MARUDHAR KESARI JAIN COLLEGE FOR WOMEN, VANIYAMBADI

PG DEPARTMENT OF COMPUTER APPLICATIONS

Subject Name : PYTHON PROGRAMMING

CLASS : I-MCA

SUBJECT CODE : 23PCA13

Unit IV

Working with Python Packages: NumPy Library-Ndarray Basic Operations Indexing, Slicing and Iteration Array manipulation - Pandas The Series The DataFrame - The Index Objects Data Vizualization with Matplotlib The Matplotlib Architecture pyplot The Plotting Window Adding Elements to the Chart Line Charts Bar Charts Pie charts.

Python - Packages

We organize a large number of files in different folders and subfolders based on some criteria, so that we can find and manage them easily. In the same way, a package in Python takes the concept of the modular approach to next logical level. As you know, a module can contain multiple objects, such as classes, functions, etc. A package can contain one or more relevant modules. Physically, a package is actually a folder containing one or more module files.

Let's create a package named mypackage, using the following steps:

- Create a new folder named D:\MyApp.
- Inside MyApp, create a subfolder with the name 'mypackage'.
- Create an empty __init__.py file in the mypackage folder.
- Using a Python-aware editor like IDLE, create modules greet.py and functions.py with the following code:

greet.py
Copy
```
def SayHello(name):
    print("Hello ", name)
```
functions.py
Copy
```
def sum(x,y):
    return x+y

def average(x,y):
    return (x+y)/2

def power(x,y):
    return x**y
```

That's it. We have created our package called mypackage. The following is a folder structure:

Package Folder Structure

Importing a Module from a Package

Now, to test our package, navigate the command prompt to the MyApp folder and invoke the Python prompt from there.

```
D:\MyApp>python
```

Import the functions module from the mypackage package and call its power() function.

```
>>> from mypackage import functions
>>> functions.power(3,2)
9
```

It is also possible to import specific functions from a module in the package.

_init__.py

The package folder contains a special file called __init__.py, which stores the package's content. It serves two purposes:

1. The Python interpreter recognizes a folder as the package if it contains __init__.py file.
2. __init__.py exposes specified resources from its modules to be imported.

An empty __init__.py file makes all functions from the above modules available when this package is imported. Note that __init__.py is essential for the folder to be recognized by Python as a package. You can optionally define functions from individual modules to be made available.

The __init__.py file is normally kept empty. However, it can also be used to choose specific functions from modules in the package folder and make them available for import. Modify __init__.py as below:

__init__.py
 Copy
```
from .functions import average, power
from .greet import SayHello
```

The specified functions can now be imported in the interpreter session or another executable script.

Create test.py in the MyApp folder to test mypackage.

test.py
 Copy
```
from mypackage import power, average, SayHello
SayHello()
x=power(3,2)
print("power(3,2) : ", x)
```

Note that functions power() and SayHello() are imported from the package and not from their respective modules, as done earlier. The output of the above script is:

```
D:\MyApp>python test.py
Hello world
power(3,2) : 9
```

Install a Package Globally

Once a package is created, it can be installed for system-wide use by running the setup script. The script calls setup() function from the setuptools module.

Let's install mypackage for system-wide use by running a setup script.

Save the following code as setup.py in the parent folder MyApp. The script calls the setup() function from the setuptools module. The setup() function takes various arguments such as name, version, author, list of dependencies, etc. The zip_safe argument defines whether the package is installed in compressed mode or regular mode.

Example: setup.py
 Copy

```
from setuptools import setup
setup(name='mypackage',
version='0.1',
description='Testing installation of Package',
url='#',
author='auth',
author_email='author@email.com',
license='MIT',
packages=['mypackage'],
zip_safe=False)
```

Now execute the following command to install mypackage using the pip utility. Ensure that the command prompt is in the parent folder, in this case D:\MyApp.

```
D:\MyApp>pip install mypackage
Processing d:\MyApp
Installing collected packages: mypack
Running setup.py install for mypack ... done
Successfully installed mypackage-0.1
```

Now mypackage is available for system-wide use and can be imported in any script or interpreter.

```
D:\>python
>>> import mypackage
>>>mypackage.average(10,20)
15.0
>>>mypackage.power(10,2)
100
```

You may also want to publish the package for public use. PyPI (stands for Python Package Index) is a repository of Python packages. Visit https://packaging.python.org/distributing to know more about the procedure of uploading a package to PyPI.

What is NumPy?

NumPy stands for numeric python which is a python package for the computation and processing of the multidimensional and single dimensional array elements.

What is NumPy

NumPy stands for numeric python which is a python package for the computation and processing of the multidimensional and single dimensional array elements.

Travis Oliphant created NumPy package in 2005 by injecting the features of the ancestor module Numeric into another module Numarray.

It is an extension module of Python which is mostly written in C. It provides various functions which are capable of performing the numeric computations with a high speed.

NumPy provides various powerful data structures, implementing multi-dimensional arrays and matrices. These data structures are used for the optimal computations regarding arrays and matrices.

In this tutorial, we will go through the numeric python library NumPy.

The need of NumPy

With the revolution of data science, data analysis libraries like NumPy, SciPy, Pandas, etc. have seen a lot of growth. With a much easier syntax than other programming languages, python is the first choice language for the data scientist.

NumPy provides a convenient and efficient way to handle the vast amount of data. NumPy is also very convenient with Matrix multiplication and data reshaping. NumPy is fast which makes it reasonable to work with a large set of data.

There are the following advantages of using NumPy for data analysis.

1. NumPy performs array-oriented computing.
2. It efficiently implements the multidimensional arrays.
3. It performs scientific computations.
4. It is capable of performing Fourier Transform and reshaping the data stored in multidimensional arrays.
5. NumPy provides the in-built functions for linear algebra and random number generation.

Nowadays, NumPy in combination with SciPy and Mat-plotlib is used as the replacement to MATLAB as Python is more complete and easier programming language than MATLAB

NumPy Environment Setup

NumPy doesn't come bundled with Python. We have to install it using **the python pip** installer. Execute the following command.

```
$ pip install numpy
```

It is best practice to install NumPy with the full SciPy stack. The binary distribution of the SciPy stack is specific to the operating systems.

Windows

On the Windows operating system, The SciPy stack is provided by the Anaconda which is a free distribution of the Python SciPy package.

It can be downloaded from the official website: https://www.anaconda.com/. It is also available for Linux and Mac.

Backward Skip 10sPlay VideoForward Skip 10s

The CanoPy also comes with the full SciPy stack which is available as free as well as commercial license. We can download it by visiting the link: https://www.enthought.com/products/canopy/

The Python (x, y) is also available free with the full SciPy distribution. Download it by visiting the link: https://python-xy.github.io/

Linux

In Linux, the different package managers are used to install the SciPy stack. The package managers are specific to the different distributions of Linux. Let's look at each one of them.

Ubuntu

Execute the following command on the terminal.

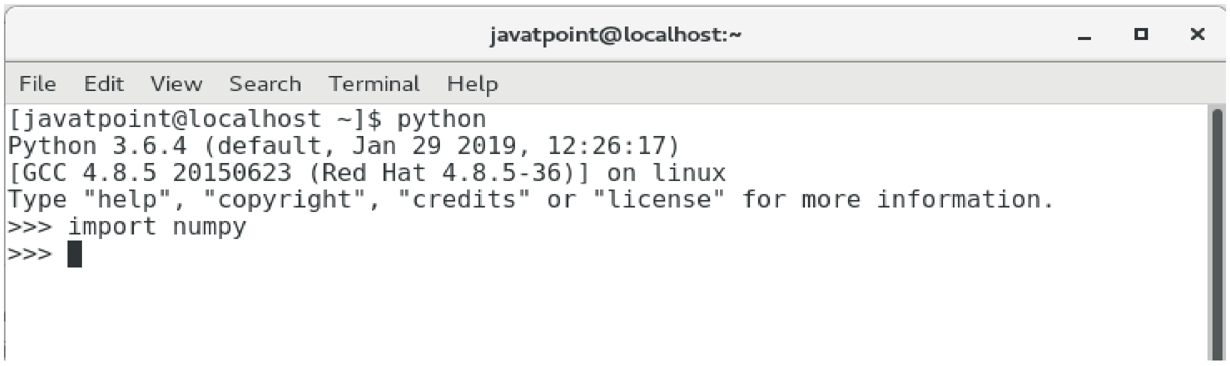
```
$ sudo apt-get install python-numpy

$ python-scipy python-matplotlibipythonipythonnotebook python-pandas

$ python-sympy python-nose
```

Redhat

On Redhat, the following commands are executed to install the Python SciPy package stack.

```
$ sudo yum install numpyscipy python-matplotlibipython

$ python-pandas sympy python-nose atlas-devel
```

To verify the installation, open the Python prompt by executing python command on the terminal (cmd in the case of windows) and try to import the module NumPy as shown in the below image. If it doesn't give the error, then it is installed successfully.



```
[javatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> 
```

NumPy Ndarray

Ndarray is the n-dimensional array object defined in the numpy which stores the collection of the similar type of elements. In other words, we can define a ndarray as the collection of the data type (dtype) objects.

The ndarray object can be accessed by using the 0 based indexing. Each element of the Array object contains the same size in the memory.

Creating a ndarray object

The ndarray object can be created by using the array routine of the numpy module. For this purpose, we need to import the numpy.

1. >>> a = numpy.array

Consider the below image.

PauseNext

Unmute

Current TimeÂ 0:00

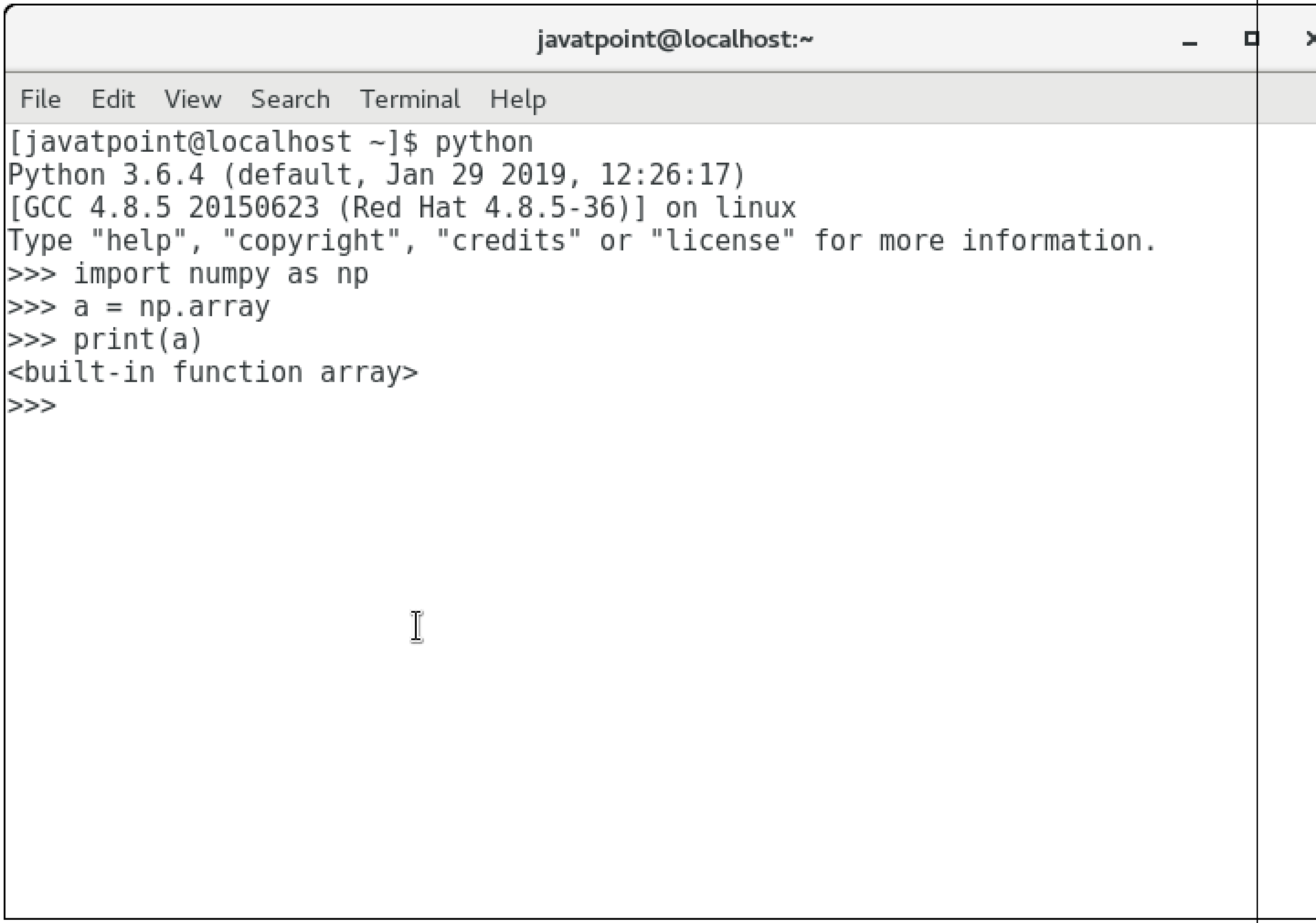
/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen



```
[javatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a = np.array
>>> print(a)
<built-in function array>
>>>
```

We can also pass a collection object into the array routine to create the equivalent n-dimensional array. The syntax is given below.

1. >>> numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)

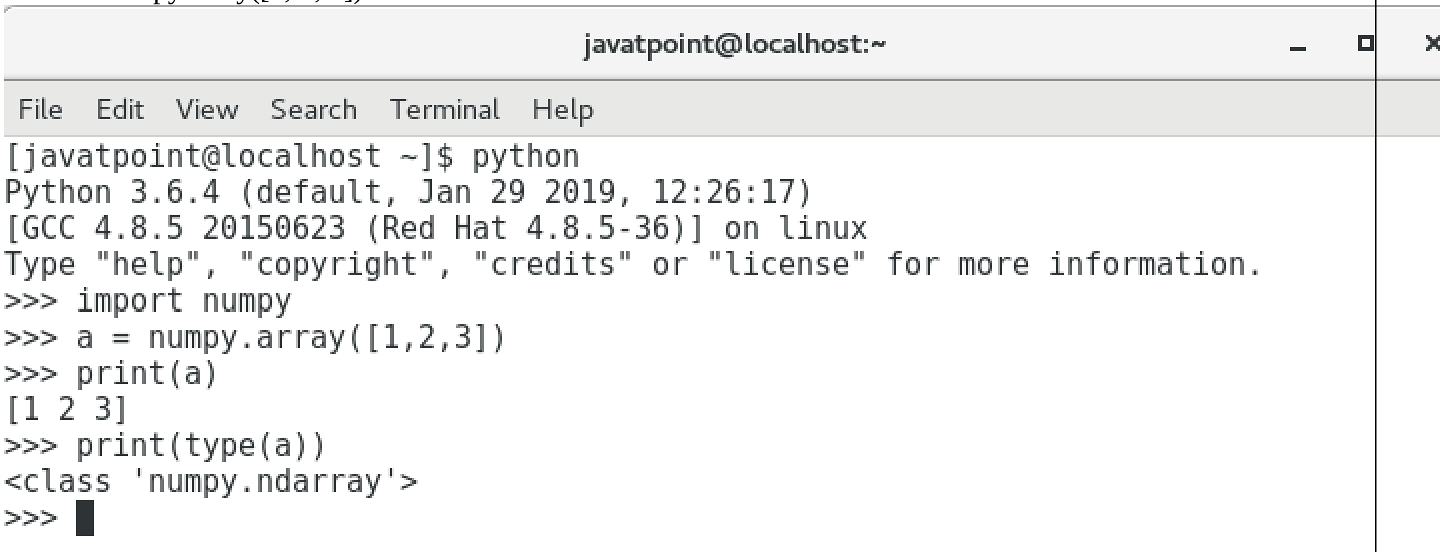
The parameters are described in the following table.

SN	Parameter	Description
1	object	It represents the collection object. It can be a list, tuple, dictionary, set, etc.

2	dtype	We can change the data type of the array elements by changing this option to the specified type. The default is none.
3	copy	It is optional. By default, it is true which means the object is copied.
4	order	There can be 3 possible values assigned to this option. It can be C (column order), R (row order), or A (any)
5	subok	The returned array will be base class array by default. We can change this to make the subclasses passes through by setting this option to true.
6	ndmin	It represents the minimum dimensions of the resultant array.

To create an array using the list, use the following syntax.

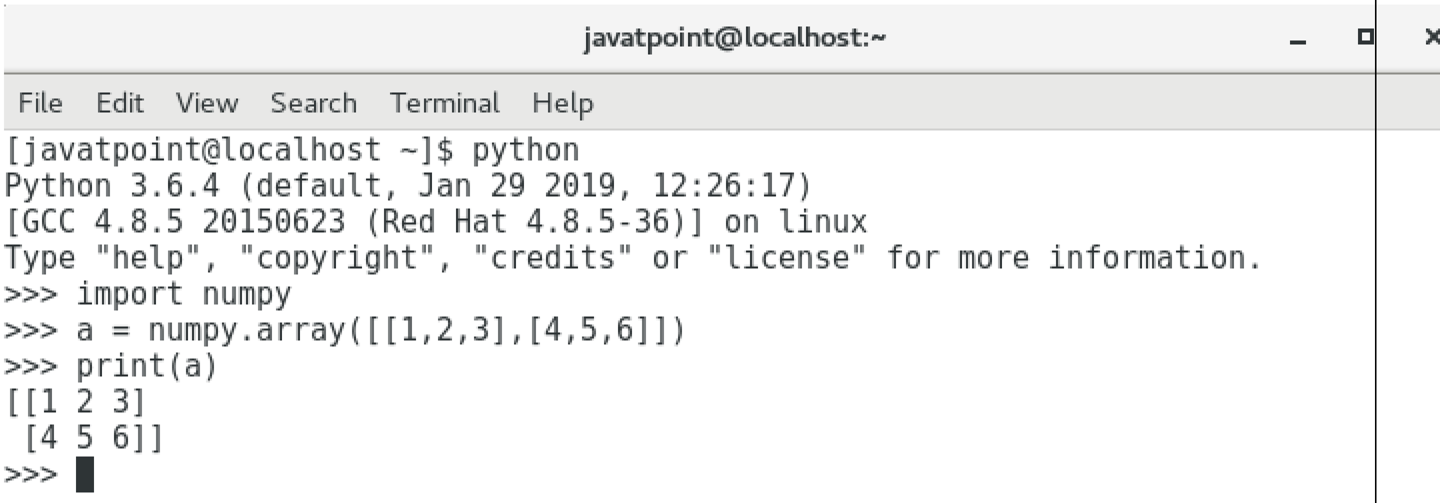
1. >>> a = numpy.array([1, 2, 3])



```
[javatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> a = numpy.array([1,2,3])
>>> print(a)
[1 2 3]
>>> print(type(a))
<class 'numpy.ndarray'>
>>> 
```

To create a multi-dimensional array object, use the following syntax.

