Legend with title

```
import matplotlib.pyplot as plt
import numpy as np
dist = np.array ([35,25,15,25])
labels = np.array (["Won","Lost","Rain","NA"])
explode = np.array ([0.05,0.05,0.05,0.05])
colors = np.array (["Black","Green","Pink","Yellow"])
plt.pie(dist,labels=labels,startangle=90,explode = explode , shadow = True, colors = colors,autopct='%1.1f%%')
plt.legend(title="Matches Status")
plt.show()
```



Class Activity

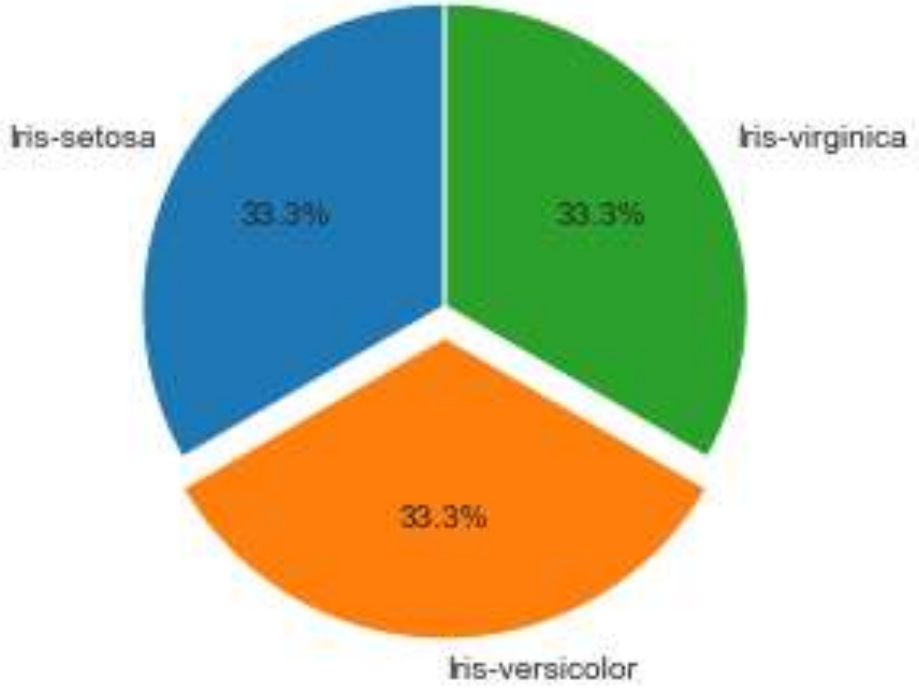
Working with Pie Chart using Iris Dataset

IRIS dataset

Iris Versicolor

Iris Setosa

Iris Virginica



	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
...	...	...	...	...	...	...
145	146	6.7	3.0	5.2	2.3	Iris-virginica
146	147	6.3	2.5	5.0	1.9	Iris-virginica
147	148	6.5	3.0	5.2	2.0	Iris-virginica
148	149	6.2	3.4	5.4	2.3	Iris-virginica
149	150	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 6 columns

```
######################################################################
# Pseudocode for the task
######################################################################
# Step 1: Read Iris file
# Step 2: Store all the instances of setosa class in df_setosa dataframe
# Step 3: Store all the instances of versicolor class in df_versicolor dataframe
# Step 4: Store all the instances of virginica class in df_virginica dataframe
# Step 5: store all the number of instances of df_setosa dataframe, df_versicolor dataframe, df_virginica dataframe in 'sizes' a
# Step 6: store all the distinct classes of Iris dataset in 'labels' array
# Step 7: draw pie chart using sizes, labels, explode and autopct parameters
# Step 8: show the chart
######################################################################
######################################################################
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
df = pd.read_csv("iris.csv")
# store all the instances of setosa class in 'setosa' dataframe
setosa = df[df['Species'] == 'Iris-setosa']
# store all the instances of versicolor class in 'versicolor' dataframe
versicolor = df[df['Species'] == 'Iris-versicolor']
# store all the instances of virginica class in 'virginica' dataframe
virginica = df[df['Species'] == 'Iris-virginica']
#store all the distinct classes in 'labels' array
labels = df['Species'].unique ()
#store all the number of instances of setosa, versicolor, and virginica classes in 'sizes' array
sizes = [setosa.shape [0] , versicolor.shape[0] , virginica.shape [0]]
#explode array for explode value between wedges
explode = np.array ([0 , 0.1 , 0])
#draw pie chart
plt.pie (sizes , explode = explode , labels = labels , autopct = '%1.1f%%')
#show pie chart
plt.show ()
```

