UNIT III:Numpy: Creating Arrays, Arrays Operations,

Multidimensional Arrays

Arrays transformation,

Array Concatenation,

Array Math Operations,

Multidimensional Array and its Operations,

Vector and Matrix.

Visualization: Visualization with matplotlib, Figures and subplots, Labeling and arranging figures, Outputting graphics.

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.

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

NumPy performs array-oriented computing.

It efficiently implements the multidimensional arrays.

It performs scientific computations.

It is capable of performing Fourier Transform and reshaping the data stored in multidimensional arrays.

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 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

```
import numpy
a = numpy.array
a
```

```
<function numpy.array>
```

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

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

```
array(<class 'object'>, dtype=object)
```

The parameters are described in the following table.

SN	Parameter	Description
	dtype	We can change the data type of the array elements by changing this option to the specified
		or A (any)

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

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

```
array([1, 2, 3])
```

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

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

```
array([[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.

Double-click (or enter) to edit

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

```
array([1.+0.j, 3.+0.j, 5.+0.j, 7.+0.j])
```

```
a = numpy.array([1, 3, 5, 7], float)
a
```

```
array([1., 3., 5., 7.])
```

Finding the dimensions of the Array

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

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

```
2
[[ 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.

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

```
Each item contains 8 bytes
144
```

```
import numpy as np
a = np.array([[1.2,2.5,3.6,5,7,9,9]])
print("Each item contains",a.itemsize,"bytes")
```

```
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

```
#finding the data type of each array item
import numpy as np
a = np.array([[1,2,3]])
print("Each item is of the type",a.dtype)
```

```
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.

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

```
Array Size: 7
Shape: (1, 7)
[[1 2 3 4 5 6 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.

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

```
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

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

```
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

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

```
[ 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.

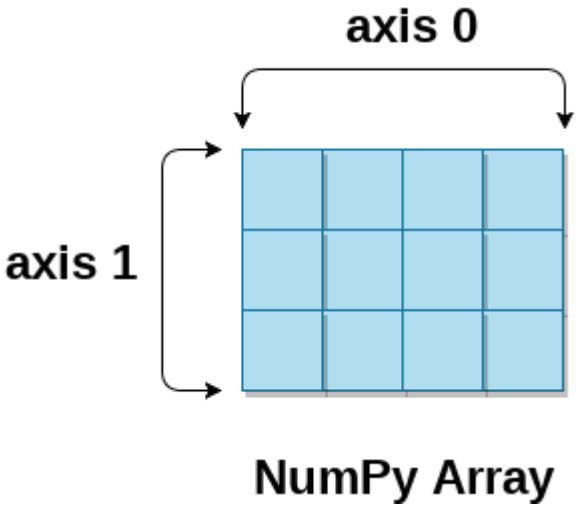
```
import numpy as np
a = np.array([1,2,3,10,15,4])
print("The array:",a)
print("The maximum element:",a.max())
print("The minimum element:",a.min())
print("The sum of the elements:",a.sum())
```

```
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

```
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))
```

```
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.

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

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

Array shape manipulation

Flattening

The numpy.ravel() functions returns contiguous flattened array(1D array with all the input-array elements and with the same type as it). A copy is made only if needed. Syntax :

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
a.ravel()
```

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

```
a.T
```

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

```
a.T.ravel()
```

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

Reshaping

Reshaping arrays

Reshaping means changing the shape of an array.

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

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

Reshape From 1-D to 2-D

Example Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

```
a.shape
```

```
(2, 3)
```

```
b = a.ravel()
b
```

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

```
b = b.reshape((2, 3))
b
```

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

```
a.reshape((2, -1))
```

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

```
a[0,0]=99
a
```

```
array([[99,  2,  3],
       [ 4,  5,  6]])
```

Joining NumPy Arrays

Joining means putting contents of two or more arrays in a single array.

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)
```

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

Example

Join two 2-D arrays along rows (axis=1):

```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)

print(arr)
```

```
[[1 2 5 6]
 [3 4 7 8]]
```

Joining Arrays Using Stack Functions

Stacking is same as concatenation, the only difference is that stacking is done along a new axis.

We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.stack((arr1, arr2), axis=1)

print(arr)
```

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

Splitting NumPy Arrays

Splitting is reverse operation of Joining.

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

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

Example Split the array in 3 parts:

```
import numpy as np

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

newarr = np.array_split(arr, 3)

print(newarr)
```

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

Searching Arrays

we can search an array for a certain value, and return the indexes that get a match.

To search an array, use the where() method.