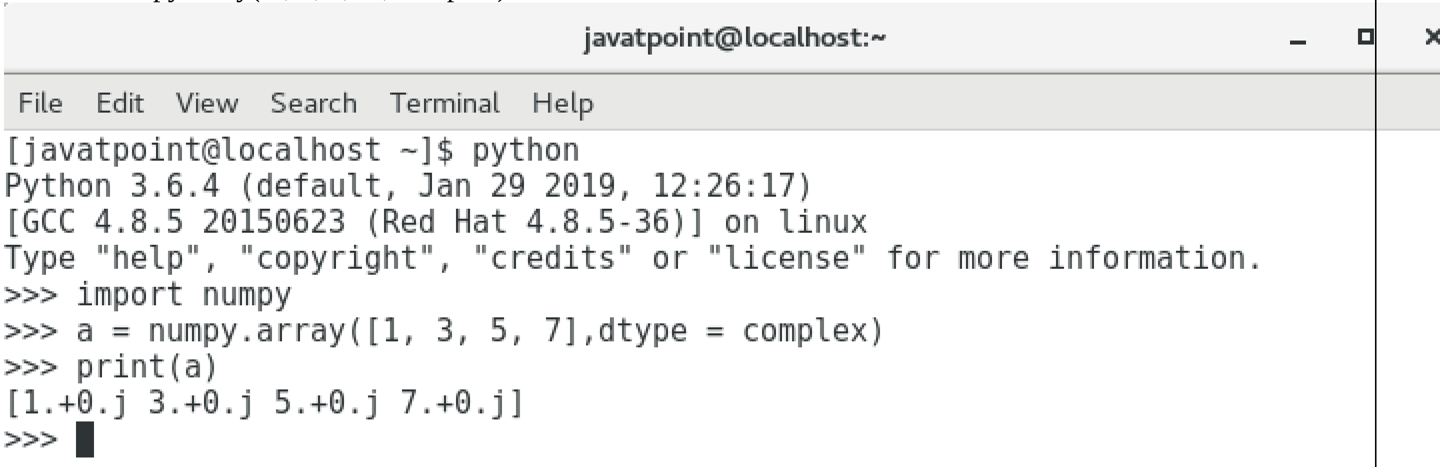
1. >>> a = numpy.array([[1, 2, 3], [4, 5, 6]])



```
[javatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> a = numpy.array([[1,2,3],[4,5,6]])
>>> print(a)
[[1 2 3]
 [4 5 6]]
>>> 
```

To change the data type of the array elements, mention the name of the data type along with the collection.

1. >>> a = numpy.array([1, 3, 5, 7], complex)

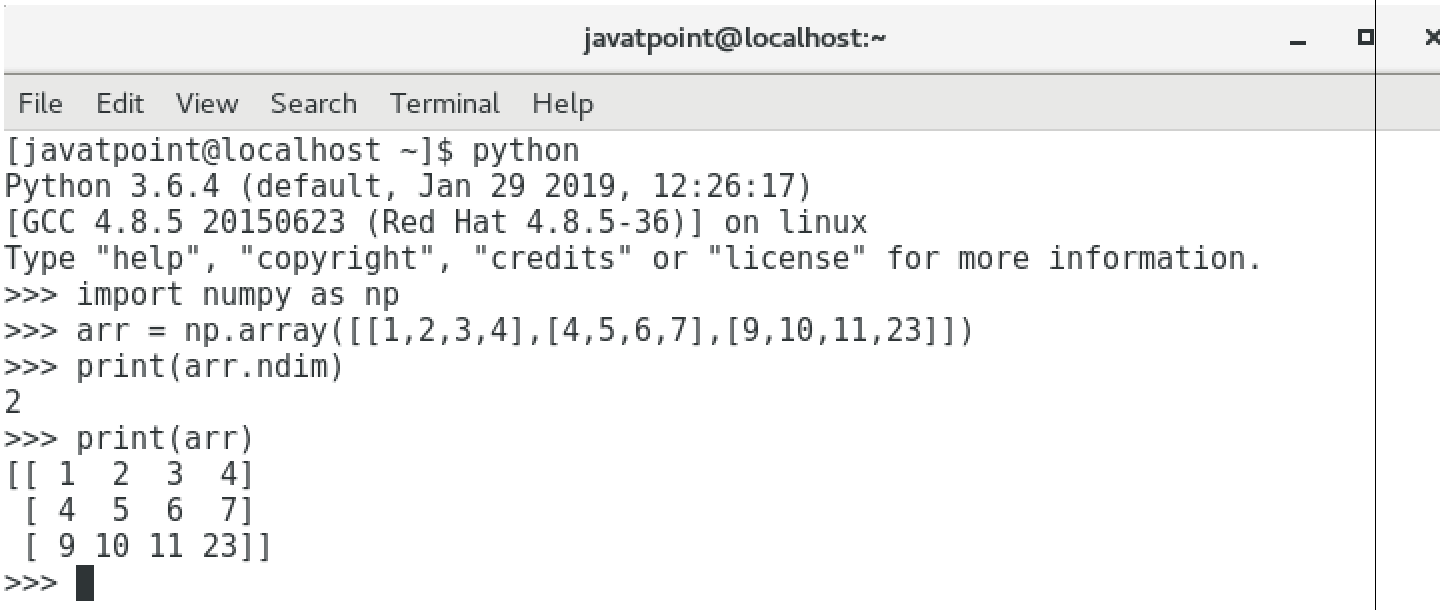


```
[javatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> a = numpy.array([1, 3, 5, 7],dtype = complex)
>>> print(a)
[1.+0.j 3.+0.j 5.+0.j 7.+0.j]
>>> 
```

Finding the dimensions of the Array

The **ndim** function can be used to find the dimensions of the array.

1. >>> **import** numpy as np
2. >>> arr = np.array([[1, 2, 3, 4], [4, 5, 6, 7], [9, 10, 11, 23]])
3.
4. >>> **print**(arr.ndim)



```
[javatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> arr = np.array([[1,2,3,4],[4,5,6,7],[9,10,11,23]])
>>> print(arr.ndim)
2
>>> print(arr)
[[ 1  2  3  4]
 [ 4  5  6  7]
 [ 9 10 11 23]]
>>> 
```

Finding the size of each array element

The itemsize function is used to get the size of each array item. It returns the number of bytes taken by each array element.

Consider the following example.

Example

1. #finding the size of each item in the array
2. **import** numpy as np
3. a = np.array([[1,2,3]])
4. **print**("Each item contains",a.itemsize,"bytes")

Output:

```
Each item contains 8 bytes.
```

Finding the data type of each array item

To check the data type of each array item, the dtype function is used. Consider the following example to check the data type of the array items.

Example

1. #finding the data type of each array item
2. **import** numpy as np

```
3. a = np.array([[1,2,3]])
4. print("Each item is of the type",a.dtype)
```

Output:

```
Each item is of the type int64
```

Finding the shape and size of the array

To get the shape and size of the array, the size and shape function associated with the numpy array is used.

Consider the following example.

Example

```
1. import numpy as np
2. a = np.array([[1,2,3,4,5,6,7]])
3. print("Array Size:",a.size)
4. print("Shape:",a.shape)
```

Output:

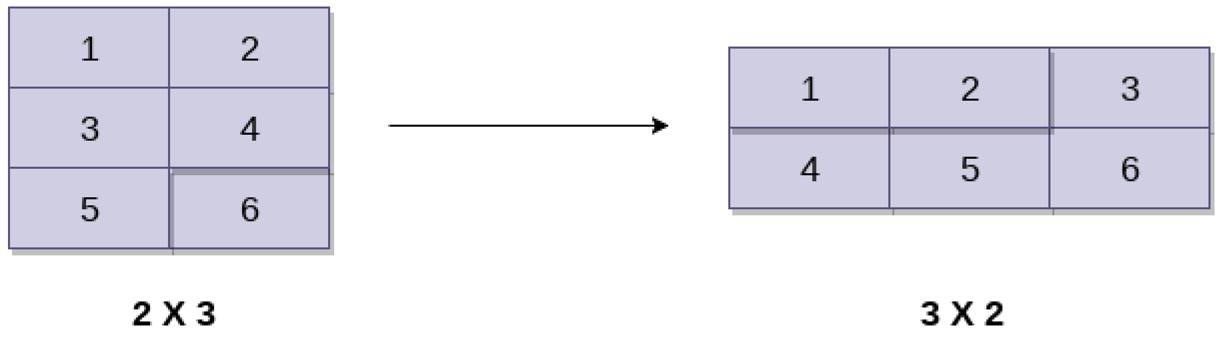
```
Array Size: 7
Shape: (1, 7)
```

Reshaping the array objects

By the shape of the array, we mean the number of rows and columns of a multi-dimensional array. However, the numpy module provides us the way to reshape the array by changing the number of rows and columns of the multi-dimensional array.

The reshape() function associated with the ndarray object is used to reshape the array. It accepts the two parameters indicating the row and columns of the new shape of the array.

Let's reshape the array given in the following image.



Example

1. **import** numpy as np
2. a = np.array([[1,2],[3,4],[5,6]])
3. **print**("printing the original array..")
4. **print**(a)
5. a=a.reshape(2,3)
6. **print**("printing the reshaped array..")
7. **print**(a)

Output:

```
printing the original array..
[[1 2]
 [3 4]
 [5 6]]
printing the reshaped array..
[[1 2 3]
 [4 5 6]]
```

Slicing in the Array

Slicing in the NumPy array is the way to extract a range of elements from an array. Slicing in the array is performed in the same way as it is performed in the python list.

Consider the following example to print a particular element of the array.

Example

1. **import** numpy as np
2. a = np.array([[1,2],[3,4],[5,6]])
3. **print**(a[0,1])
4. **print**(a[2,0])

Output:

```
2
5
```

The above program prints the 2nd element from the 0th index and 0th element from the 2nd index of the array.

Linspace

The linspace() function returns the evenly spaced values over the given interval. The following example returns the 10 evenly separated values over the given interval 5-15

Example

```
import numpy as np
a=np.linspace(5,15,10) #prints 10 values which are evenly spaced over the given interval 5-15
print(a)
```

Output:

```
[ 5.         6.11111111  7.22222222  8.33333333  9.44444444 10.55555556
 11.66666667 12.77777778 13.88888889 15.        ]
```

Finding the maximum, minimum, and sum of the array elements

The NumPy provides the max(), min(), and sum() functions which are used to find the maximum, minimum, and sum of the array elements respectively.

Consider the following example.

Example

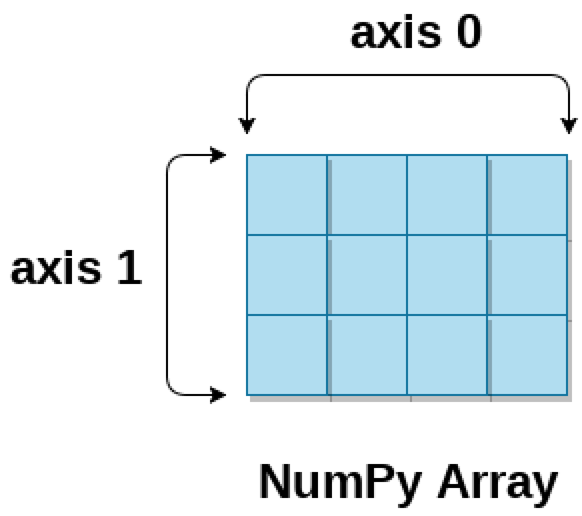
```
import numpy as np
a = np.array([1,2,3,10,15,4])
print("The array:",a)
print("The maximum element:",a.max())
print("The minimum element:",a.min())
print("The sum of the elements:",a.sum())
```

Output:

```
The array: [ 1  2  3 10 15  4]
The maximum element: 15
The minimum element: 1
The sum of the elements: 35
```

NumPy Array Axis

A NumPy multi-dimensional array is represented by the axis where axis-0 represents the columns and axis-1 represents the rows. We can mention the axis to perform row-level or column-level calculations like the addition of row or column elements.



To calculate the maximum element among each column, the minimum element among each row, and the addition of all the row elements, consider the following example.

Example

```
import numpy as np
a = np.array([[1,2,30],[10,15,4]])
print("The array:",a)
print("The maximum elements of columns:",a.max(axis = 0))
print("The minimum element of rows",a.min(axis = 1))
print("The sum of all rows",a.sum(axis = 1))
```

Output:

```
The array: [[1  2  30]
                  [10  15  4]]
The maximum elements of columns: [10 15 30]
The minimum element of rows [1 4]
The sum of all rows [33 29]
```

Finding square root and standard deviation

The sqrt() and std() functions associated with the numpy array are used to find the square root and standard deviation of the array elements respectively.

Standard deviation means how much each element of the array varies from the mean value of the numpy array.

Consider the following example.

Example

```
import numpy as np
a = np.array([[1,2,30],[10,15,4]])
print(np.sqrt(a))
print(np.std(a))
```

Output:

```
[[1.         1.41421356 5.47722558]
 [3.16227766 3.87298335 2.        ]]
10.044346115546242
```

Arithmetic operations on the array

The numpy module allows us to perform the arithmetic operations on multi-dimensional arrays directly.

In the following example, the arithmetic operations are performed on the two multi-dimensional arrays a and b.

Example

```
import numpy as np
a = np.array([[1,2,30],[10,15,4]])
b = np.array([[1,2,3],[12, 19, 29]])
print("Sum of array a and b\n",a+b)
print("Product of array a and b\n",a*b)
print("Division of array a and b\n",a/b)
```

Array Concatenation

The numpy provides us with the vertical stacking and horizontal stacking which allows us to concatenate two multi-dimensional arrays vertically or horizontally.

Consider the following example.

Example

```
import numpy as np
a = np.array([[1,2,30],[10,15,4]])
b = np.array([[1,2,3],[12, 19, 29]])
print("Arrays vertically concatenated\n",np.vstack((a,b)));
print("Arrays horizontally concatenated\n",np.hstack((a,b)))
```

Output:

```
Arrays vertically concatenated
 [[ 1  2 30]
 [10 15  4]
 [ 1  2  3]
 [12 19 29]]
Arrays horizontally concatenated
 [[ 1  2 30  1  2  3]
 [10 15  4 12 19 29]]
```

Python Operators

The operator is a symbol that performs a specific operation between two operands, according to one definition. Operators serve as the foundation upon which logic is constructed in a program in a particular programming language. In every programming language, some operators perform several tasks. Same as other languages, Python also has some operators, and these are given below –

- Arithmetic operators
- Comparison operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators
- Arithmetic Operators

Arithmetic Operators

Arithmetic operators used between two operands for a particular operation. There are many arithmetic operators. It includes the exponent (**) operator as well as the + (addition), - (subtraction), * (multiplication), / (divide), % (reminder), and // (floor division) operators.

Consider the following table for a detailed explanation of arithmetic operators.

Operator	**Description**
+ (Addition)	It is used to add two operands. For example, if a = 10, b = 10 => a+b = 20

- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if a = 20, b = 5 => a - b = 15
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0
*** (Multiplication)**	It is used to multiply one operand with the other. For example, if a = 20, b = 4 => a * b = 80
% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0
**** (Exponent)**	As it calculates the first operand's power to the second operand, it is an exponent operator.
// (Floor division)	It provides the quotient's floor value, which is obtained by dividing the two operands.

Program Code:

Now we give code examples of arithmetic operators in Python. The code is given below -

```
a = 32    # Initialize the value of a
b = 6      # Initialize the value of b
print('Addition of two numbers:',a+b)
print('Subtraction of two numbers:',a-b)
print('Multiplication of two numbers:',a*b)
print('Division of two numbers:',a/b)
print('Reminder of two numbers:',a%b)
print('Exponent of two numbers:',a**b)
print('Floor division of two numbers:',a//b)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Addition of two numbers: 38
Subtraction of two numbers: 26
Multiplication of two numbers: 192
Division of two numbers: 5.333333333333333
Reminder of two numbers: 2
Exponent of two numbers: 1073741824
Floor division of two numbers: 5
```

Comparison operator

Comparison operators mainly use for comparison purposes. Comparison operators compare the values of the two operands and return a true or false Boolean value in accordance. The example of comparison operators are ==, !=, <=, >=, >, <. In the below table, we explain the works of the operators.

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal, then the condition becomes true.
<=	The condition is met if the first operand is smaller than or equal to the second operand.
>=	The condition is met if the first operand is greater than or equal to the second operand.
>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

Program Code:

Now we give code examples of Comparison operators in Python. The code is given below -

```
a = 32      # Initialize the value of a
b = 6       # Initialize the value of b
print('Two numbers are equal or not:',a==b)
print('Two numbers are not equal or not:',a!=b)
print('a is less than or equal to b:',a<=b)
print('a is greater than or equal to b:',a>=b)
print('a is greater b:',a>b)
print('a is less than b:',a<b)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Two numbers are equal or not: False
Two numbers are not equal or not: True
a is less than or equal to b: False
a is greater than or equal to b: True
a is greater b: True
a is less than b: False
```

Assignment Operators

Using the assignment operators, the right expression's value is assigned to the left operand. There are some examples of assignment operators like =, +=, -=, *=, %=, **=, //=. In the below table, we explain the works of the operators.

Operator	**Description**
=	It assigns the value of the right expression to the left operand.
+=	By multiplying the value of the right operand by the value of the left operand, the left operand receives a changed value. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

Program Code:

Now we give code examples of Assignment operators in Python. The code is given below -

```
a = 32         # Initialize the value of a
b = 6          # Initialize the value of b
print('a=b:', a==b)
print('a+=b:', a+b)
print('a-=b:', a-b)
print('a*=b:', a*b)
print('a%=b:', a%b)
print('a**=b:', a**b)
print('a//=b:', a//b)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
a=b: False
a+=b: 38
a-=b: 26
a*=b: 192
a%=b: 2
a**=b: 1073741824
a//=b: 5
```

Bitwise Operators

The two operands' values are processed bit by bit by the bitwise operators. The examples of Bitwise operators are bitwise OR (|), bitwise AND (&), bitwise XOR (^), negation (~), Left shift (<<), and Right shift (>>). Consider the case below.