Seaborn

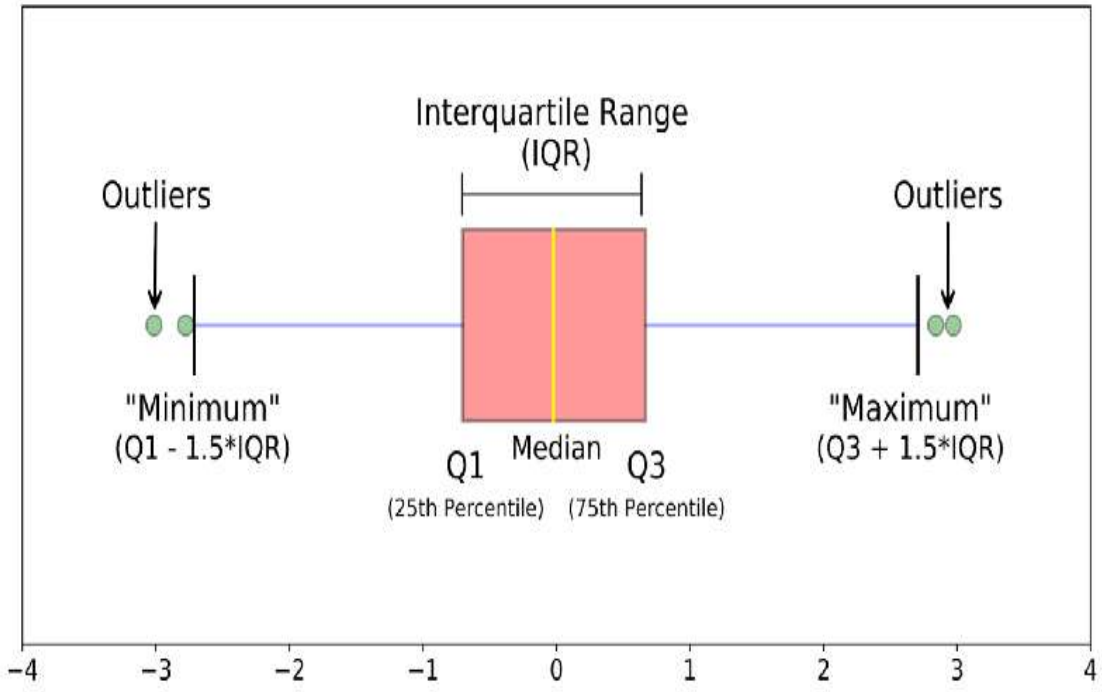
Searborn

- Seaborn is a library for making statistical graphics in Python.
- It is built on top of matplotlib.
- Installation

–pip install seaborn

–conda install seaborn

Box Plot



- Boxplots are a standardized way of displaying the distribution of data:

–“minimum”,

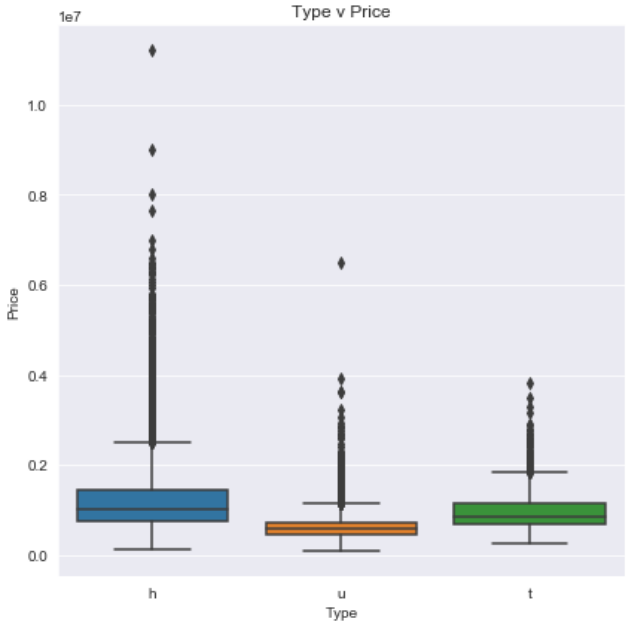
–first quartile (Q1),

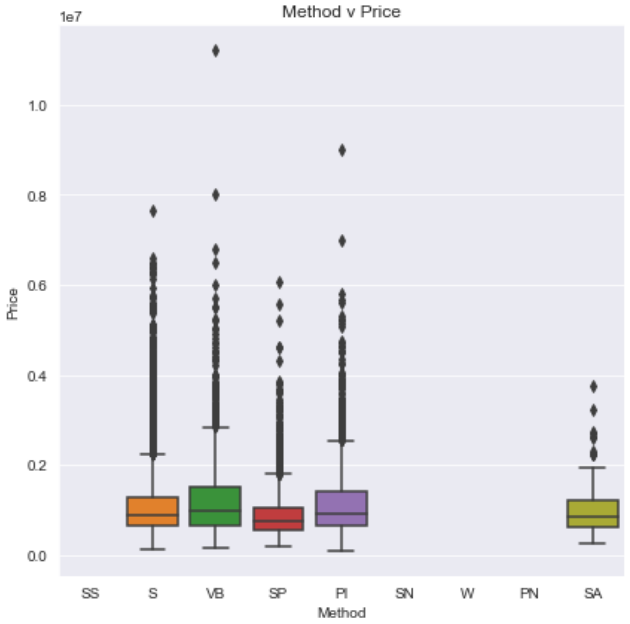
– median, t

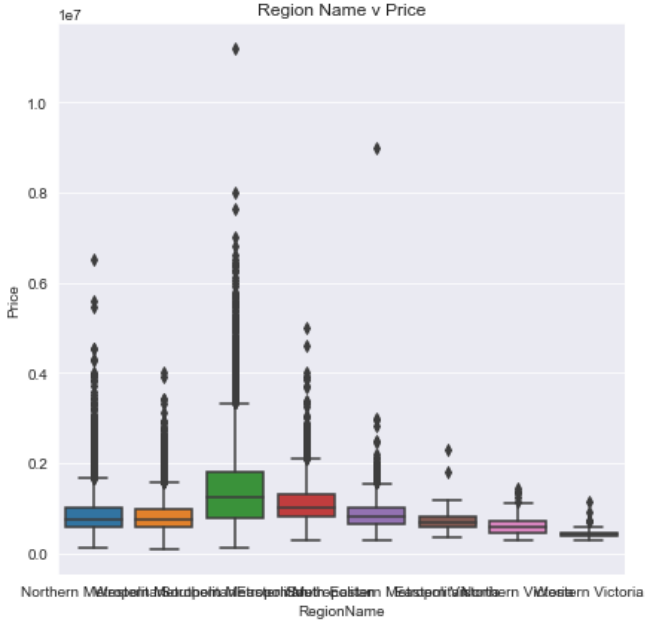
–third quartile (Q3)