```
#Find the indexes where the value is 4:

import numpy as np

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

x = np.where(arr == 4)

print(x)
```

```
(array([3, 5, 6]),)
```

Example

Find the indexes where the values are even:

```
import numpy as np

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

x = np.where(arr%2 == 0)

print(x)
```

```
(array([1, 3, 5, 7]),)
```

Example

Find the indexes where the values are odd:

```
import numpy as np

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

x = np.where(arr%2 == 1)

print(x)
```

```
(array([0, 2, 4, 6]),)
```

Conditional Selection Using NumPy Arrays

```
import numpy as np

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

```
[False False False False False False  True  True]
```

we can also generate a new array of values that satisfy this condition by passing the condition into the square brackets (just like we do for indexing).

An example of this is below:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
b= arr[arr > 6]
print(b)
print(type(b))
```

```
[7 8]
<class 'numpy.ndarray'>
```

The example above will return a tuple: (array([3, 5, 6],)

Which means that the value 4 is present at index 3, 5, and 6.

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.

```
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)
```

```
Sum of array a and b
 [[ 2  4 33]
 [22 34 33]]
Product of array a and b
 [[  1   4  90]
 [120 285 116]]
Division of array a and b
 [[ 1.          1.         10.        ]
 [ 0.83333333  0.78947368  0.13793103]]
```

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)))
```

```
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]]
```

NumPy Datatypes

The NumPy provides a higher range of numeric data types than that provided by the Python. A list of numeric data types is given in the following table.

1	bool_	It represents the boolean value indicating true or false. It is stored as a byte.
2	int_	It is the default type of integer. It is identical to long type in C that contains 64 bit or 32-bit integer.
3	intc	It is similar to the C integer (c int) as it represents 32 or 64-bit int.
4	intp	It represents the integers which are used for indexing.
5	int8	It is the 8-bit integer identical to a byte. The range of the value is -128 to 127.
6	int16	It is the 2-byte (16-bit) integer. The range is -32768 to 32767.
7	int32	It is the 4-byte (32-bit) integer. The range is -2147483648 to 2147483647.
8	int64	It is the 8-byte (64-bit) integer. The range is -9223372036854775808 to 9223372036854775807.
9	uint8	It is the 1-byte (8-bit) unsigned integer.
10	uint16	It is the 2-byte (16-bit) unsigned integer.
11	uint32	It is the 4-byte (32-bit) unsigned integer.
12	uint64	It is the 8 bytes (64-bit) unsigned integer.
13	float_	It is identical to float64.
14	float16	It is the half-precision float. 5 bits are reserved for the exponent. 10 bits are reserved for mantissa, and 1 bit is reserved for the sign.
15	float32	It is a single precision float. 8 bits are reserved for the exponent, 23 bits are reserved for mantissa, and 1 bit is reserved for the sign.
16	float64	It is the double precision float. 11 bits are reserved for the exponent, 52 bits are reserved for mantissa, 1 bit is used for the sign.
17	complex_	It is identical to complex128.
18	complex64	It is used to represent the complex number where real and imaginary part shares 32 bits each.

NumPy dtype

All the items of a numpy array are data type objects also known as numpy dtypes. A data type object implements the fixed size of memory corresponding to an array.

We can create a dtype object by using the following syntax.

numpy.dtype(object, align, copy) *

The constructor accepts the following object.

Object: It represents the object which is to be converted to the data type.

Align: It can be set to any boolean value. If true, then it adds extra padding to make it equivalent to a C struct.

Copy: It creates another copy of the dtype object.

```
import numpy as np
d = np.dtype(np.int32)
print(d)
```

```
int32
```

Double-click (or enter) to edit

Creating a Structured data type

We can create a map-like (dictionary) data type which contains the mapping between the values. For example, it can contain the mapping between employees and salaries or the students and the age, etc.

Consider the following example.

```
import numpy as np
d = np.dtype([('salary',np.float)])
print(d)
```

```
[('salary', '<f8')]
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: DeprecationWarning: `
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/re
```

```
import numpy as np
d=np.dtype([('salary',np.float)])
arr = np.array([(10000.12,),(20000.50,)],dtype=d)
print(arr['salary'])
```

```
[10000.12 20000.5 ]
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: DeprecationWarning: `
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/re
```

Numpy Array Creation

The ndarray object can be constructed by using the following routines.