For example,

```
if a = 7
   b = 6
then, binary (a) = 0111
    binary (b) = 0110

```

6. hence, a & b = 0011
7. a | b = 0111
8. a ^ b = 0100
9. ~ a = 1000
10. Let, Binary of x = 0101
11. Binary of y = 1000
12. Bitwise OR = 1101
13. 8 4 2 1
14. 1 1 0 1 = 8 + 4 + 1 = 13
15.
16. Bitwise AND = 0000
17. 0000 = 0
18.
19. Bitwise XOR = 1101
20. 8 4 2 1
21. 1 1 0 1 = 8 + 4 + 1 = 13
22. Negation of x = ~x = (-x) - 1 = (-5) - 1 = -6
23. ~x = -6

In the below table, we are explaining the works of the bitwise operators.

Operator	**Description**
& (binary and)	A 1 is copied to the result if both bits in two operands at the same location are 1. If not, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1.

^ (binary xor)	If the two bits are different, the outcome bit will be 1, else it will be 0.
~ (negation)	The operand's bits are calculated as their negations, so if one bit is 0, the next bit will be 1, and vice versa.
<< (left shift)	The number of bits in the right operand is multiplied by the leftward shift of the value of the left operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

Program Code:

Now we give code examples of Bitwise operators in Python. The code is given below -

```
a = 5          # initialize the value of a
b = 6          # initialize the value of b
print('a&b:', a&b)
print('a|b:', a|b)
print('a^b:', a^b)
print('~a:', ~a)
print('a<<b:', a<<b)
print('a>>b:', a>>b)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
a&b: 4
a|b: 7
a^b: 3
~a: -6
a<>b: 0
```

Logical Operators

The assessment of expressions to make decisions typically uses logical operators. The examples of logical operators are and, or, and not. In the case of logical AND, if the first one is 0, it does not depend upon the second one. In the case of logical OR, if the first one is 1, it does not depend on the second one. Python supports the following logical operators. In the below table, we explain the works of the logical operators.

Operator	Description
and	The condition will also be true if the expression is true. If the two expressions a and b are the same, then a and b must both be true.
or	The condition will be true if one of the phrases is true. If a and b are the two expressions, then an or b must be true if and is true and b is false.
not	If an expression **a** is true, then not (a) will be false and vice versa.

Program Code:

Now we give code examples of arithmetic operators in Python. The code is given below -

```
a = 5          # initialize the value of a
print(Is this statement true?:',a > 3 and a < 5)
print('Any one statement is true?:',a > 3 or a < 5)
print('Each statement is true then return False and vice-versa:',(not(a > 3 and a < 5)))
```

Output:

Now we give code examples of Bitwise operators in Python. The code is given below -

```
Is this statement true?: False
Any one statement is true?: True
Each statement is true then return False and vice-versa: True
```

Membership Operators

The membership of a value inside a Python data structure can be verified using Python membership operators. The result is true if the value is in the data structure; otherwise, it returns false.

Operator	Description
in	If the first operand cannot be found in the second operand, it is evaluated to be true (list, tuple, or dictionary).
not in	If the first operand is not present in the second operand, the evaluation is true (list, tuple, or dictionary).

Program Code:

Now we give code examples of Membership operators in Python. The code is given below -

```
x = ["Rose", "Lotus"]
print(' Is value Present?', "Rose" in x)
print(' Is value not Present?', "Riya" not in x)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Is value Present? True
Is value not Present? True
```

Identity Operators

Operator	**Description**
is	If the references on both sides point to the same object, it is determined to be true.
is not	If the references on both sides do not point at the same object, it is determined to be true.

Program Code:

Now we give code examples of Identity operators in Python. The code is given below -

```
a = ["Rose", "Lotus"]
b = ["Rose", "Lotus"]
c = a
print(a is c)
print(a is not c)
print(a is b)
print(a is not b)
print(a == b)
print(a != b)
```

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
True
False
False
True
True
False
```

Operator Precedence

The order in which the operators are examined is crucial to understand since it tells us which operator needs to be considered first. Below is a list of the Python operators' precedence tables.

Operator	**Description**
**	Overall other operators employed in the expression, the exponent operator is given precedence.
~ + -	the minus, unary plus, and negation.
* / % //	the division of the floor, the modules, the division, and the multiplication.
+ -	Binary plus, and minus
>> <<	Left shift. and right shift
&	Binary and.
^ |	Binary xor, and or
<= < > >=	Comparison operators (less than, less than equal to, greater than, greater then equal to).
<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Python List index() Method

Python **index()** method returns index of the passed element. This method takes an argument and returns index of it. If the element is not present, it raises a ValueError.

Scientific computing and web development are just a few of the many uses for Python, a powerful and adaptable programming language. The list is one of Python's fundamental data structures, allowing for the storage and manipulation of a collection of items. Python has a plethora of built-in methods, including the index() method, to make efficient list operations easier. We will examine the Python list index() method in depth and comprehend its functionality and application in this article.

If list contains duplicate elements, it returns index of first occurred element.

This method takes two more optional parameters start and end which are used to search index within a limit.

Signature

```
index(x[, start[, end]])
```

Parameters

No parameter

Return

It returns None.

Let's see some examples of index() method to understand it's functionality.

Python List index() Method Example 1

Let's first see a simple example to get index of an element using index() method.

```
# Python list index() Method
# Creating a list
apple = ['a','p','p','l','e']
# Method calling
index = apple.index('p')
# Displaying result
print("Index of p :",index)
```

Output:

```
Index of p : 1
```

Python List index() Method Example 2

The method returns index of first element occurred, if the element is duplicate. See the following example.

```
# Python list index() Method
# Creating a list
apple = ['a','p','p','l','e']
# Method calling
index = apple.index('l') # 3
index2 = apple.index('p') # always 1
# Displaying result
print("Index of l :",index)
print("Index of p :",index2)
```

Output:

```
Index of l : 3
Index of p : 1
```

Python List index() Method Example 3

The index() method throws an error ValueError if the element is not present in the list. See the example below.

```
# Python list index() Method
# Creating a list
apple = ['a','p','p','l','e']
# Method calling
index = apple.index('z')
# Displaying result
print("Index of l :",index)
```

Output:

```
index = apple.index('z')
ValueError: 'z' is not in list
```

Example 2:

```
fruits = ['apple', 'banana', 'mango', 'apple', 'orange']
print(fruits.index('apple', 2))
```

Output:

```
Output: 3
```

```
print(fruits.index('apple', 2, 4))
```

Output:

```
Output: 3
```

Python List index() Method Example 4

Apart from element, the method takes two more optional parameters start and end. These are used to find index within a limit.

```
# Python list index() Method
# Creating a list
apple = ['a','p','p','l','e']
# Method calling
index = apple.index('p',2)  # start index
index2 = apple.index('a',0,3)   # end index
# Displaying result
print("Index of p :",index)
print("Index of a :",index2)
```

Output:

```
Index of p : 2
Index of a : 0
```

Beginning at the specified start index, the index() method conducts a linear search for the element. Using other data structures like sets or dictionaries to speed up search operations is recommended if you have a large list or need to perform frequent index lookups.

In conclusion, Python lists' index() method makes it simple to determine the index of an element's first occurrence. When working with lists of data and needing to quickly locate specific elements, it is a useful tool. You can quickly find where a list item is by using the index() method, allowing you to carry out subsequent operations or make educated choices based on the index.

It is important to note that the index() method only returns the index of the element's first occurrence. You would need to use a different strategy, such as a loop or list comprehension, if there are multiple instances of the same element in the list and you want to find more of them.

Additionally, keep in mind that if the specified element is not found in the list, the index() method will throw a ValueError. Use a try-except block to catch the exception and provide alternative instructions or error messages to handle this situation gracefully.

When using the index() method, you can also specify parameters for the start and end. Instead of scanning the entire list, you can define a specific range within the list to search for the element. When working with large lists, you can improve your code's performance by limiting the search range.

When setting the parameters for the start and end, extreme caution is required. The index() method will throw a ValueError, indicating an invalid range, if the end parameter is smaller than the start parameter. As a result, it is critical to ensure that the range is correctly defined to prevent unforeseen errors.

Consider using alternative data structures like sets or dictionaries if you need to perform frequent index lookups or the order of the elements is not important. While dictionaries offer key-value pairs that enable you to associate particular values with unique keys, sets offer lookup operations that operate in constant time. These data structures may offer more effective solutions than the index() method, depending on your use case.

Python slice() Function

In Python, we have a number of inbuilt functions. One such function is the Python slice() function. The Python slice function is used to get a slice or a portion of elements from the collection of elements such as a list, tuple, or string.

The slice function offers a simple and effective method for extracting a portion of data and modifying data from a sequence of elements.

You can use the slice function to encapsulate slice logic, such as start, stop, and step parameters.

Python provides two overloaded slice functions. The first function takes a single argument, while the second function takes three arguments and returns a slice object. This slice object can be used to get a subsection of the collection. For example, if we want to get the first two elements from the list of elements, here slice can be used.

Introduction to Slicing:

Before we deep dive into the Python slice function. We must understand the concept of slicing.

Slicing is a process of extracting data from a collection of data by specifying the starting and ending indices and steps to be taken.

The general syntax of slicing is as follows:

```
Collection[start:end:step]
```

The collection can be any list, string, or tuple (an iterable). The **start** parameter denotes the starting index, the **end** parameter denotes the ending index, and the **step** parameter denotes the number of elements to be skipped during the slice operation.

Introduction to Python Slice Function:

Signature

The signature of the slice function is given below.:

```
slice (stop)
slice (start, stop[, step])
```

Parameters

- **start:** The start parameter in the slice function is used to set the starting position or index of the slicing. The default value of the start is 0.
- **stop:** The stop parameter in the slice function is used to set the end position or index of the slicing.
- **step:** The step parameter in the slice function is used to set the number of steps to jump. The default value of the step is 1.

The **start, stop**, and **step** parameters are similar to those used in the slicing syntax, and it returns a slice object.

Let's see some examples of slice() function to understand its functionality.

Python slice() Function Example 1 - Creating a slice object

```
# Python program to demonstrate
# how to create a slice() object

# Calling function
slice1 = slice(5) # returns a slice object
slice2 = slice(0,5,3) # returns a slice object

# Displaying the result
print(slice1)
print(slice1)
```

Output:

```
slice(None, 5, None)
slice(0, 5, 3)
```

In this example, we create two slice objects using slice(5) and slice(0, 5, 3). The slice(5) represents a slice that starts from index 0 and ends at index 5 (exclusive). And the slice(0, 5, 3) represents a slice that starts from index 0, ends at index 5 (exclusive), and selects every third element from the sequence.

Python slice() Function Example 2 - Using a slice object

Let's use this slice object to extract data from a list of elements:

```
# Python program to demonstrate
# how to use a slice() object

# Calling function
slice1 = slice(5) # returns a slice object
slice2 = slice(0, 5, 3) # returns a slice object

# Defining a list
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Extracting data and displaying the result
print("Slice 1 of My list:", my_list[slice1])
print("Slice 2 of My list:", my_list[slice2])
```

Output:

```
Slice 1 of My list: [1, 2, 3, 4, 5]
Slice 2 of My list: [1, 4]
```

In the above example, the slice1 object extracts the portion of data from the my_list starting from index 0 and ending at index 4. And the slice2 object extracts the portion of data from the my_list starting from index 1 and selects every third element from the index.

Python slice() Function Example 3 - Slicing string

```
# Python program to demonstrate
# how to slice strings using a slice object

# Defining a string
```

```
my_str = "Javatpoint"

# Creating slice objects
slice1 = slice(0, 10, 3) # returns slice object
slice2 = slice(-1, 0, -3) # returns slice object

# We can use these slice objects to slice the string
result1 = my_str[slice1]
result2 = my_str[slice2] # returns elements in reverse order

# Displaying result
print("Result 1:", result1)
print("Result 2:", result2)
```

Output:

```
Result 1: Jaot
Result 2: toa
```

Explanation:

In the above example, slice1 represents a slice object that starts extracting data from index 0 and extracts every third element from the string ending at index 10 (exclusively).

And slice2 represents a slice object that starts extracting data from the last index (index = -1) and extracts every third element in reverse order ending at index 0 (exclusively).

Python slice() Function Example 4 - Slicing Tuple

```
# Python program to demonstrate
# how to slice a tuple using a slice object

# Creating a tuple
tup = (45, 68, 955, 1214, 41, 558, 636, 66)

# Creating slice objects
slice1 = slice(0, 10, 3)  # returns slice object
slice2 = slice(-1, 0, -3)  # returns slice object

```

```python
# We can use these slice objects to slice a tuple
result1 = tup[slice1]
result2 = tup[slice2]  # returns elements in reverse order

# Displaying result
print("Result 1:", result1)
print("Result 2:", result2)
```

Output:

```
Result 1: (45, 1214, 636)
Result 2: (66, 41, 68)
```

Explanation:

As we discussed in the previous example, the slice1 object extracts every third element from 'tup' starting from index 0 and ending at index 10 (exclusively).

And the slice2 object extracts every third element starting from index -1 and ending at index 0 (exclusively).

Python slice() Function Example 5 - General Slicing vs. Slice Function

```python
# Python program to demonstrate
# general slicing and slicing using a slice object

# Creating a tuple
tup = (45, 68, 955, 1214, 41, 558, 636, 66)

# Creating a slice object
slice1 = slice(0, 10, 3)  # returns slice object

# Slicing the tuple using a slice object
result1 = tup[slice1]
print("Slicing using slice object:", result1)

# General Slicing
result2 = tup[0:10:3]  # fetch the same elements

```

17. # Displaying result
18. **print**("Slicing using general slicing:", result2)

Output:

```
Slicing using slice object: (45, 1214, 636)
Slicing using general slicing: (45, 1214, 636)
```

Explanation:

In the above example, both slicing methods extract every third element from the 'tup' starting from index 0 and ending at index 10 (exclusively).

Advantages of Slice() function in Python:

In Python, the slice function offers many advantages while working with sequences or collections of data such as lists, tuples, or strings. Some of the main advantages of the slice function are as follows:

1. **Reusability -** Once created, the slice object can be applied to slice many sequences making it advantageous in terms of reusability.
2. **Readability -** The slice object is used to encapsulate the slicing logic, which enhances the readability of the code.
3. **Flexibility -** The slice function takes three parameters start, stop, and steps. You can leave any of these parameters or use negative indices for the same.

Python Arrays

Introduction:

In this article, we are discussing Arrays in Python. The Array is used in every programming language, like C, C++, Java, Python, R, JavaScript, etc. By using an array, we can store more than one data. The Array is a process of memory allocation. It is performed as a dynamic memory allocation. We can declare an array like x[100], storing 100 data in x. It is a container that can hold a fixed number of items, and these items should be the same type. An array is popular in most programming languages like C/C++, JavaScript, etc.

The Array is an idea of storing multiple items of the same type together, making it easier to calculate the position of each element by simply adding an offset to the base value. A combination of the arrays could save a lot of time by reducing the overall size of the code. It is used to store multiple values in a single variable. If you have a list of items that are stored in their corresponding variables like this:

1. car1 = "Lamborghini"
2. car2 = "Bugatti"

3. car3 = "Koenigsegg"

If you want to loop through cars and find a specific one, you can use the Array. You can use an array to store more than one item in a specific variable.

The Array can be handled in Python by a module named **Array**. It is useful when we must manipulate only specific data values. The following are the terms to understand the concept of an array:

Element - Each item stored in an array is called an element.

Index - The location of an element in an array has a numerical index, which is used to identify the element's position. The index value is very much important in an Array.

Array Representation:

An array can be declared in various ways and in different languages. The important points that should be considered are as follows:

1. The index starts with 0.
2. We can easily find any elements within this Array using the Index value.
3. The length of the Array defines the capacity to store the elements. It is written like x[100], which means the length of array x is specified by 100.

Array operations

Some of the basic operations supported by an array are as follows:

- **Traverse -** It prints all the elements one by one.
- **Insertion -** It adds an element at the given index.
- **Deletion -** It deletes an element at the given index.
- **Search -** It searches an element using the given index or by the value.
- **Update -** It updates an element at the given index.

The Array can be created in Python by importing the array module to the python program.

1. from array **import** *
2. arrayName = array(typecode, [initializers])

Accessing array elements

We can access the array elements using the respective indices of those elements.

Program code:

Here we give an example of how we access the elements of an array using its index value in Python. The code is given below -

```
import array as arr
a = arr.array('i', [2, 4, 5, 6])
print("First element is:", a[0])
print("Second element is:", a[1])
print("Third element is:", a[2])
print("Forth element is:", a[3])
print("last element is:", a[-1])
print("Second last element is:", a[-2])
print("Third last element is:", a[-3])
print("Forth last element is:", a[-4])
print(a[0], a[1], a[2], a[3], a[-1],a[-2],a[-3],a[-4])
```

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
First element is: 2
Second element is: 4
Third element is: 5
Forth element is: 6
last element is: 6
Second last element is: 5
Third last element is: 4
Forth last element is: 2
2 4 5 6 6 5 4 2
```

Explanation:

In the above example, we have imported an array, defined a variable named "a" that holds the elements of an array, and printed the elements by accessing elements through the indices of an array. Here we can easily find out the array element by using the array index like a[0], a[1], a[-1], and so on.

How to change or add elements?

Arrays are mutable, and their elements can be changed similarly to lists.