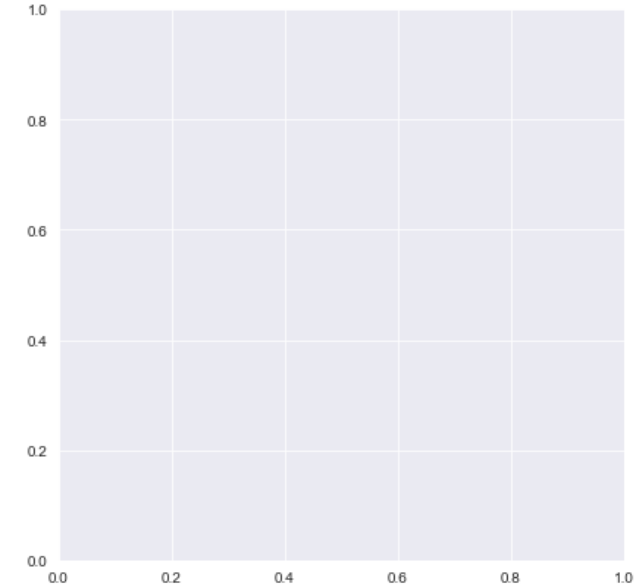
– “maximum”

Example 6


Type v Price
1e7
1.0
0.8
0.6
0.4
0.2
0.0
Price
h
u
t
Type


Method v Price
1e7
1.0
0.8
0.6
0.4
0.2
0.0
Price
SS
S
VB
SP
PI
SN
W
PN
SA
Method


Region Name v Price
1e7
1.0
0.8
0.6
0.4
0.2
0.0
Price
RegionName


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