Numpy.empty

As the name specifies, The empty routine is used to create an uninitialized array of specified shape and data type.

The syntax is given below.

numpy.empty(shape, dtype = float, order = 'C')

Shape: The desired shape of the specified array.

dtype: The data type of the array items. The default is the float.

Order: The default order is the c-style row-major order. It can be set to F for FORTRAN-style

```
import numpy as np
arr = np.empty((3,2), dtype = int)
print(arr)
```

```
[[ 1  2]
 [ 3 12]
 [19 29]]
```

NumPy.Zeros

This routine is used to create the numpy array with the specified shape where each numpy array item is initialized to 0.

The syntax is given below.

numpy.zeros(shape, dtype = float, order = 'C')

It accepts the following parameters.

Shape: The desired shape of the specified array.

dtype: The data type of the array items. The default is the float.

Order: The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

```
#Example
import numpy as np
arr = np.zeros((3,2), dtype = int)
print(arr)
```

```
[[0 0]
 [0 0]
 [0 0]]
```

NumPy.ones

This routine is used to create the numpy array with the specified shape where each numpy array item is initialized to 1.

The syntax to use this module is given below.

numpy.ones(shape, dtype = none, order = 'C')

It accepts the following parameters.

Shape: The desired shape of the specified array.

dtype: The data type of the array items.

Order: The default order is the c-style row-major order.

It can be set to F for FORTRAN-style column-major order.

```
import numpy as np
arr = np.ones((3,2), dtype = int)
print(arr)
```

```
[[1 1]
 [1 1]
 [1 1]]
```

Numpy array from existing data

NumPy provides us the way to create an array by using the existing data.

numpy.asarray This routine is used to create an array by using the existing data in the form of lists, or tuples. This routine is useful in the scenario where we need to convert a python sequence into the numpy array object.

The syntax to use the asarray() routine is given below.

numpy.asarray(sequence, dtype = None, order = None)

It accepts the following parameters.

sequence: It is the python sequence which is to be converted into the python array.

dtype: It is the data type of each item of the array.

order: It can be set to C or F. The default is C.

```
import numpy as np
l=[1,2,3,4,5,6,7]
a = np.asarray(l);
print(type(a))
print(a)
```

```
<class 'numpy.ndarray'>
[1 2 3 4 5 6 7]
```

```
#Example: creating a numpy array using Tuple
import numpy as np
l=(1,2,3,4,5,6,7)
a = np.asarray(l);
print(type(a))
print(a)
```

```
<class 'numpy.ndarray'>
[1 2 3 4 5 6 7]
```

```
#Example: creating a numpy array using more than one list
```

```
import numpy as np
l=[[1,2,3,4,5,6,7],[8,9]]
a = np.asarray(l);
print(type(a))
print(a)
```

```
<class 'numpy.ndarray'>
[list([1, 2, 3, 4, 5, 6, 7]) list([8, 9])]
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4: VisibleDeprecationWar
  after removing the cwd from sys.path.
```

numpy.frombuffer

This function is used to create an array by using the specified buffer. The syntax to use this buffer is given below.

numpy.frombuffer(buffer, dtype = float, count = -1, offset = 0)
It accepts the following parameters.

buffer: It represents an object that exposes a buffer interface. dtype: It represents the data type of the returned data type array. The default value is 0.

count: It represents the length of the returned ndarray. The default value is -1. offset: It represents the starting position to read from. The default value is 0.

Example

```
import numpy as np
l = b'hello world'
print(type(l))
a = np.frombuffer(l, dtype = "S1")
print(a)
print(type(a))
```

```
<class 'bytes'>
[b'h' b'e' b'l' b'l' b'o' b' ' b'w' b'o' b'r' b'l' b'd']
<class 'numpy.ndarray'>
```

numpy.fromiter

This routine is used to create a ndarray by using an iterable object. It returns a one-dimensional ndarray object.

The syntax is given below.

numpy.fromiter(iterable, dtype, count = - 1)

It accepts the following parameters.

Iterable: It represents an iterable object.

dtype: It represents the data type of the resultant array items.

count: It represents the number of items to read from the buffer in the array. Example

```
import numpy as np
list = [0,2,4,6]
it = iter(list)
x = np.fromiter(it, dtype = float)
print(x)
print(type(x))
```

```
[0. 2. 4. 6.]
<class 'numpy.ndarray'>
```

Numpy Arrays within the numerical range

This section of the tutorial illustrates how the numpy arrays can be created using some given specified range.