Program code:

Here in this example, we can change or add or replace any element from the Array in Python. The code is given below -

```
import array as arr
numbers = arr.array('i', [1, 2, 3, 5, 7, 10])

# changing first element 1 by the value 0.
numbers[0] = 0
print(numbers)          # Output: array('i', [0, 2, 3, 5, 7, 10])

# changing last element 10 by the value 8.
numbers[5] = 8
print(numbers)          # Output: array('i', [0, 2, 3, 5, 7, 10])

# replace the value of 3rd to 5th element by 4, 6 and 8
numbers[2:5] = arr.array('i', [4, 6, 8])
print(numbers)          # Output: array('i', [0, 2, 4, 6, 8, 10])
```

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
array('i', [0, 2, 3, 5, 7, 10])
array('i', [0, 2, 3, 5, 7, 8])
array('i', [0, 2, 4, 6, 8, 8])
```

Explanation:

In the above example, we have imported an array and defined a variable named "numbers," which holds the value of an array. If we want to change or add the elements in an array, we can do it by defining the index of an array where we want to change or add the elements. Here we just mentioned the index number of elements you want to change and declared the new value by which you want to replace the old elements.

Why use Arrays in Python?

A combination of arrays saves a lot of time. The Array can reduce the overall size of the code. Using an array, we can solve a problem quickly in any language. The Array is used for dynamic memory allocation.

How to Delete Elements from an Array?

The elements can be deleted from an array using Python's **del** statement. If we want to delete any value from the Array, we can use the indices of a particular element.

```
import array as arr
number = arr.array('i', [1, 2, 3, 3, 4])
del number[2]                          # removing third element
print(number)                          # Output: array('i', [1, 2, 3, 4])
```

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
array('i', [10, 20, 40, 60])
```

Explanation: In the above example, we have imported an array and defined a variable named as "number" which stores the values of an array. Here, by using del statement, we are removing the third element [3] of the given array.

The length of an array is defined as the number of elements present in an array. It returns an integer value that is equal to the total number of the elements present in that array.

Syntax

By using the syntax below, we can easily find the length of the given Array. The syntax is -

```
len(array_name)
```

Array Concatenation

We can easily concatenate any two arrays using the + symbol.

Example 1:

```
a=arr.array('d',[1.1 , 2.1 ,3.1,2.6,7.8])
b=arr.array('d',[3.7,8.6])
c=arr.array('d')
c=a+b
print("Array c = ",c)
```

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
Array c= array('d', [1.1, 2.1, 3.1, 2.6, 7.8, 3.7, 8.6])
```

Explanation

In the above example, we have defined variables named as "a, b, c" that hold the values of an array.

Example 2:

1. **import** array as arr
2. x = arr.array('i', [4, 7, 19, 22]) # Initialize the array elements
3. print("First element:", x[0])
4. print("Second element:", x[1])
5. print("Second last element:", x[-1])

Output:

```
First element: 4
Second element: 7
Second last element: 22
```

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

Explanation:

In the above example, first, we imported an array and defined a variable named "x," which holds the value of an array. Then, we printed the elements of an array using the index value of this array element.

Pandas Introduction

The name of Pandas is gotten from the word Board Information, and that implies an Econometrics from Multi-faceted information. It was created in 2008 by Wes McKinney and is used for data analysis in Python.

Processing, such as restructuring, cleaning, merging, etc., is necessary for data analysis. Numpy, Scipy, Cython, and Panda are just a few of the fast data processing tools available. Yet, we incline toward Pandas since working with Pandas is quick, basic and more expressive than different apparatuses.

Since Pandas is built on top of the Numpy bundle, it is expected that Numpy will work with Pandas.

Before Pandas, Python was able for information planning, however it just offered restricted help for information investigation. As a result, Pandas entered the picture and enhanced data analysis capabilities. Regardless of the source of the data, it can carry out the five crucial steps that are necessary for processing and analyzing it: load, manipulate, prepare, model, and analyze.

Key Features of Pandas

- It has a DataFrame object that is quick and effective, with both standard and custom indexing.
- Utilized for reshaping and turning of the informational indexes.
- For aggregations and transformations, group by data.
- It is used to align the data and integrate the data that is missing.
- Provide Time Series functionality.
- Process a variety of data sets in various formats, such as matrix data, heterogeneous tabular data, and time series.
- Manage the data sets' multiple operations, including subsetting, slicing, filtering, groupBy, reordering, and reshaping.
- It incorporates with different libraries like SciPy, and scikit-learn.
- Performs quickly, and the Cython can be used to accelerate it even further.

Benefits of Pandas

The following are the advantages of pandas overusing other languages:

Representation of Data: Through its DataFrame and Series, it presents the data in a manner that is appropriate for data analysis.

Clear code: Pandas' clear API lets you concentrate on the most important part of the code. In this way, it gives clear and brief code to the client.

DataFrame and Series are the two data structures that Pandas provides for processing data. These data structures are discussed below:

1) Series

A one-dimensional array capable of storing a variety of data types is how it is defined. The term "index" refers to the row labels of a series. We can without much of a stretch believer the rundown, tuple, and word reference into series utilizing "series' technique. Multiple columns cannot be included in a Series. Only one parameter exists:

Data: It can be any list, dictionary, or scalar value.

Creating Series from Array:

Before creating a Series, Firstly, we have to import the numpy module and then use array() function in the program.

```
import pandas as pd
import numpy as np
```

```
info = np.array(['P','a','n','d','a','s'])
a = pd.Series(info)
print(a)
```

Output

```
0   P
1   a
2   n
3   d
4   a
5   s
dtype: object
```

Explanation: In this code, firstly, we have imported the **pandas** and **numpy** library with the **pd** and **np** alias. Then, we have taken a variable named "info" that consist of an array of some values. We have called the **info** variable through a **Series** method and defined it in an "**a**" variable. The Series has printed by calling the **print(a)** method.

Python Pandas DataFrame

It is a generally utilized information design of pandas and works with a two-layered exhibit with named tomahawks (lines and segments). As a standard method for storing data, DataFrame has two distinct indexes-row index and column index. It has the following characteristics:

The sections can be heterogeneous sorts like int, bool, etc.

It can be thought of as a series structure dictionary with indexed rows and columns. It is referred to as "columns" for rows and "index" for columns.

Create a DataFrame using List:

We can easily create a DataFrame in Pandas using list.

```
import pandas as pd
# a list of strings
x = ['Python', 'Pandas']

# Calling DataFrame constructor on list
df = pd.DataFrame(x)
print(df)
```

Output

```
     0
0   Python
1   Pandas
```

Explanation: In this code, we have characterized a variable named "x" that comprise of string values. On a list, the values are being printed by calling the DataFrame constructor.

Python Pandas Series

The Pandas Series can be defined as a one-dimensional array that is capable of storing various data types. We can easily convert the list, tuple, and dictionary into series using "**series**' method. The row labels of series are called the index. A Series cannot contain multiple columns. It has the following parameter:

- **data:** It can be any list, dictionary, or scalar value.
- **index:** The value of the index should be unique and hashable. It must be of the same length as data. If we do not pass any index, default **np.arrange(n)** will be used.
- **dtype:** It refers to the data type of series.
- **copy:** It is used for copying the data.

Creating a Series:

We can create a Series in two ways:

1. Create an empty Series
2. Create a Series using inputs.

Create an Empty Series:

We can easily create an empty series in Pandas which means it will not have any value.

The syntax that is used for creating an Empty Series:

```
<series object> = pandas.Series()
```

The below example creates an Empty Series type object that has no values and having default datatype, i.e., **float64**.

Example

```
import pandas as pd
x = pd.Series()
print (x)
```

Output

```
Series([], dtype: float64)
```

Creating a Series using inputs:

We can create Series by using various inputs:

- Array
- Dict
- Scalar value

Creating Series from Array:

Before creating a Series, firstly, we have to import the **numpy** module and then use array() function in the program. If the data is ndarray, then the passed index must be of the same length.

If we do not pass an index, then by default index of **range(n)** is being passed where n defines the length of an array, i.e., [0,1,2,....**range(len(array))-1**].

Example

```
import pandas as pd
import numpy as np
info = np.array(['P','a','n','d','a','s'])
a = pd.Series(info)
print(a)
```

Output

```
0    P
1    a
2    n
3    d
4    a
5    s
dtype: object
```

Create a Series from dict

We can also create a Series from dict. **If the dictionary object is being passed as an input and the index is not specified, then the dictionary keys are taken in a sorted order to construct the index**.

If index is passed, then values correspond to a particular label in the index will be extracted from the **dictionary**.

```
#import the pandas library
import pandas as pd
import numpy as np
info = {'x' : 0., 'y' : 1., 'z' : 2.}
a = pd.Series(info)
print (a)
```

Output

```
x     0.0
y     1.0
z     2.0
dtype: float64
```

Create a Series using Scalar:

If we take the scalar values, then the index must be provided. The scalar value will be repeated for matching the length of the index.

```
#import pandas library
import pandas as pd
import numpy as np
x = pd.Series(4, index=[0, 1, 2, 3])
print (x)
```

Output

```
0      4
1      4
2      4
3      4
dtype: int64
```

Accessing data from series with Position:

Once you create the Series type object, you can access its indexes, data, and even individual elements.

The data in the Series can be accessed similar to that in the ndarray.

```
import pandas as pd
x = pd.Series([1,2,3],index = ['a','b','c'])
#retrieve the first element
```

4. print (x[0])

   5. **Output**

```
1
```

Python Pandas DataFrame

Pandas DataFrame is a widely used data structure which works with a two-dimensional array with labeled axes (rows and columns). DataFrame is defined as a standard way to store data that has two different indexes, i.e., **row index** and **column index**. It consists of the following properties:

- The columns can be heterogeneous types like int, bool, and so on.
- It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.

Parameter & Description:

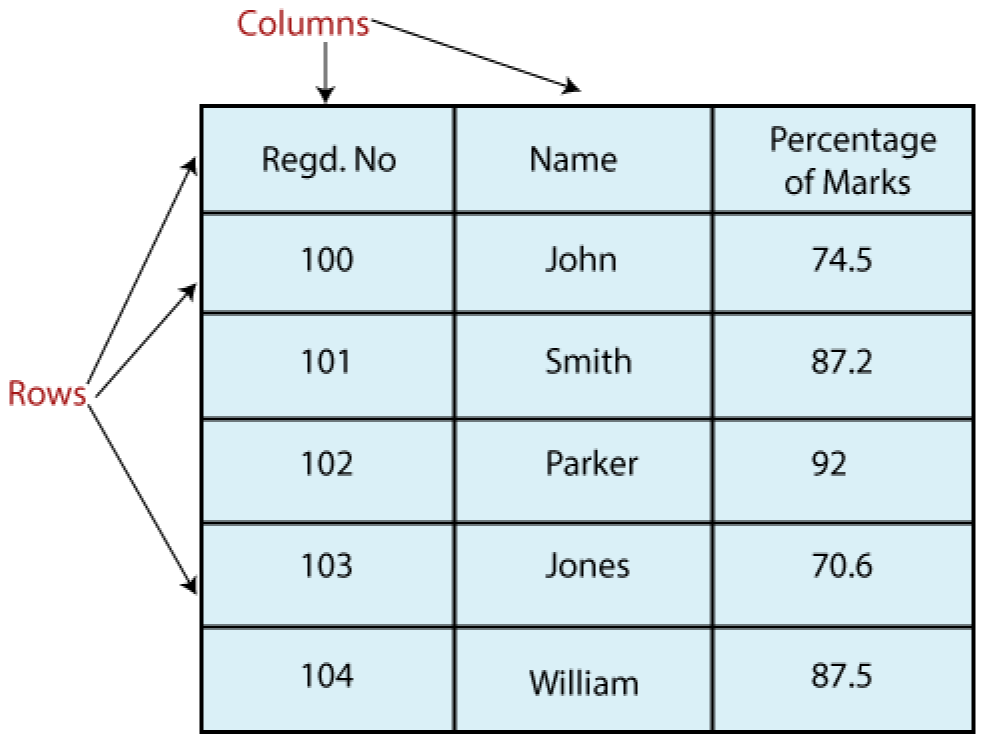
data: It consists of different forms like ndarray, series, map, constants, lists, array.

index: The Default np.arrange(n) index is used for the row labels if no index is passed.

columns: The default syntax is np.arrange(n) for the column labels. It shows only true if no index is passed.

dtype: It refers to the data type of each column.

copy(): It is used for copying the data.


Columns
Rows

Regd. No	Name	Percentage of Marks
100	John	74.5
101	Smith	87.2
102	Parker	92
103	Jones	70.6
104	William	87.5

Create a DataFrame

We can create a DataFrame using following ways:

- **dict**
- **Lists**
- **Numpy ndarrrays**
- **Series**

Create an empty DataFrame

The below code shows how to create an empty DataFrame in Pandas:

```
# importing the pandas library
import pandas as pd
df = pd.DataFrame()
print (df)
```

Output

```
Empty DataFrame
Columns: []
Index: []
```

Explanation: In the above code, first of all, we have imported the pandas library with the alias **pd** and then defined a variable named as **df** that consists an empty DataFrame. Finally, we have printed it by passing the **df** into the **print**.

Create a DataFrame using List:

We can easily create a DataFrame in Pandas using list.

```
# importing the pandas library
import pandas as pd
# a list of strings
x = ['Python', 'Pandas']

# Calling DataFrame constructor on list
df = pd.DataFrame(x)
print(df)
```

Output

```
     0
0   Python
1   Pandas
```

Explanation: In the above code, we have defined a variable named "x" that consist of string values. The DataFrame constructor is being called for a list to print the values.

Create a DataFrame from Dict of ndarrays/ Lists

```
# importing the pandas library
import pandas as pd
info = {'ID' :[101, 102, 103],'Department' :['B.Sc','B.Tech','M.Tech',]}
df = pd.DataFrame(info)
print (df)
```

Output

```
     ID     Department
0     101       B.Sc
1     102       B.Tech
2     103       M.Tech
```

Explanation: In the above code, we have defined a dictionary named "info" that consist **list** of **ID** and **Department**. For printing the values, we have to call the info dictionary through a variable called **df** and pass it as an argument in **print()**.

Create a DataFrame from Dict of Series:

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)
print (d1)
```

Output

```
    one    two
a   1.0     1
b   2.0     2
c   3.0     3
d   4.0     4
e   5.0     5
f   6.0     6
g   NaN     7
h   NaN     8
```

Explanation: In the above code, a dictionary named "**info**" consists of two **Series** with its respective index. For printing the values, we have to call the **info** dictionary through a variable called **d1** and pass it as an argument in **print()**.

Column Selection

We can select any column from the DataFrame. Here is the code that demonstrates how to select a column from the DataFrame.

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)
```

```
print (d1 ['one'])
```

Output

```
a      1.0
b      2.0
c      3.0
d      4.0
e      5.0
f      6.0
g      NaN
h      NaN
Name: one, dtype: float64
```

Explanation: In the above code, a dictionary named "**info**" consists of two **Series** with its respective **index**. Later, we have called the **info** dictionary through a variable **d1** and selected the "**one**" Series from the DataFrame by passing it into the **print()**.

Column Addition

We can also add any new column to an existing DataFrame. The below code demonstrates how to add any new column to an existing DataFrame:

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}

df = pd.DataFrame(info)

# Add a new column to an existing DataFrame object

print ("Add new column by passing series")
df['three']=pd.Series([20,40,60],index=['a','b','c'])
print (df)

print ("Add new column using existing DataFrame columns")
df['four']=df['one']+df['three']

print (df)
```

19. **Output**

```
Add new column by passing series
      one    two     three
a    1.0     1       20.0
b    2.0     2       40.0
c    3.0     3       60.0
d    4.0     4       NaN
e    5.0     5       NaN
f    NaN     6       NaN

Add new column using existing DataFrame columns
       one     two      three     four
a     1.0      1        20.0     21.0
b     2.0      2        40.0     42.0
c     3.0      3        60.0     63.0
d     4.0      4        NaN      NaN
e     5.0      5        NaN      NaN
f     NaN      6        NaN      NaN
```

37. **Explanation:** In the above code, a dictionary named as **f** consists two **Series** with its respective **index**. Later, we have called the **info** dictionary through a variable **df**.
38. To add a new column to an existing DataFrame object, we have passed a new series that contain some values concerning its index and printed its result using **print()**.
39. We can add the new columns using the existing DataFrame. The "**four**" column has been added that stores the result of the addition of the two columns, i.e., **one** and **three**.

Column Deletion:

We can also delete any column from the existing DataFrame. This code helps to demonstrate how the column can be deleted from an existing DataFrame:

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2], index= ['a', 'b']),
   'two' : pd.Series([1, 2, 3], index=['a', 'b', 'c'])}

df = pd.DataFrame(info)
print ("The DataFrame:")
print (df)

# using del function
print ("Delete the first column:")
del df['one']
```

```
print (df)
# using pop function
print ("Delete the another column:")
df.pop('two')
print (df)
```

Output

```
The DataFrame:
     one    two
a     1.0     1
b     2.0     2
c     NaN     3

Delete the first column:
    two
a     1
b     2
c     3

Delete the another column:
Empty DataFrame
Columns: []
Index: [a, b, c]
```

Explanation:

In the above code, the **df** variable is responsible for calling the **info** dictionary and print the entire values of the dictionary. We can use the **delete** or **pop** function to delete the columns from the DataFrame.

In the first case, we have used the **delete** function to delete the "**one**" column from the DataFrame whereas in the second case, we have used the **pop** function to remove the "**two**" column from the DataFrame.

Row Selection, Addition, and Deletion

Row Selection:

We can easily select, add, or delete any row at anytime. First of all, we will understand the row selection. Let's see how we can select a row using different ways that are as follows:

Selection by Label:

We can select any row by passing the row label to a **loc** function.

```
# importing the pandas library
import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}

df = pd.DataFrame(info)
print (df.loc['b'])
```

Output

```
one    2.0
two    2.0
Name: b, dtype: float64
```

Explanation: In the above code, a dictionary named as **info** that consists two **Series** with its respective **index**.

For selecting a row, we have passed the row label to a **loc** function.

Selection by integer location:

The rows can also be selected by passing the integer location to an **iloc** function.

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
df = pd.DataFrame(info)
print (df.iloc[3])
```

Output

```
one    4.0
two    4.0
Name: d, dtype: float64
```

Explanation: Explanation: In the above code, we have defined a dictionary named as **info** that consists two **Series** with its respective **index**.

For selecting a row, we have passed the integer location to an **iloc** function.

Slice Rows

It is another method to select multiple rows using **':'** operator.

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
   'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
df = pd.DataFrame(info)
print (df[2:5])
```

Output

```
     one    two
c    3.0     3
d    4.0     4
e    5.0     5
```

Explanation: In the above code, we have defined a range from 2:5 for the selection of row and then printed its values on the console.

Addition of rows:

We can easily add new rows to the DataFrame using **append** function. It adds the new rows at the end.