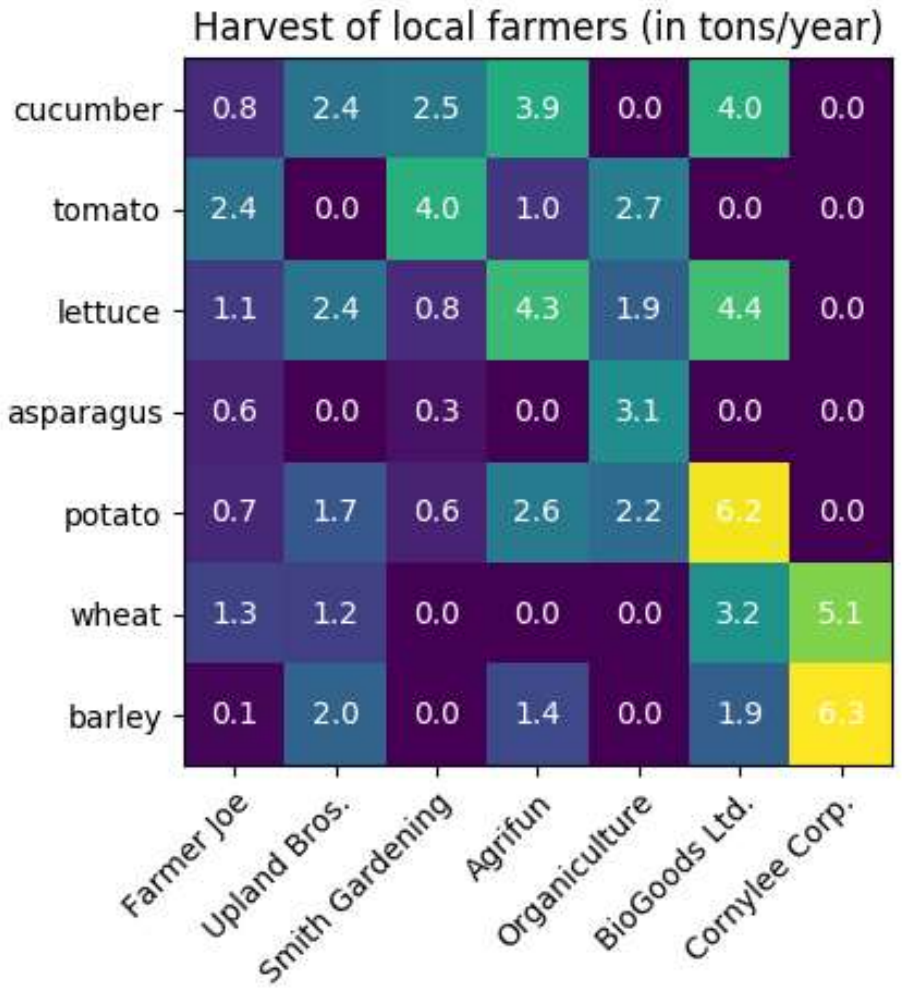
Example 6

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
df = pd.read_csv('Home_Prices.csv')
df.shape
df
```

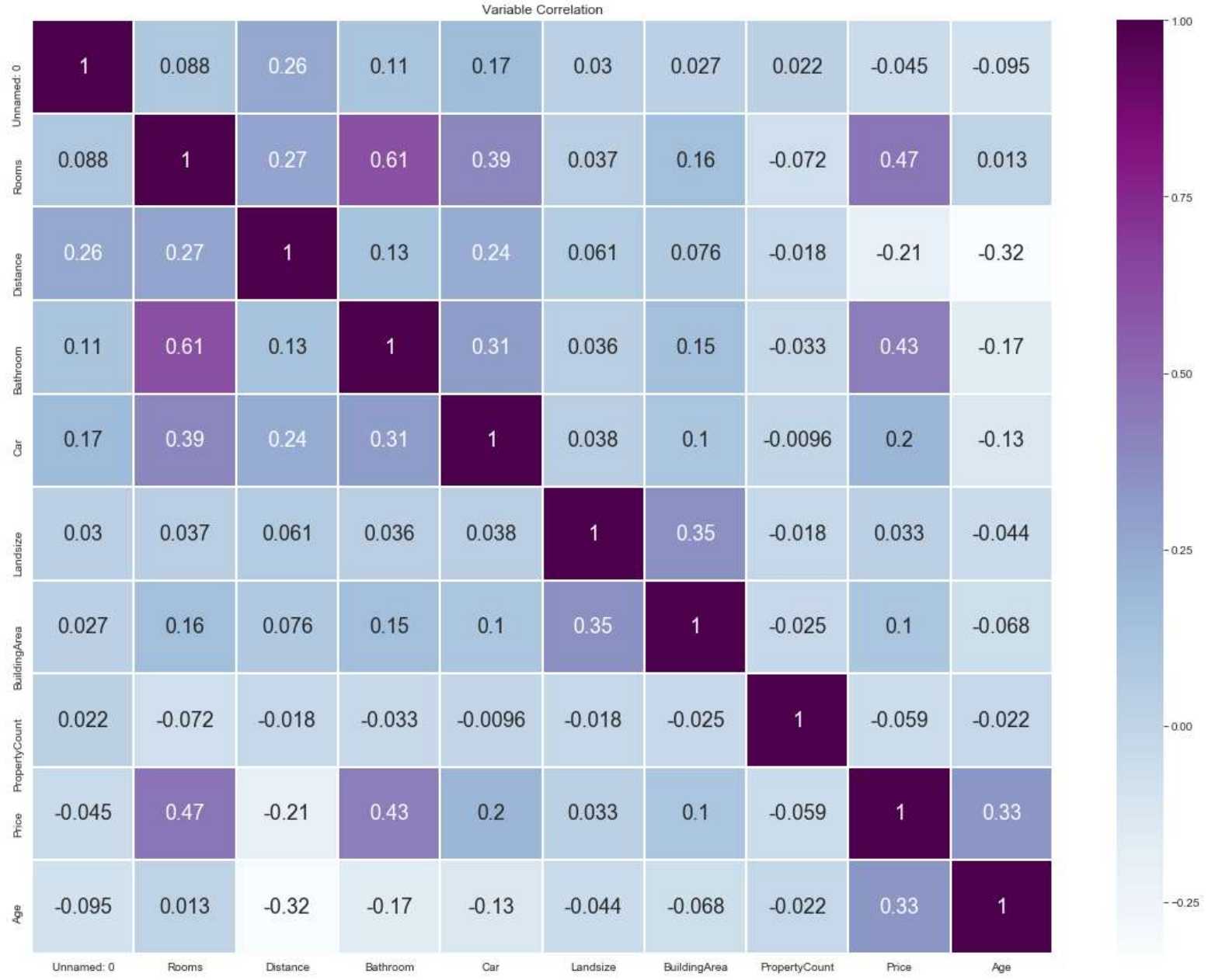
```
sns.set_style('darkgrid')
f, axes = plt.subplots(2,2, figsize = (15,15))