Numpy.arrange It creates an array by using the evenly spaced values over the given interval. The syntax to use the function is given below.

numpy.arrange(start, stop, step, dtype)
It accepts the following parameters.

start: The starting of an interval. The default is 0.

stop: represents the value at which the interval ends excluding this value.

step: The number by which the interval values change.

dtype: the data type of the numpy array items.

```
#Example
import numpy as np
arr = np.arange(0,10,2,float)
print(arr)
```

```
[0. 2. 4. 6. 8.]
```

```
#Example
import numpy as np
arr = np.arange(10,100,5,int)
print("The array over the given range is ",arr)
```

```
The array over the given range is  [10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 9
```

NumPy.linspace

It is similar to the arrange function. However, it doesn?t allow us to specify the step size in the syntax.

Instead of that, it only returns evenly separated values over a specified period. The system implicitly calculates the step size.

The syntax is given below.

numpy.linspace(start, stop, num, endpoint, retstep, dtype)

It accepts the following parameters.

start: It represents the starting value of the interval.

stop: It represents the stopping value of the interval.

num: The amount of evenly spaced samples over the interval to be generated. The default is 50.

endpoint: Its true value indicates that the stopping value is included in the interval.

rettstep: This has to be a boolean value. Represents the steps and samples between the consecutive numbers.

dtype: It represents the data type of the array items.

Example

```
#Example
import numpy as np
arr = np.linspace(10, 20, 5)
print("The array over the given range is ",arr)
```

```
The array over the given range is  [10.  12.5 15.  17.5 20. ]
```

```
#Example
import numpy as np
arr = np.linspace(10, 20, 7, endpoint = False)
print("The array over the given range is ",arr)
```

```
The array over the given range is  [10.         11.42857143 12.85714286 14.28571429 1
 18.57142857]
```

NumPy Broadcasting

In Mathematical operations, we may need to consider the arrays of different shapes. NumPy can perform such operations where the array of different shapes are involved.

For example, if we consider the matrix multiplication operation, if the shape of the two matrices is the same then this operation will be easily performed. However, we may also need to operate

if the shape is not similar.

Consider the following example to multiply two arrays.

Example

```
import numpy as np
a = np.array([1,2,3,4,5,6,7])
b = np.array([2,4,6,8,10,12,14])
c = a*b;
print(c)
```

```
[ 2  8 18 32 50 72 98]
```

However, in the above example, if we consider arrays of different shapes, we will get the errors as shown below.

Example

```
import numpy as np
a = np.array([1,2,3,4,5,6,7])
b = np.array([2,4,6,8,10,12,14,19])
c = a*b;
print(c)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-50-9e8a25e9ac52> in <module>
      2 a = np.array([1,2,3,4,5,6,7])
      3 b = np.array([2,4,6,8,10,12,14,19])
----> 4 c = a*b;
      5 print(c)

ValueError: operands could not be broadcast together with shapes (7,) (8,)
```

SEARCH STACK OVERFLOW

In the above example, we can see that the shapes of the two arrays are not similar and therefore they cannot be multiplied together. NumPy can perform such operation by using the concept of broadcasting.

In broadcasting, the smaller array is broadcast to the larger array to make their shapes compatible with each other.

Broadcasting Rules

Broadcasting is possible if the following cases are satisfied.

The smaller dimension array can be appended with '1' in its shape.

Size of each output dimension is the maximum of the input sizes in the dimension.

An input can be used in the calculation if its size in a particular dimension matches the output size or its value is exactly 1.

If the input size is 1, then the first data entry is used for the calculation along the dimension.

Broadcasting can be applied to the arrays if the following rules are satisfied.

All the input arrays have the same shape.

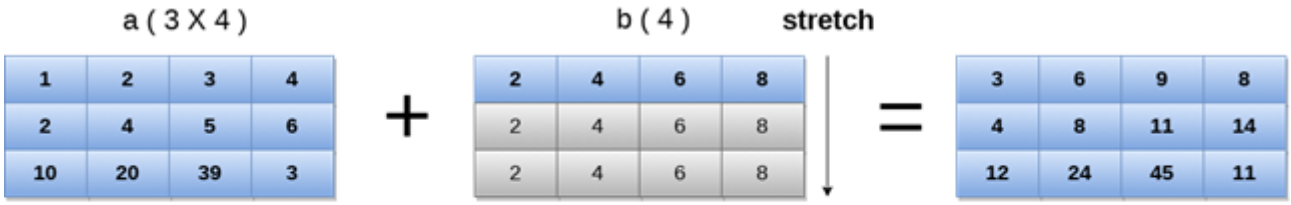
Arrays have the same number of dimensions, and the length of each dimension is either a common length or 1.