```
# importing the pandas library
import pandas as pd
d = pd.DataFrame([[7, 8], [9, 10]], columns = ['x','y'])
d2 = pd.DataFrame([[11, 12], [13, 14]], columns = ['x','y'])
d = d.append(d2)
print (d)
```

Output

```
     x     y
0    7     8
1    9     10
0    11    12
1    13    14
```

Explanation: In the above code, we have defined two separate lists that contains some rows and columns. These columns have been added using the **append** function and then result is displayed on the console.

Deletion of rows:

We can delete or drop any rows from a DataFrame using the **index** label. If in case, the label is duplicate then multiple rows will be deleted.

```
# importing the pandas library
import pandas as pd

a_info = pd.DataFrame([[4, 5], [6, 7]], columns = ['x','y'])
b_info = pd.DataFrame([[8, 9], [10, 11]], columns = ['x','y'])

a_info = a_info.append(b_info)

# Drop rows with label 0
a_info = a_info.drop(0)
```

Output

```
x    y
1    6    7
1    10   11
```

Explanation: In the above code, we have defined two separate lists that contains some rows and columns.

Here, we have defined the index label of a row that needs to be deleted from the list

DataFrame Functions

There are lots of functions used in DataFrame which are as follows:

Functions	**Description**
Pandas DataFrame.append()	Add the rows of other dataframe to the end of the given dataframe.
Pandas DataFrame.apply()	Allows the user to pass a function and apply it to every single value of the Pandas series.
Pandas DataFrame.assign()	Add new column into a dataframe.
Pandas DataFrame.astype()	Cast the Pandas object to a specified dtype.astype() function.
Pandas DataFrame.concat()	Perform concatenation operation along an axis in the DataFrame.
Pandas DataFrame.count()	Count the number of non-NA cells for each column or row.

Pandas DataFrame.describe()	Calculate some statistical data like percentile, mean and std of the numerical values of the Series or DataFrame.
Pandas DataFrame.drop_duplicates()	Remove duplicate values from the DataFrame.
Pandas DataFrame.groupby()	Split the data into various groups.
Pandas DataFrame.head()	Returns the first n rows for the object based on position.
Pandas DataFrame.hist()	Divide the values within a numerical variable into "bins".
Pandas DataFrame.iterrows()	Iterate over the rows as (index, series) pairs.
Pandas DataFrame.mean()	Return the mean of the values for the requested axis.
Pandas DataFrame.melt()	Unpivots the DataFrame from a wide format to a long format.
Pandas DataFrame.merge()	Merge the two datasets together into one.
Pandas DataFrame.pivot_table()	Aggregate data with calculations such as Sum, Count, Average, Max, and Min.
Pandas DataFrame.query()	Filter the dataframe.
Pandas DataFrame.sample()	Select the rows and columns from the dataframe randomly.
Pandas DataFrame.shift()	Shift column or subtract the column value with the previous row value from the dataframe.
Pandas DataFrame.sort()	Sort the dataframe.
Pandas DataFrame.sum()	Return the sum of the values for the requested axis by the user.
Pandas DataFrame.to_excel()	Export the dataframe to the excel file.
Pandas DataFrame.transpose()	Transpose the index and columns of the dataframe.
Pandas DataFrame.where()	Check the dataframe for one or more conditions.

Index objects#

Index

Pandas Index

Pandas Index is defined as a vital tool that selects particular rows and columns of data from a DataFrame. Its task is to organize the data and to provide fast accessing of data. It can also be called a **Subset Selection**.

The values are in **bold** font in the index, and the individual value of the index is called a **label**.

If we want to compare the data accessing time with and without indexing, we can use **%%timeit** for comparing the time required for various access-operations.

We can also define an index like an address through which any data can be accessed across the Series or DataFrame. A DataFrame is a combination of three different components, the **index**, **columns,** and the **data**.

Axis and axes

An axis is defined as a common terminology that refers to rows and columns, whereas axes are collection of these rows and columns.

Creating index

First, we have to take a csv file that consist some data used for indexing.

1. # importing pandas **package**
2. **import** pandas as pd
3. data = pd.read_csv("aa.csv")
4. data

Output:

```
   Name            Hire Date    Salary      Leaves Remaining
0  John Idle         03/15/14    50000.0       10
1  Smith Gilliam      06/01/15    65000.0       8
2  Parker Chapman     05/12/14    45000.0       10
3  Jones Palin        11/01/13    70000.0       3
4  Terry Gilliam      08/12/14    48000.0       7
5   Michael Palin     05/23/13    66000.0       8
```

Example1

1. # importing pandas **package**
2. **import** pandas as pd

```
# making data frame from csv file
info = pd.read_csv("aa.csv", index_col ="Name")
# retrieving multiple columns by indexing operator
a = info[["Hire Date", "Salary"]]
print(a)
```

Output:

```
   Name           Hire Date    Salary
0  John Idle        03/15/14    50000.0
1  Smith Gilliam     06/01/15    65000.0
2  Parker Chapman    05/12/14    45000.0
3  Jones Palin       11/01/13    70000.0
4  Terry Gilliam     08/12/14    48000.0
5  Michael Palin     05/23/13    66000.0
```

Example2:

```
# importing pandas package
importpandas as pd

# making data frame from csv file
info =pd.read_csv("aa.csv", index_col ="Name")

# retrieving columns by indexing operator
a =info["Salary"]
print(a)
```

Output:

```
    Name           Salary
0  John Idle        50000.0
1  Smith Gilliam     65000.0
2  Parker Chapman    45000.0
3  Jones Palin       70000.0
4  Terry Gilliam     48000.0
5   Michael Palin    66000.0
```

Set index

The '**set_index**' is used to set the DataFrame index using existing columns. An index can replace the existing index and can also expand the existing index.

It set a list, Series or DataFrame as the index of the DataFrame.

```
info = pd.DataFrame({'Name': ['Parker', 'Terry', 'Smith', 'William'],
'Year': [2011, 2009, 2014, 2010],
'Leaves': [10, 15, 9, 4]})
info
info.set_index('Name')
info.set_index(['year', 'Name'])
info.set_index([pd.Index([1, 2, 3, 4]), 'year'])
a = pd.Series([1, 2, 3, 4])
info.set_index([a, a**2])
```

Output:

```
          Name       Year    Leaves
1   1     Parker      2011     10
2   4     Terry       2009     15
3   9     Smith       2014     9
4   16    William     2010     4
```

Multiple Index

We can also have multiple indexes in the data.

Example1:

```
import pandas as pd
import numpy as np
pd.MultiIndex(levels=[[np.nan, None, pd.NaT, 128, 2]],
codes=[[0, -1, 1, 2, 3, 4]])
```

Output:

```
MultiIndex(levels=[[nan, None, NaT, 128, 2]],
codes=[[0, -1, 1, 2, 3, 4]])
```

Reset index

We can also reset the index using the '**reset_index**' command. Let's look at the '**cm**' DataFrame again.

Example:

```
info = pd.DataFrame([('William', 'C'),
('Smith', 'Java'),
```

3. ('Parker', 'Python'),
4. ('Phill', np.nan)],
5. index=[1, 2, 3, 4],
6. columns=('name', 'Language'))
7. info
8. info.reset_index()

Output:

```
   index   name      Language
0             1     William     C
1             2     Smith      Java
2             3     Parker     Python
3             4     Phill      NaN
```

Data Visualization using Matplotlib

Data Visualization is the process of presenting data in the form of graphs or charts. It helps to understand large and complex amounts of data very easily. It allows the decision-makers to make decisions very efficiently and also allows them in identifying new trends and patterns very easily. It is also used in high-level data analysis for Machine Learning and Exploratory Data Analysis (EDA). Data visualization can be done with various tools like Tableau, Power BI, Python.

In this article, we will discuss how to visualize data with the help of the Matplotlib library of Python.

Matplotlib

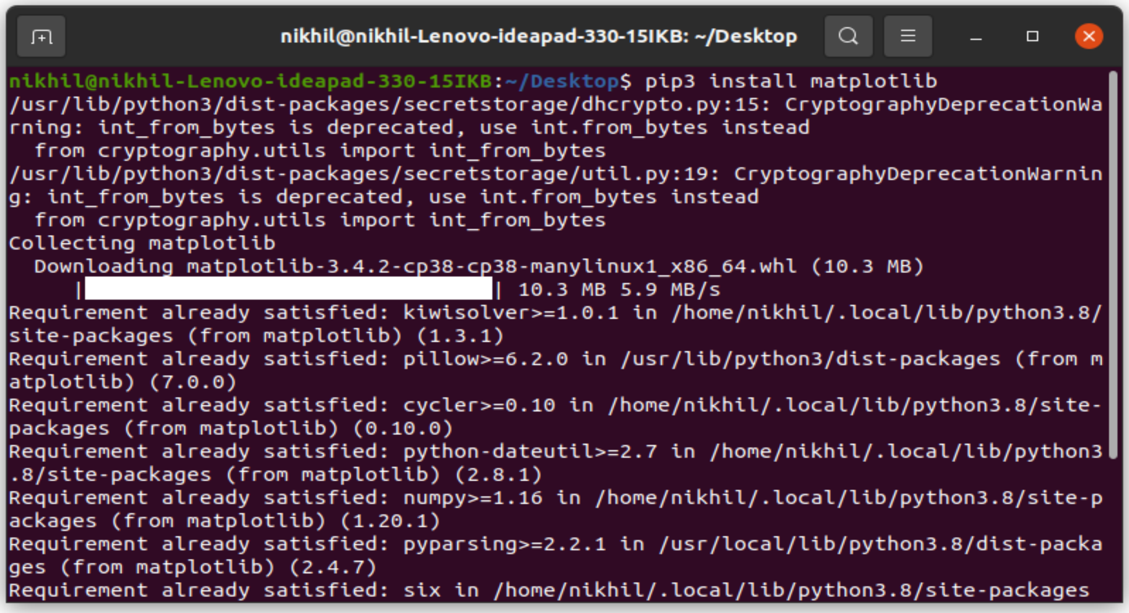
Matplotlib is a low-level library of Python which is used for data visualization. It is easy to use and emulates MATLAB like graphs and visualization. This library is built on the top of NumPy arrays and consist of several plots like line chart, bar chart, histogram, etc. It provides a lot of flexibility but at the cost of writing more code.

Installation

We will use the pip command to install this module. If you do not have pip installed then refer to the article, **Download and install pip Latest Version.**

To install Matplotlib type the below command in the terminal.

```
pip install matplotlib
```



```
nikhil@nikhil-Lenovo-ideapad-330-15IKB:~/Desktop$ pip3 install matplotlib
/usr/lib/python3/dist-packages/secretstorage/dhcrypto.py:15: CryptographyDeprecationWa
rning: int_from_bytes is deprecated, use int.from_bytes instead
  from cryptography.utils import int_from_bytes
/usr/lib/python3/dist-packages/secretstorage/util.py:19: CryptographyDeprecationWarnin
g: int_from_bytes is deprecated, use int.from_bytes instead
  from cryptography.utils import int_from_bytes
Collecting matplotlib
  Downloading matplotlib-3.4.2-cp38-cp38-manylinux1_x86_64.whl (10.3 MB)
     |████████████████████████████████| 10.3 MB 5.9 MB/s
Requirement already satisfied: kiwisolver>=1.0.1 in /home/nikhil/.local/lib/python3.8/
site-packages (from matplotlib) (1.3.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/lib/python3/dist-packages (from m
atplotlib) (7.0.0)
Requirement already satisfied: cycler>=0.10 in /home/nikhil/.local/lib/python3.8/site-
packages (from matplotlib) (0.10.0)
Requirement already satisfied: python-dateutil>=2.7 in /home/nikhil/.local/lib/python3
.8/site-packages (from matplotlib) (2.8.1)
Requirement already satisfied: numpy>=1.16 in /home/nikhil/.local/lib/python3.8/site-p
ackages (from matplotlib) (1.20.1)
Requirement already satisfied: pyparsing>=2.2.1 in /usr/local/lib/python3.8/dist-packa
ges (from matplotlib) (2.4.7)
Requirement already satisfied: six in /home/nikhil/.local/lib/python3.8/site-packages
```

Pyplot

Pyplot is a Matplotlib module that provides a MATLAB-like interface. Matplotlib is designed to be as usable as MATLAB, with the ability to use Python and the advantage of being free and open-source. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc. The various plots we can utilize using Pyplot are Line Plot, Histogram, Scatter, 3D Plot, Image, Contour, and Polar.
After knowing a brief about Matplotlib and pyplot let’s see how to create a simple plot.

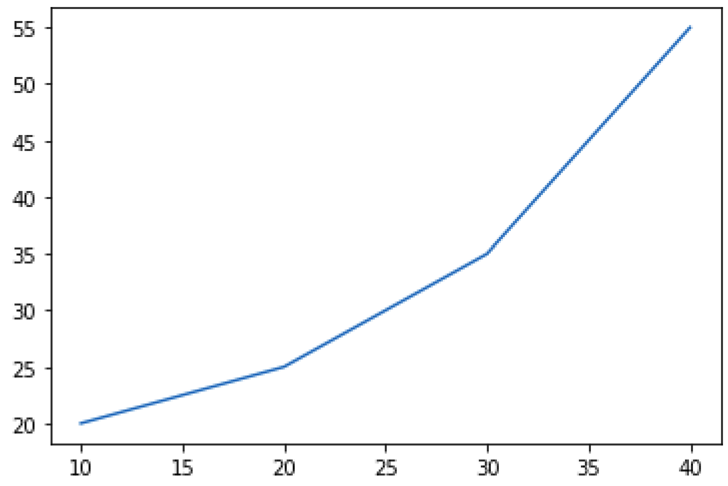
Example:

- Python3

```
import matplotlib.pyplot as plt



# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]
```

```
# plotting the data

plt.plot(x, y)



plt.show()
```

Output:



Now let see how to add some basic elements like title, legends, labels to the graph.

Note: For more information about Pyplot, refer Pyplot in Matplotlib

Adding Title

The title() method in matplotlib module is used to specify the title of the visualization depicted and displays the title using various attributes.

Syntax:

*matplotlib.pyplot.title(label, fontdict=None, loc='center', pad=None, **kwargs)*

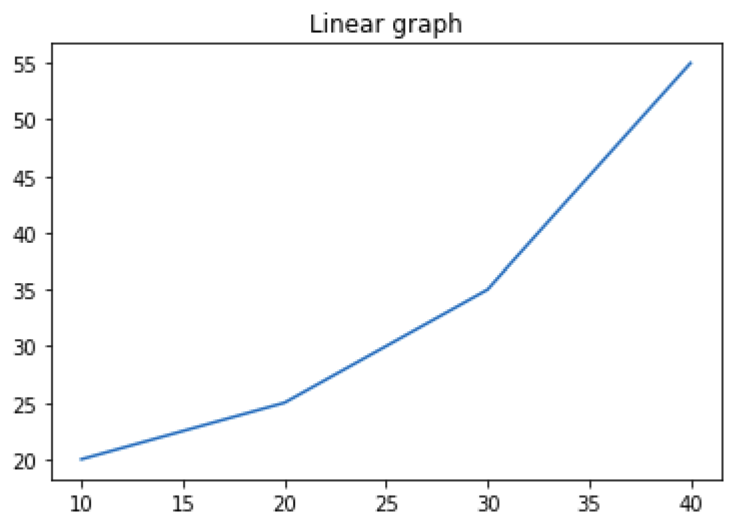
Example:

- Python3

```
import matplotlib.pyplot as plt



# initializing the data
```

```
x = [10, 20, 30, 40]

y = [20, 25, 35, 55]



# plotting the data

plt.plot(x, y)



# Adding title to the plot

plt.title("Linear graph")



plt.show()
```

Output:



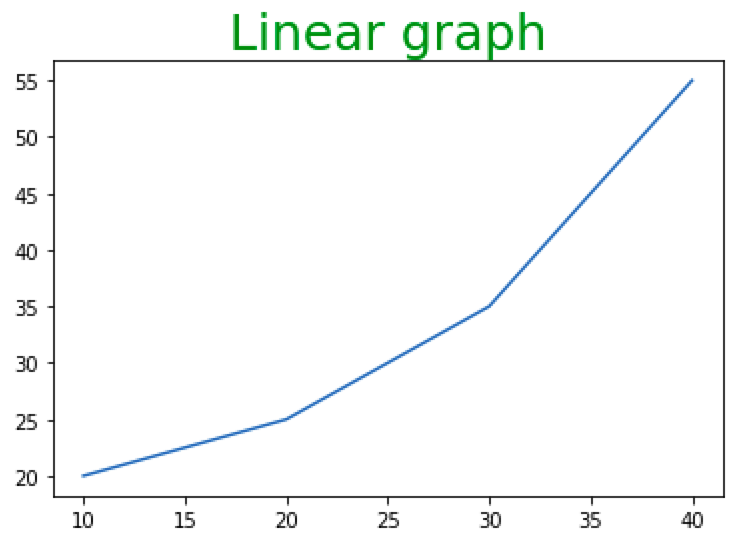
We can also change the appearance of the title by using the parameters of this function.

Example:

- Python3

```
import matplotlib.pyplot as plt


# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]


# plotting the data

plt.plot(x, y)


# Adding title to the plot

plt.title("Linear graph", fontsize=25, color="green")


plt.show()
```

Output:



Note: For more information about adding the title and its customization, refer Matplotlib.pyplot.title() in Python

Adding X Label and Y Label

In layman's terms, the X label and the Y label are the titles given to X-axis and Y-axis respectively. These can be added to the graph by using the **xlabel()** and **ylabel()** methods.

Syntax:

*matplotlib.pyplot.xlabel(xlabel, fontdict=None, labelpad=None, **kwargs)*

*matplotlib.pyplot.ylabel(ylabel, fontdict=None, labelpad=None, **kwargs)*

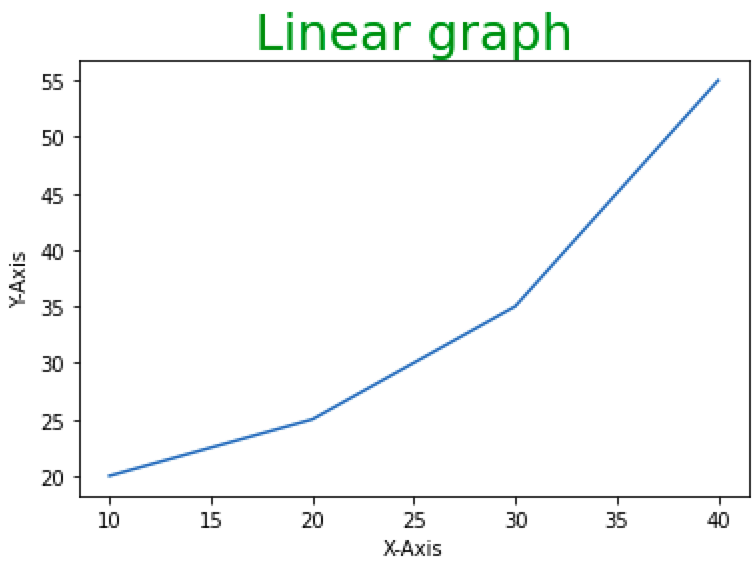
Example:

- Python3