sns.boxplot(data = df, x = 'Type', y = 'Price', ax = axes[0,0])
axes[0,0].set_xlabel('Type')
axes[0,0].set_ylabel('Price')
axes[0,0].set_title('Type v Price')

sns.boxplot(x = 'Method', y = 'Price', data = df, ax = axes[0,1])
axes[0,1].set_xlabel('Method')
axes[0,1].set_ylabel('Price')
axes[0,1].set_title('Method v Price')

sns.boxplot(x = 'RegionName', y = 'Price', data = df, ax = axes[1,0])
axes[1,0].set_xlabel('RegionName')
axes[1,0].set_ylabel('Price')
axes[1,0].set_title('Region Name v Price')

plt.show()
```

Heatmap

- The heatmap is a way of representing the data in a 2-dimensional form.
- The data values are represented as colors in the graph.
- The goal of the heatmap is to provide a colored visual summary of information.



Example 7


Variable Correlation
Unnamed: 0
Rooms
Distance
Bathroom
Car
Landsize
BuildingArea
PropertyCount
Price
Age
1 0.088 0.26 0.11 0.17 0.03 0.027 0.022 -0.045 -0.095
0.088 1 0.27 0.61 0.39 0.037 0.16 -0.072 0.47 0.013
0.26 0.27 1 0.13 0.24 0.061 0.076 -0.018 -0.21 -0.32
0.11 0.61 0.13 1 0.31 0.036 0.15 -0.033 0.43 -0.17
0.17 0.39 0.24 0.31 1 0.038 0.1 -0.0096 0.2 -0.13
0.03 0.037 0.061 0.036 0.038 1 0.35 -0.018 0.033 -0.044
0.027 0.16 0.076 0.15 0.1 0.35 1 -0.025 0.1 -0.068