```
import matplotlib.pyplot as plt



# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]
```

```
# plotting the data

plt.plot(x, y)



# Adding title to the plot

plt.title("Linear graph", fontsize=25, color="green")



# Adding label on the y-axis

plt.ylabel('Y-Axis')



# Adding label on the x-axis

plt.xlabel('X-Axis')



plt.show()
```

Output:



Setting Limits and Tick labels

You might have seen that Matplotlib automatically sets the values and the markers(points) of the X and Y axis, however, it is possible to set the limit and markers manually. **xlim()** and **ylim()** functions are used to set the limits of the X-axis and Y-axis respectively. Similarly, **xticks()** and **yticks()** functions are used to set tick labels.

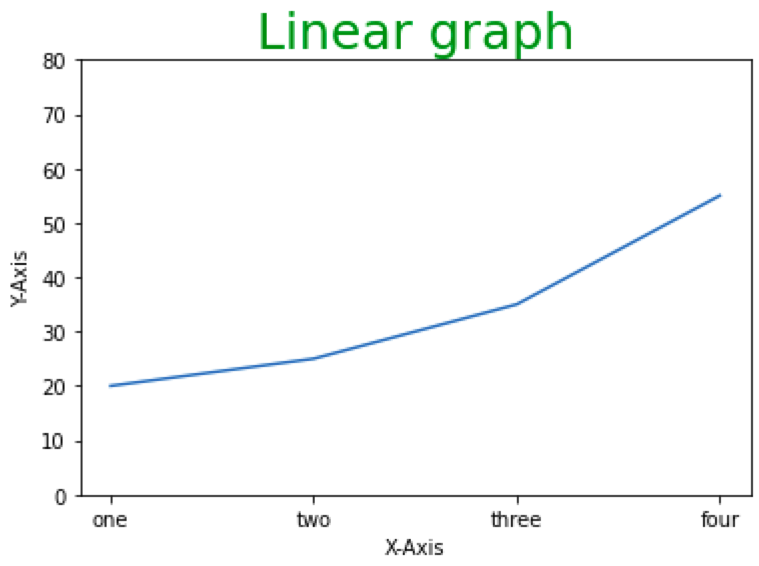
Example: In this example, we will be changing the limit of Y-axis and will be setting the labels for X-axis.

- Python3

```
import matplotlib.pyplot as plt



# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]



# plotting the data
```

```
plt.plot(x, y)



# Adding title to the plot

plt.title("Linear graph", fontsize=25, color="green")



# Adding label on the y-axis

plt.ylabel('Y-Axis')



# Adding label on the x-axis

plt.xlabel('X-Axis')



# Setting the limit of y-axis

plt.ylim(0, 80)



# setting the labels of x-axis

plt.xticks(x, labels=["one", "two", "three", "four"])



plt.show()
```

Output:



Adding Legends

A legend is an area describing the elements of the graph. In simple terms, it reflects the data displayed in the graph's Y-axis. It generally appears as the box containing a small sample of each color on the graph and a small description of what this data means.

The attribute bbox_to_anchor=(x, y) of legend() function is used to specify the coordinates of the legend, and the attribute ncol represents the number of columns that the legend has. Its default value is 1.

Syntax:
matplotlib.pyplot.legend([“name1”, “name2”], bbox_to_anchor=(x, y), ncol=1)

Example:

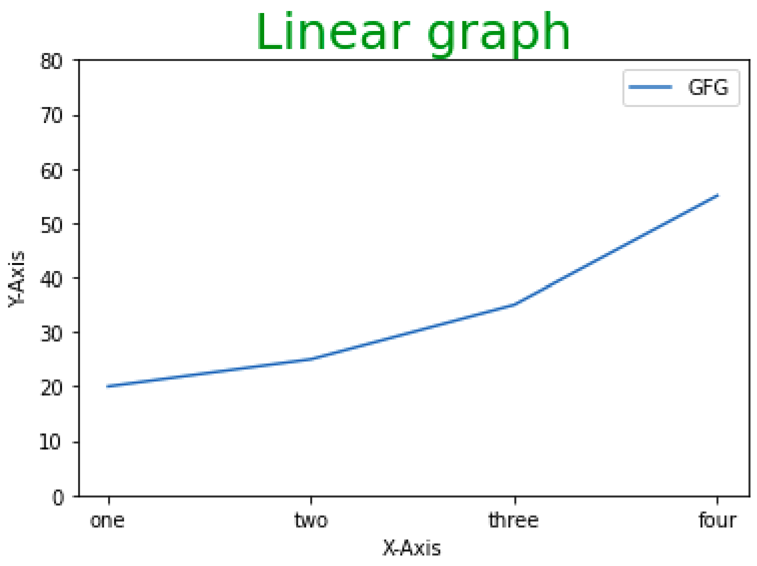
- Python3

```
import matplotlib.pyplot as plt



# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]
```

```
# plotting the data

plt.plot(x, y)



# Adding title to the plot

plt.title("Linear graph", fontsize=25, color="green")



# Adding label on the y-axis

plt.ylabel('Y-Axis')



# Adding label on the x-axis

plt.xlabel('X-Axis')



# Setting the limit of y-axis

plt.ylim(0, 80)



# setting the labels of x-axis

plt.xticks(x, labels=["one", "two", "three", "four"])
```

```
# Adding legends

plt.legend(["GFG"])



plt.show()
```

Output:



Before moving any further with Matplotlib let's discuss some important classes that will be used further in the tutorial. These classes are –

- **Figure**
- **Axes**

Note: Matplotlib take care of the creation of inbuilt defaults like Figure and Axes.

Figure class

Consider the figure class as the overall window or page on which everything is drawn. It is a top-level container that contains one or more axes. A figure can be created using the **figure()** method.

Syntax:
class matplotlib.figure.Figure(figsize=None, dpi=None, facecolor=None, edgecolor=None, linewidth=0.0, frameon=None, subplotpars=None, tight_layout=None, constrained_layout=None)

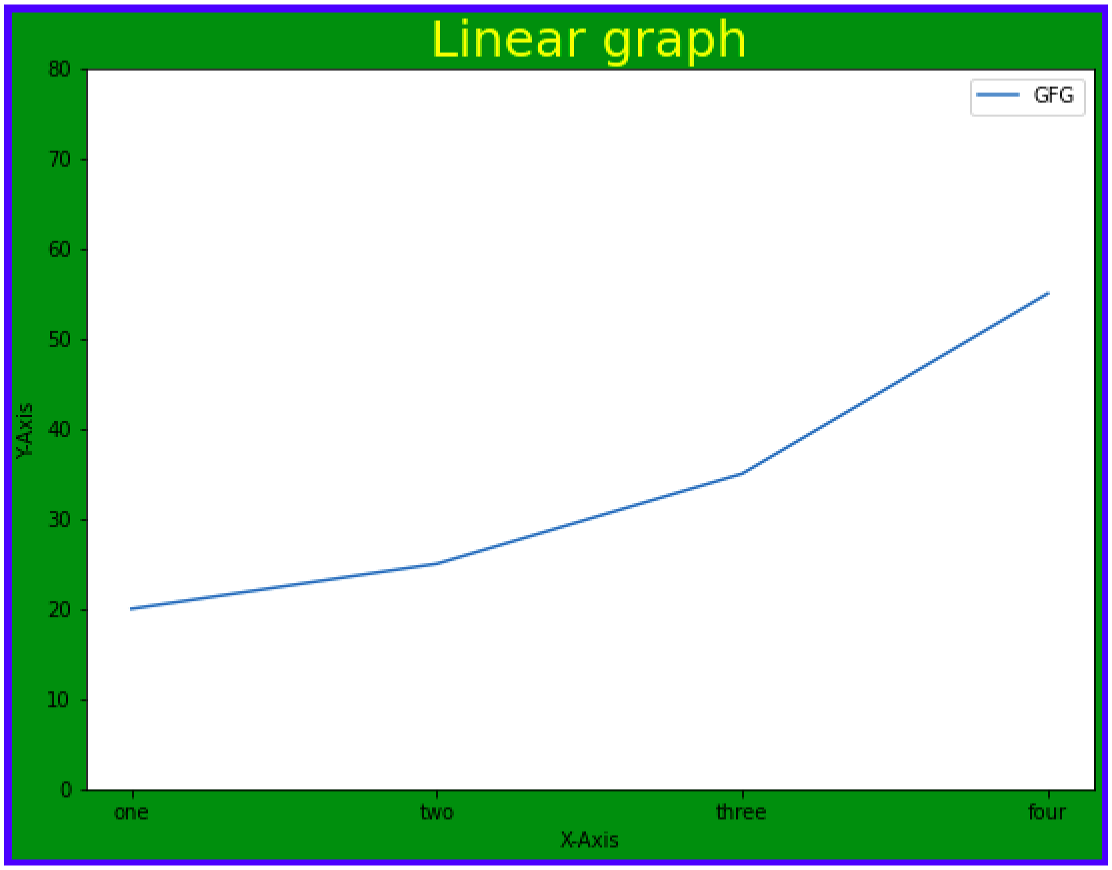
Example:

- Python3

```
# Python program to show pyplot module

import matplotlib.pyplot as plt

from matplotlib.figure import Figure



# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]



# Creating a new figure with width = 7 inches

# and height = 5 inches with face color as

# green, edgecolor as red and the line width

# of the edge as 7

fig = plt.figure(figsize =(7, 5), facecolor='g',

                 edgecolor='b', linewidth=7)



# Creating a new axes for the figure

ax = fig.add_axes([1, 1, 1, 1])



# Adding the data to be plotted
```

```
ax.plot(x, y)



# Adding title to the plot

plt.title("Linear graph", fontsize=25, color="yellow")



# Adding label on the y-axis

plt.ylabel('Y-Axis')



# Adding label on the x-axis

plt.xlabel('X-Axis')



# Setting the limit of y-axis

plt.ylim(0, 80)



# setting the labels of x-axis

plt.xticks(x, labels=["one", "two", "three", "four"])



# Adding legends

plt.legend(["GFG"])
plt.show()
```

Output:



>>> **More Functions in Figure Class**

Axes Class

Axes class is the most basic and flexible unit for creating sub-plots. A given figure may contain many axes, but a given axes can only be present in one figure. The axes() function creates the axes object.

Syntax:

axes([left, bottom, width, height])

Just like pyplot class, axes class also provides methods for adding titles, legends, limits, labels, etc. Let’s see a few of them –

- **Adding Title –** ax.set_title()
- **Adding X Label and Y label –** ax.set_xlabel(), ax.set_ylabel()
- **Setting Limits –** ax.set_xlim(), ax.set_ylim()
- **Tick labels –** ax.set_xticklabels(), ax.set_yticklabels()
- **Adding Legends –** ax.legend()

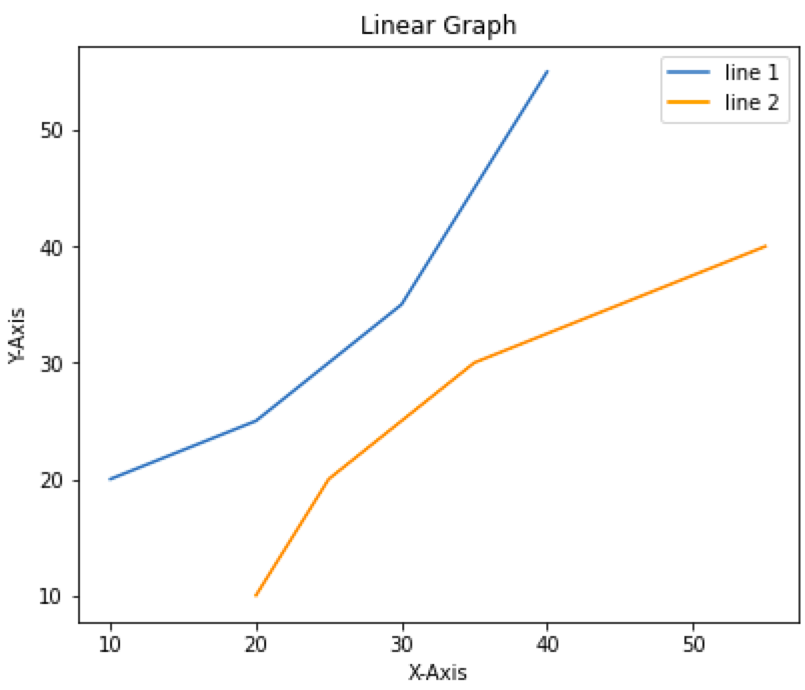
Example:

- Python3

```
# Python program to show pyplot module

import matplotlib.pyplot as plt

from matplotlib.figure import Figure



# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]



fig = plt.figure(figsize = (5, 4))



# Adding the axes to the figure

ax = fig.add_axes([1, 1, 1, 1])



# plotting 1st dataset to the figure

ax1 = ax.plot(x, y)



# plotting 2nd dataset to the figure

ax2 = ax.plot(y, x)
```

```
# Setting Title

ax.set_title("Linear Graph")



# Setting Label

ax.set_xlabel("X-Axis")

ax.set_ylabel("Y-Axis")



# Adding Legend

ax.legend(labels = ('line 1', 'line 2'))



plt.show()
```

Output:



Multiple Plots

We have learned about the basic components of a graph that can be added so that it can convey more information. One method can be by calling the plot function again and again with a different set of values as shown in the above example. Now let's see how to plot multiple graphs using some functions and also how to plot subplots.

Method 1: Using the add_axes() method
The add_axes() method is used to add axes to the figure. This is a method of figure class

Syntax:
*add_axes(self, *args, **kwargs)*

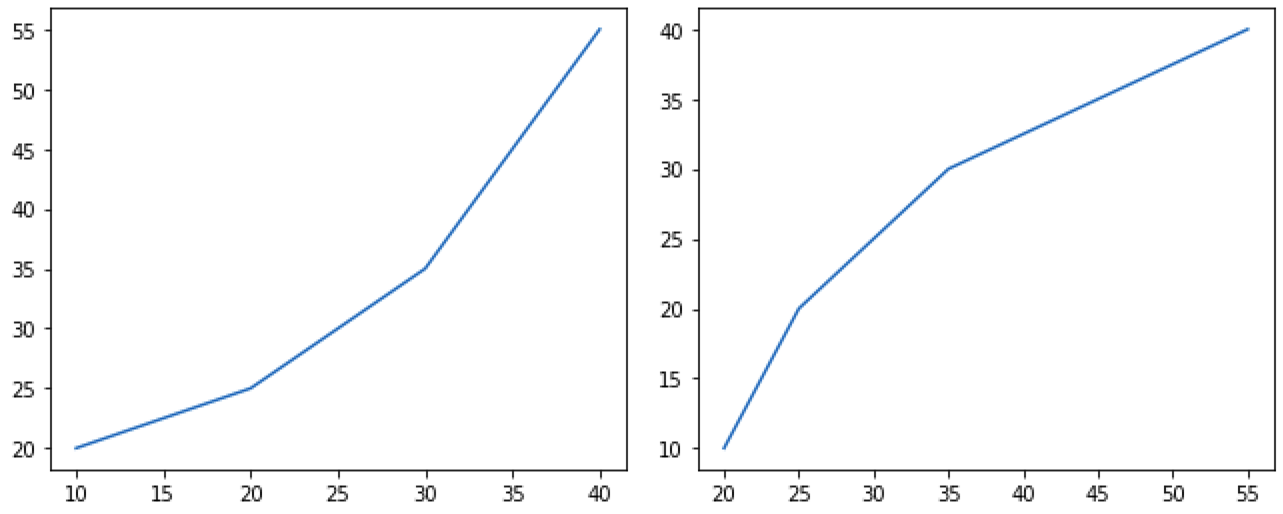
Example:

- Python3

```
# Python program to show pyplot module

import matplotlib.pyplot as plt

from matplotlib.figure import Figure
```

```
# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]



# Creating a new figure with width = 5 inches

# and height = 4 inches

fig = plt.figure(figsize =(5, 4))



# Creating first axes for the figure

ax1 = fig.add_axes([0.1, 0.1, 0.8, 0.8])



# Creating second axes for the figure

ax2 = fig.add_axes([1, 0.1, 0.8, 0.8])



# Adding the data to be plotted

ax1.plot(x, y)

ax2.plot(y, x)



plt.show()
```

Output:



Method 2: Using subplot() method.
This method adds another plot at the specified grid position in the current figure.

Syntax:
*subplot(nrows, ncols, index, **kwargs)*

*subplot(pos, **kwargs)*

subplot(ax)

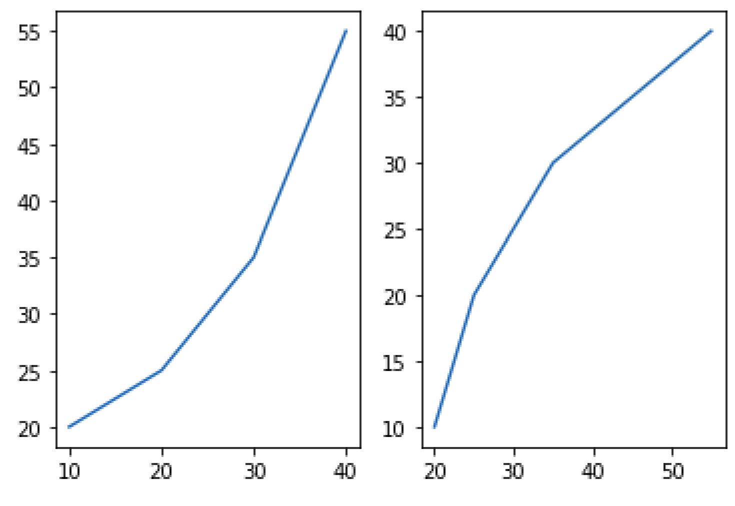
Example:

- Python3