0.022 -0.072 -0.018 -0.033 -0.0096 -0.018 -0.025 1 -0.059 -0.022
-0.045 0.47 -0.21 0.43 0.2 0.033 0.1 -0.059 1 0.33
-0.095 0.013 -0.32 -0.17 -0.13 -0.044 -0.068 -0.022 0.33 1
Unnamed: 0
Rooms
Distance
Bathroom
Car
Landsize
BuildingArea
PropertyCount
Price
Age
1.00
0.75
0.50
0.25
0.00
−0.25

Example 7

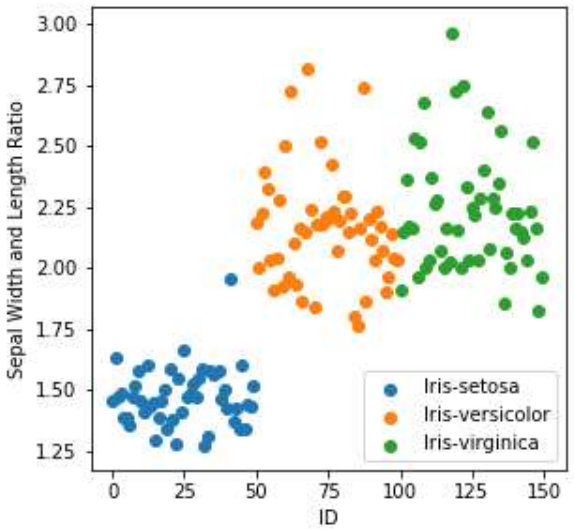
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

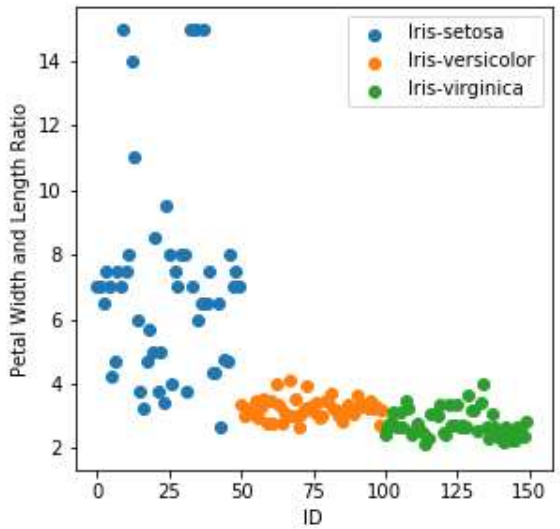
```
df = pd.read_csv('Home_Prices.csv')
df.shape
df.head ()
```

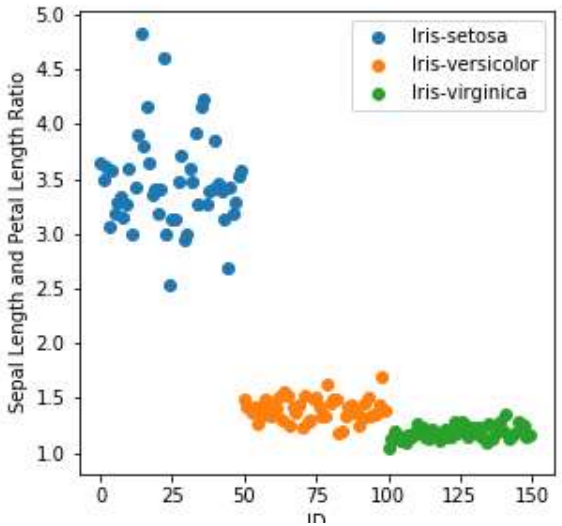
```
plt.figure(figsize=(20,15))
sns.heatmap(df.corr(),cmap = 'BuPu',linewidth = 1,annot= True, annot_kws={"size": 18})
#cmap values can be (Blues, Greens, coolwarm...)
plt.title('Variable Correlation')
```

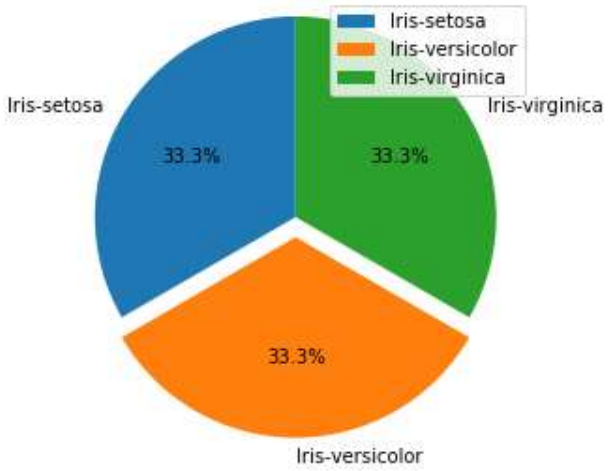
Class Assignment

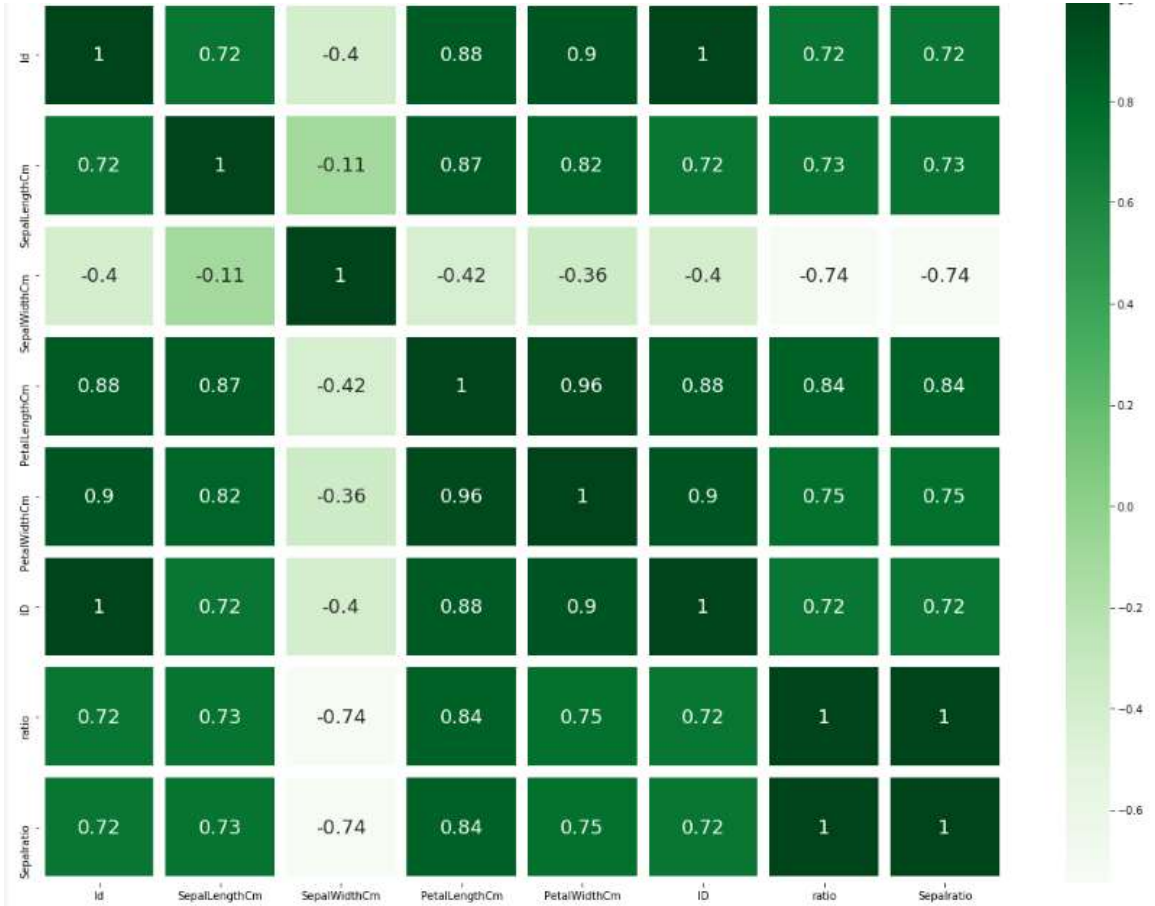
- Create a graph having four subplots using IRIS dataset. The first subplot should show scatter graph showing the id on x-axis and the ratio of sepal width and sepal width on y-axis of all three species. The second subplot should show scatter graph showing the id on x-axis and the ratio of sepal width and sepal width on y-axis of all three species. The third subplot should show scatter graph showing the id on x-axis and the ratio of sepal width and sepal width on y-axis of all three species. The fourth subplot should show the distribution of instances of all three species in pie chart. In addition, also create a heatmap to show the correlation between the sepal length, sepal width, petal length and petal width. Save both the diagrams and send with code to you instructor email id and upload on google classroom.


Sepal Width and Length Ratio
ID
Iris-setosa
Iris-versicolor
Iris-virginica


Petal Width and Length Ratio
ID
Iris-setosa
Iris-versicolor
Iris-virginica


Sepal Length and Petal Length Ratio
ID
Iris-setosa
Iris-versicolor
Iris-virginica


Iris-setosa
Iris-versicolor
Iris-virginica
33.3%
33.3%
33.3%


Id 1 0.72 -0.4 0.88 0.9 1 0.72 0.72
SepalLengthCm 0.72 1 -0.11 0.87 0.82 0.72 0.73 0.73
SepalWidthCm -0.4 -0.11 1 -0.42 -0.36 -0.4 -0.74 -0.74
PetalLengthCm 0.88 0.87 -0.42 1 0.96 0.88 0.84 0.84
PetalWidthCm 0.9 0.82 -0.36 0.96 1 0.9 0.75 0.75
ID 1 0.72 -0.4 0.88 0.9 1 0.72 0.72
ratio 0.72 0.73 -0.74 0.84 0.75 0.72 1 1
Sepalratio 0.72 0.73 -0.74 0.84 0.75 0.72 1 1
Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm ID ratio Sepalratio