```
import matplotlib.pyplot as plt



# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]
```

```
# Creating figure object

plt.figure()



# adding first subplot

plt.subplot(121)

plt.plot(x, y)



# adding second subplot

plt.subplot(122)

plt.plot(y, x)
```

Output:



Method 3: Using subplots() method
This function is used to create figures and multiple subplots at the same time.

Syntax:

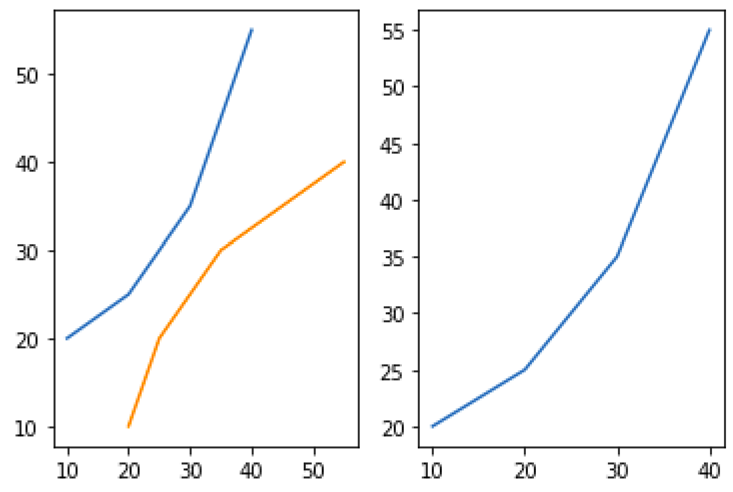
*matplotlib.pyplot.subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True, subplot_kw=None, gridspec_kw=None, **fig_kw)*

Example:

- Python3

```
import matplotlib.pyplot as plt



# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]


# Creating the figure and subplots

# according the argument passed

fig, axes = plt.subplots(1, 2)


# plotting the data in the

# 1st subplot

axes[0].plot(x, y)


# plotting the data in the 1st
# subplot only

axes[0].plot(y, x)



# plotting the data in the 2nd

# subplot only

axes[1].plot(x, y)
```

Output:



Method 4: Using subplot2grid() method
This function creates axes object at a specified location inside a grid and also helps in spanning the axes object across multiple rows or columns. In simpler words, this function is used to create multiple charts within the same figure.

Syntax:
Plt.subplot2grid(shape, location, rowspan, colspan)

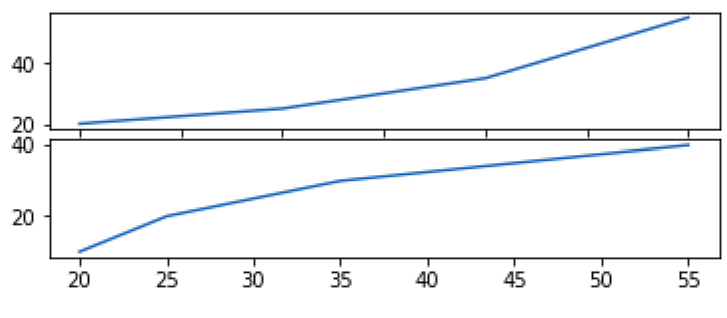
Example:

- Python3

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

```
# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]



# adding the subplots

axes1 = plt.subplot2grid (

(7, 1), (0, 0), rowspan = 2, colspan = 1)



axes2 = plt.subplot2grid (

(7, 1), (2, 0), rowspan = 2, colspan = 1)



# plotting the data

axes1.plot(x, y)

axes2.plot(y, x)
```

Output:



Different types of Matplotlib Plots

Matplotlib supports a variety of plots including line charts, bar charts, histograms, scatter plots, etc. We will discuss the most commonly used charts in this article with the help of some good examples and will also see how to customize each plot.

Note: Some elements like axis, color are common to each plot whereas some elements are pot specific.

Line Chart

Line chart is one of the basic plots and can be created using the **plot()** function. It is used to represent a relationship between two data X and Y on a different axis.
Syntax:
*matplotlib.pyplot.plot(*args, scalex=True, scaley=True, data=None, **kwargs)*

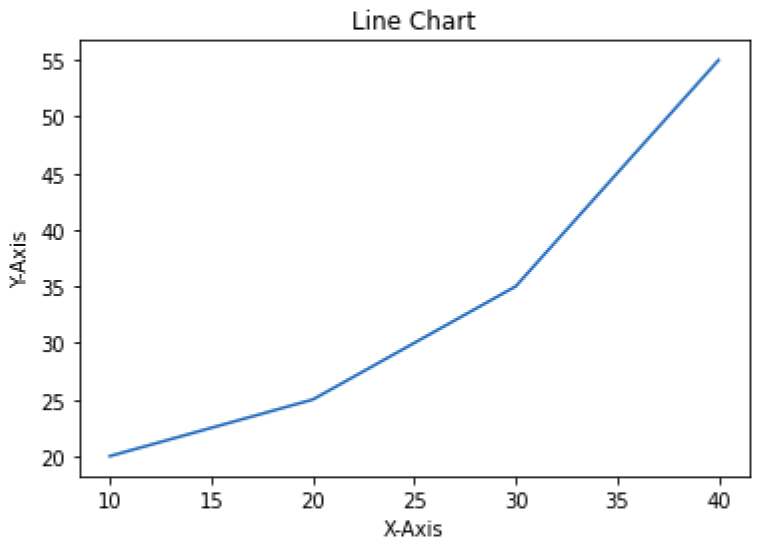
Example:

- Python3

```
import matplotlib.pyplot as plt



# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]


# plotting the data

plt.plot(x, y)


# Adding title to the plot

plt.title("Line Chart")
```

```
# Adding label on the y-axis

plt.ylabel('Y-Axis')



# Adding label on the x-axis

plt.xlabel('X-Axis')



plt.show()
```

Output:



Let’s see how to customize the above-created line chart. We will be using the following properties –

- **color:** Changing the color of the line
- **linewidth:** Customizing the width of the line
- **marker:** For changing the style of actual plotted point
- **markersize:** For changing the size of the markers
- **linestyle:** For defining the style of the plotted line

Plotting x and y points

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

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

The function takes parameters for specifying points in the diagram.

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

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

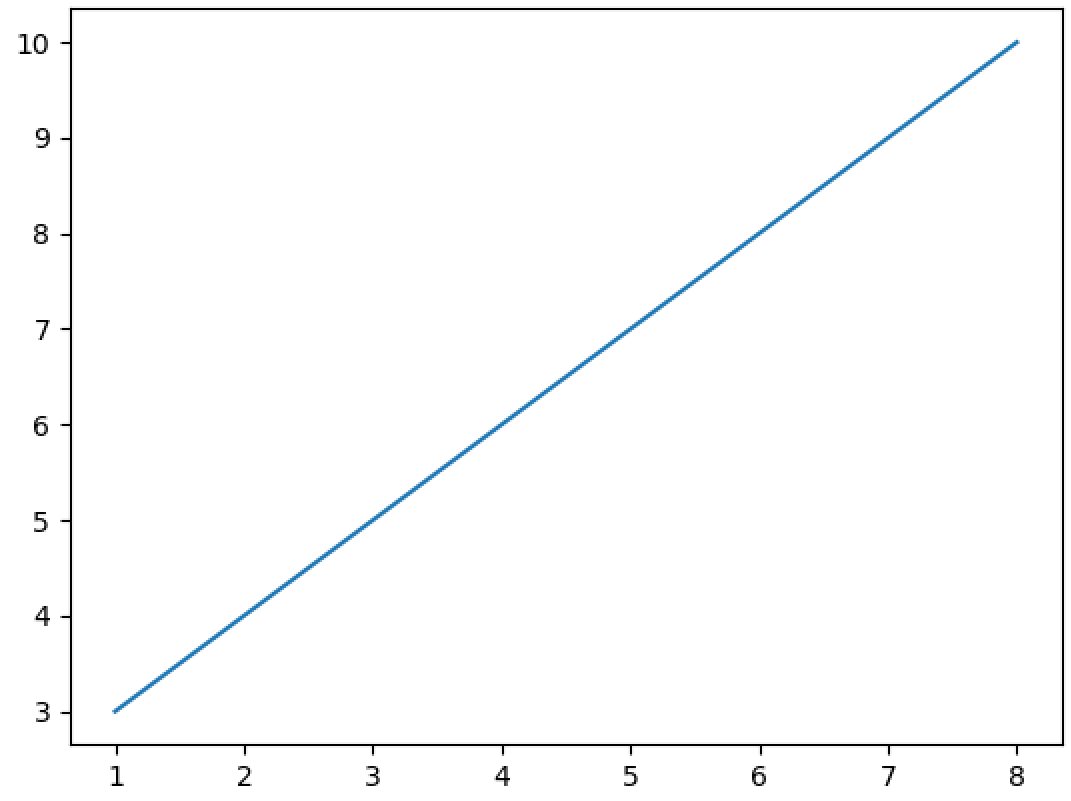
If we need to plot a line from (1, 3) to (8, 10), we have to pass two arrays [1, 8] and [3, 10] to the plot function.

ExampleGet your own Python Server

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

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

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

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

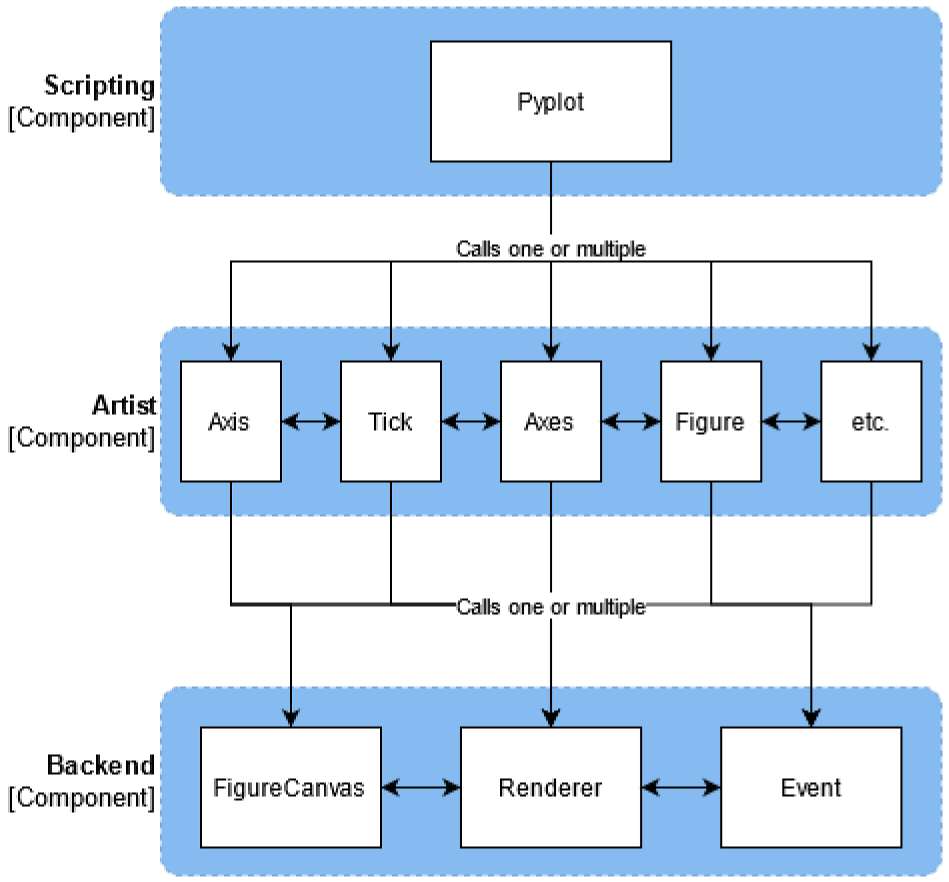
Result:



The **x-axis** is the horizontal axis.

The **y-axis** is the vertical axis.

matplotlib architecture



Different Linestyle available

Character	Definition
_	Solid line
__	Dashed line

Character	**Definition**
-.	dash-dot line
:	Dotted line
.	Point marker
o	Circle marker
,	Pixel marker
v	triangle_down marker
^	triangle_up marker
<	triangle_left marker
>	triangle_right marker
1	tri_down marker
2	tri_up marker
3	tri_left marker
4	tri_right marker
s	square marker

Character	Definition
p	pentagon marker
*****	star marker
h	hexagon1 marker
H	hexagon2 marker
+	Plus marker
x	X marker
D	Diamond marker
d	thin_diamond marker
|	vline marker
_	hline marker

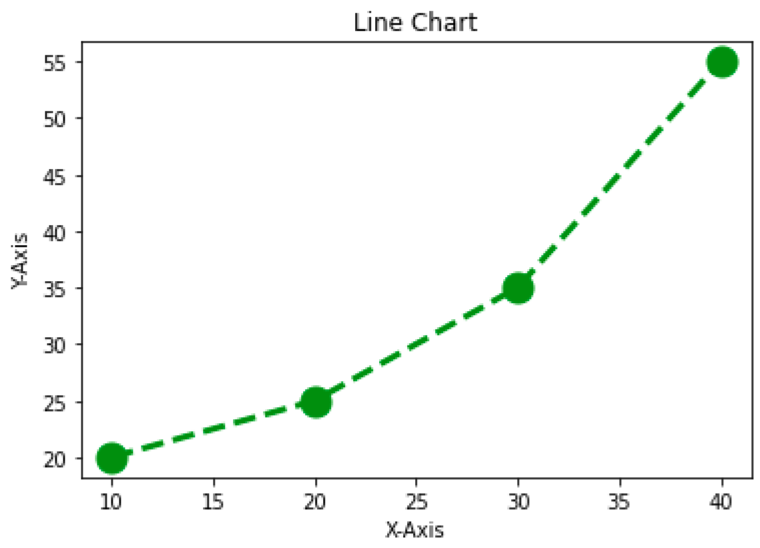
Example:

- Python3

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

```
# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]



# plotting the data

plt.plot(x, y, color='green', linewidth=3, marker='o',

         markersize=15, linestyle='--')



# Adding title to the plot

plt.title("Line Chart")



# Adding label on the y-axis

plt.ylabel('Y-Axis')



# Adding label on the x-axis

plt.xlabel('X-Axis')



plt.show()
```

Output:



Note: For more information, refer Line plot styles in Matplotlib

Bar Chart

A **bar chart** is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. The bar plots can be plotted horizontally or vertically. A bar chart describes the comparisons between the discrete categories. It can be created using the bar() method.
In the below example, we will use the tips dataset. Tips database is the record of the tip given by the customers in a restaurant for two and a half months in the early 1990s. It contains 6 columns as total_bill, tip, sex, smoker, day, time, size.

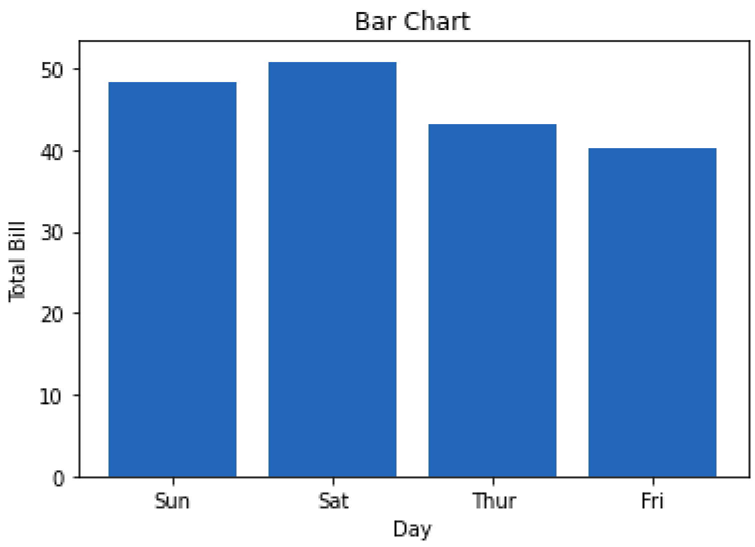
Example:

- Python3

```
import matplotlib.pyplot as plt

import pandas as pd



# Reading the tips.csv file

data = pd.read_csv('tips.csv')
```

```
# initializing the data

x = data['day']

y = data['total_bill']



# plotting the data

plt.bar(x, y)



# Adding title to the plot

plt.title("Tips Dataset")



# Adding label on the y-axis

plt.ylabel('Total Bill')



# Adding label on the x-axis

plt.xlabel('Day')



plt.show()
```

Output:



Customization that is available for the Bar Chart –

- **color:** For the bar faces
- **edgecolor:** Color of edges of the bar
- **linewidth:** Width of the bar edges
- **width:** Width of the bar

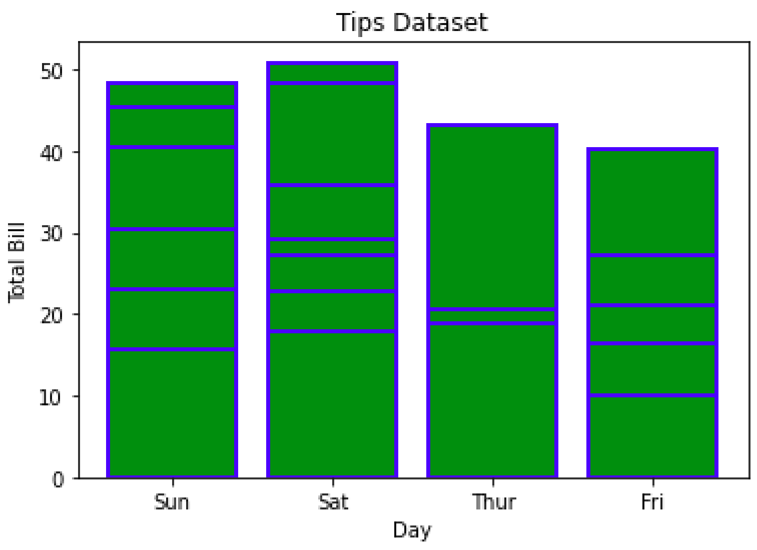
Example:

- Python3

```
import matplotlib.pyplot as plt

import pandas as pd



# Reading the tips.csv file

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



# initializing the data

x = data['day']
```

```
y = data['total_bill']



# plotting the data

plt.bar(x, y, color='green', edgecolor='blue',

        linewidth=2)



# Adding title to the plot

plt.title("Tips Dataset")



# Adding label on the y-axis

plt.ylabel('Total Bill')



# Adding label on the x-axis

plt.xlabel('Day')



plt.show()
```

Output:



Note: The lines in between the bars refer to the different values in the Y-axis of the particular value of the X-axis.

Histogram

A **histogram** is basically used to represent data provided in a form of some groups. It is a type of bar plot where the X-axis represents the bin ranges while the Y-axis gives information about frequency. The **hist()** function is used to compute and create histogram of x.

Syntax:

*matplotlib.pyplot.hist(x, bins=None, range=None, density=False, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, *, data=None, **kwargs)*

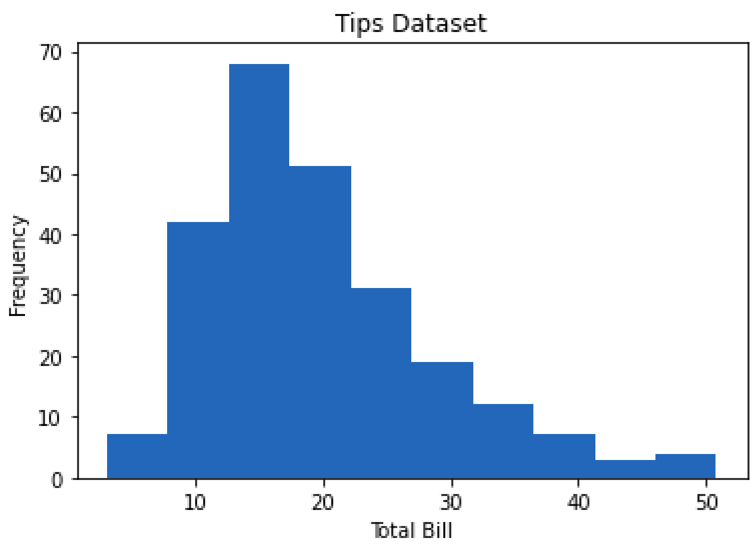
Example:

- Python3

```
import matplotlib.pyplot as plt

import pandas as pd



# Reading the tips.csv file

data = pd.read_csv('tips.csv')
```

```
# initializing the data

x = data['total_bill']



# plotting the data

plt.hist(x)



# Adding title to the plot

plt.title("Tips Dataset")



# Adding label on the y-axis

plt.ylabel('Frequency')



# Adding label on the x-axis

plt.xlabel('Total Bill')



plt.show()
```

Output:



Customization that is available for the Histogram –

- **bins:** Number of equal-width bins
- **color:** For changing the face color
- **edgecolor:** Color of the edges
- **linestyle:** For the edgelines
- **alpha:** blending value, between 0 (transparent) and 1 (opaque)

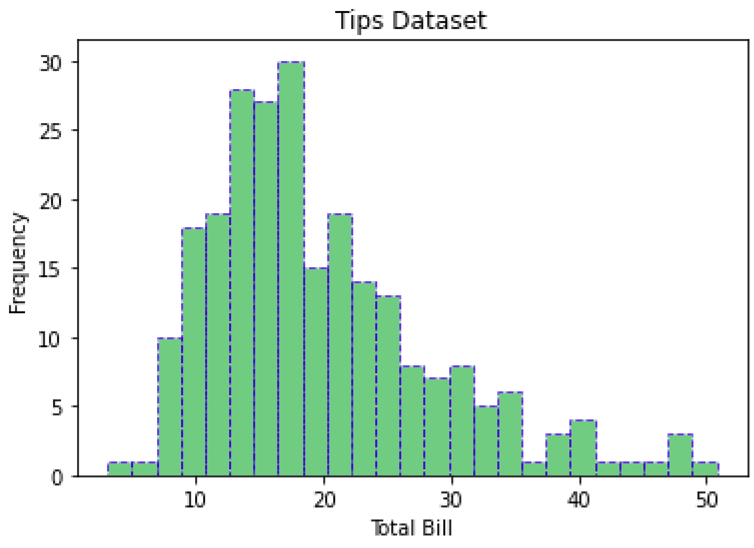
Example:

- Python3

```
import matplotlib.pyplot as plt

import pandas as pd



# Reading the tips.csv file

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



# initializing the data

x = data['total_bill']
```

```
# plotting the data

plt.hist(x, bins=25, color='green', edgecolor='blue',

        linestyle='--', alpha=0.5)



# Adding title to the plot

plt.title("Tips Dataset")



# Adding label on the y-axis

plt.ylabel('Frequency')



# Adding label on the x-axis

plt.xlabel('Total Bill')



plt.show()
```

Output:



Scatter Plot

Scatter plots are used to observe relationships between variables. The **scatter()** method in the matplotlib library is used to draw a scatter plot.
Syntax:
matplotlib.pyplot.scatter(x_axis_data, y_axis_data, s=None, c=None, marker=None, cmap=None, vmin=None, vmax=None, alpha=None, linewidths=None, edgecolors=None

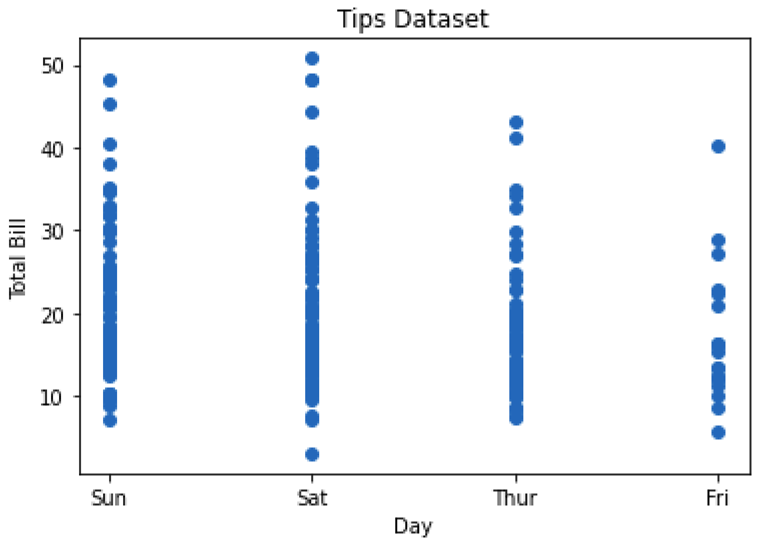
Example:

- Python3

```
import matplotlib.pyplot as plt

import pandas as pd



# Reading the tips.csv file

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



# initializing the data
```

```
x = data['day']

y = data['total_bill']



# plotting the data

plt.scatter(x, y)



# Adding title to the plot

plt.title("Tips Dataset")



# Adding label on the y-axis

plt.ylabel('Total Bill')



# Adding label on the x-axis

plt.xlabel('Day')



plt.show()
```

Output:



Customizations that are available for the scatter plot are –

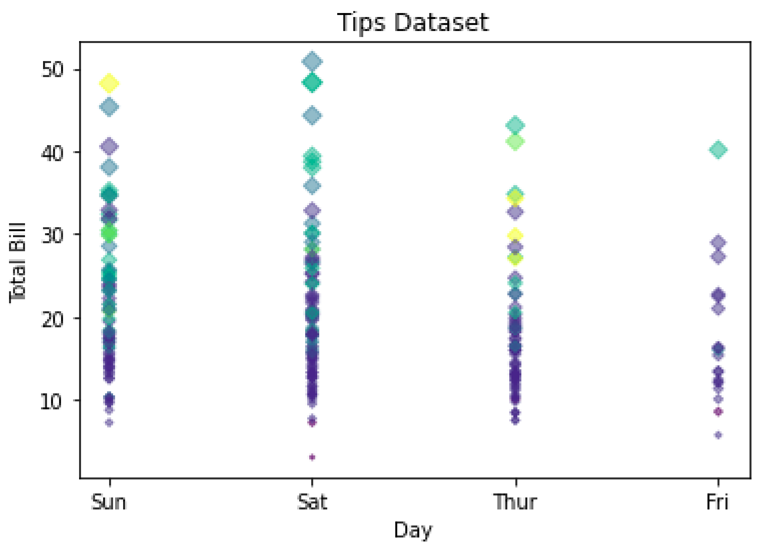
- **s:** marker size (can be scalar or array of size equal to size of x or y)
- **c:** color of sequence of colors for markers
- **marker:** marker style
- **linewidths:** width of marker border
- **edgecolor:** marker border color
- **alpha:** blending value, between 0 (transparent) and 1 (opaque)

- Python3

```
import matplotlib.pyplot as plt

import pandas as pd



# Reading the tips.csv file

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



# initializing the data
```

```
x = data['day']

y = data['total_bill']



# plotting the data

plt.scatter(x, y, c=data['size'], s=data['total_bill'],

            marker='D', alpha=0.5)



# Adding title to the plot

plt.title("Tips Dataset")



# Adding label on the y-axis

plt.ylabel('Total Bill')



# Adding label on the x-axis

plt.xlabel('Day')



plt.show()
```

Output:



Pie Chart

Pie chart is a circular chart used to display only one series of data. The area of slices of the pie represents the percentage of the parts of the data. The slices of pie are called wedges. It can be created using the pie() method.

Syntax:

matplotlib.pyplot.pie(data, explode=None, labels=None, colors=None, autopct=None, shadow=False)

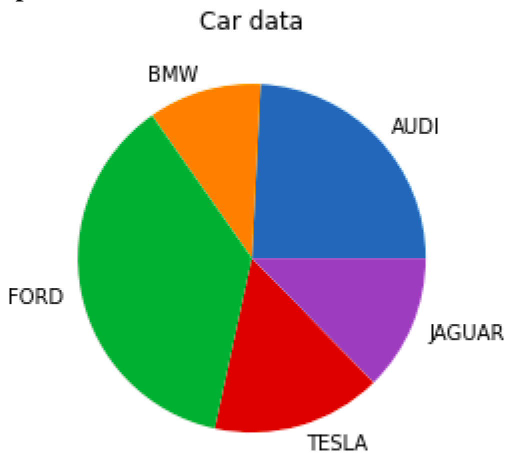
Example:

- Python3

```
import matplotlib.pyplot as plt

import pandas as pd



# Reading the tips.csv file

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



# initializing the data
```

```
cars = ['AUDI', 'BMW', 'FORD',

        'TESLA', 'JAGUAR',]

data = [23, 10, 35, 15, 12]



# plotting the data

plt.pie(data, labels=cars)



# Adding title to the plot

plt.title("Car data")



plt.show()
```

Output:



Customizations that are available for the Pie chart are –

- **explode:** Moving the wedges of the plot
- **autopct:** Label the wedge with their numerical value.

- **color:** Attribute is used to provide color to the wedges.
- **shadow:** Used to create shadow of wedge.

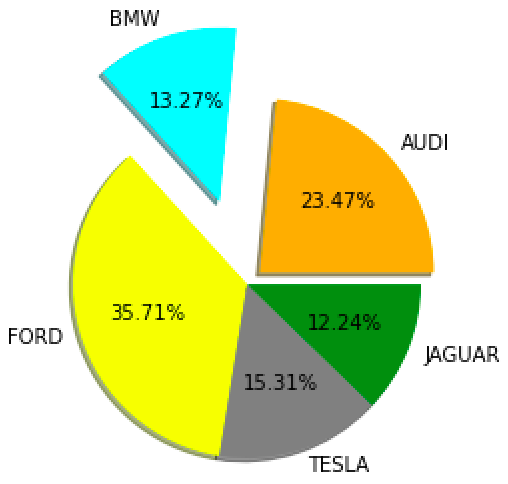
Example:

- Python3

```
import matplotlib.pyplot as plt

import pandas as pd



# Reading the tips.csv file

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



# initializing the data

cars = ['AUDI', 'BMW', 'FORD',

        'TESLA', 'JAGUAR',]

data = [23, 13, 35, 15, 12]



explode = [0.1, 0.5, 0, 0, 0]



colors = ( "orange", "cyan", "yellow",

          "grey", "green",)
```

```
# plotting the data

plt.pie(data, labels=cars, explode=explode, autopct='%1.2f%%',

        colors=colors, shadow=True)



plt.show()
```

Output:



Saving a Plot

For saving a plot in a file on storage disk, savefig() method is used. A file can be saved in many formats like .png, .jpg, .pdf, etc.

Syntax:

pyplot.savefig(fname, dpi=None, facecolor='w', edgecolor='w', orientation='portrait', papertype=None, format=None, transparent=False, bbox_inches=None, pad_inches=0.1, frameon=None, metadata=None)

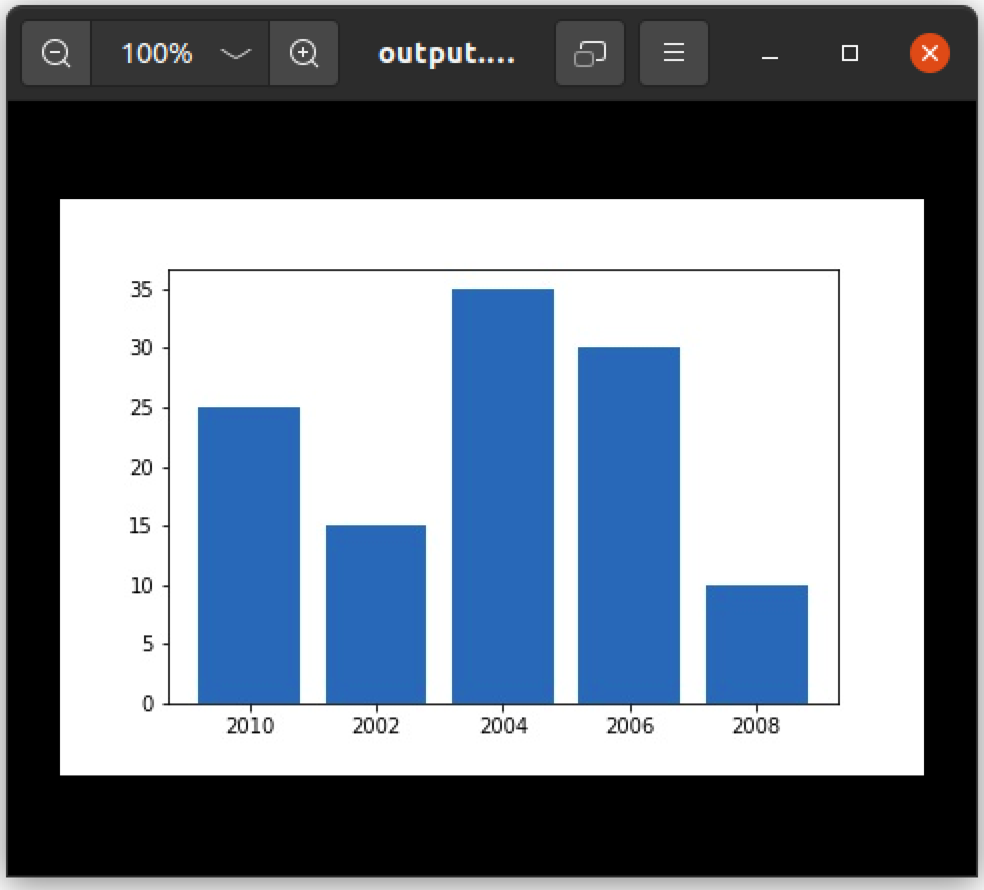
Example:

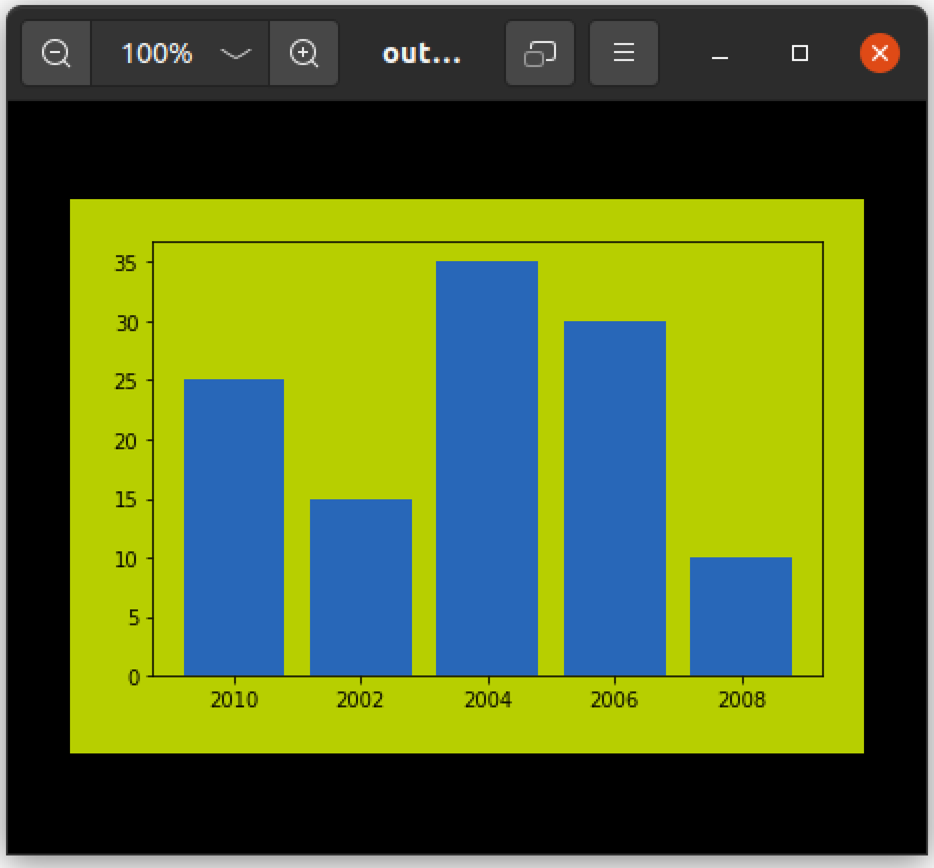
- Python3

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

```
# Creating data

year = ['2010', '2002', '2004', '2006', '2008']

production = [25, 15, 35, 30, 10]



# Plotting barchart

plt.bar(year, production)



# Saving the figure.

plt.savefig("output.jpg")



# Saving figure by changing parameter values

plt.savefig("output1", facecolor='y', bbox_inches="tight",

            pad_inches=0.3, transparent=True)
```

Output:


100%
output....
35
30
25
20
15
10
5
0
2010
2002
2004
2006
2008


100%
out...
35
30
25
20
15
10
5
0
2010
2002
2004
2006
2008