Array with the fewer dimension can be appended with '1' in its shape.

Let's see an example of broadcasting.

```
import numpy as np
a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
b = np.array([2,4,6,8])
print("\nprinting array a..")
print(a)
print("\nprinting array b..")
print(b)
print("\nAdding arrays a and b ..")
c = a + b;
print(c)
```

```
printing array a..
[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]

printing array b..
[2 4 6 8]

Adding arrays a and b ..
[[ 3  6  9 12]
 [ 4  8 11 14]
 [12 24 45 11]]
```



a = [[1, 2, 3, 4], [2, 4, 5, 6], [10, 20, 39, 3]]

b = [[2, 4, 6, 8]]

NumPy Array Iteration

NumPy provides an iterator object, i.e., nditer which can be used to iterate over the given array using python standard Iterator interface.

Consider the following example.

```
#Example
import numpy as np
a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
print("Printing array:")
print(a);
print("Iterating over the array:")
for x in np.nditer(a):
    print(x,end=' ')
```

```
Printing array:
[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]
Iterating over the array:
1 2 3 4 2 4 5 6 10 20 39 3
```

Order of the iteration doesn't follow any special ordering like row-major or column-order. However, it is intended to match the memory layout of the array.

Let's iterate over the transpose of the array given in the above example.

Example

```
import numpy as np
a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
print("Printing the array:")
print(a)
print("Printing the transpose of the array:")
at = a.T
print(at)

#this will be same as previous
for x in np.nditer(at):
    print(x,end=' ')

for x in np.nditer(a):
    print(x,end=' ')
```

```
 Printing the array:
 [[ 1  2  3  4]
  [ 2  4  5  6]
  [10 20 39  3]]
 Printing the transpose of the array:
 [[ 1  2 10]
  [ 2  4 20]
  [ 3  5 39]
  [ 4  6  3]]
 1 2 3 4 2 4 5 6 10 20 39 3 1 2 3 4 2 4 5 6 10 20 39 3
```

Order of Iteration

As we know, there are two ways of storing values into the numpy arrays:

F-style order C-style order Let's see an example of how the numpy Iterator treats the specific orders (F or C).

Example

```
import numpy as np

a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])

print("\nPrinting the array:\n")

print(a)

print("\nPrinting the transpose of the array:\n")
at = a.T

print(at)

print("\nIterating over the transposed array\n")

for x in np.nditer(at):
    print(x, end= ' ')
```

```
print("\nSorting the transposed array in C-style:\n")

c = at.copy(order = 'C')

print(c)

print("\nIterating over the C-style array:\n")
for x in np.nditer(c):
    print(x,end=' ')


d = at.copy(order = 'F')

print(d)
print("Iterating over the F-style array:\n")
for x in np.nditer(d):
    print(x,end=' ')
```

```
 Printing the array:

[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]

Printing the transpose of the array:

[[ 1  2 10]
 [ 2  4 20]
 [ 3  5 39]
 [ 4  6  3]]

Iterating over the transposed array

1 2 3 4 2 4 5 6 10 20 39 3
Sorting the transposed array in C-style:

[[ 1  2 10]
 [ 2  4 20]
 [ 3  5 39]
 [ 4  6  3]]

Iterating over the C-style array:

1 2 10 2 4 20 3 5 39 4 6 3 [[ 1  2 10]
 [ 2  4 20]
 [ 3  5 39]
 [ 4  6  3]]
Iterating over the F-style array:

1 2 3 4 2 4 5 6 10 20 39 3
```

NumPy String Functions

1	add()	It is used to concatenate the corresponding array elements (strings).
2	multiply()	It returns the multiple copies of the specified string, i.e., if a string 'hello' is multiplied by 3 then, a string 'hello hello' is returned.
3	center()	It returns the copy of the string where the original string is centered with the left and right padding filled with the specified number of fill characters.
4	capitalize()	It returns a copy of the original string in which the first letter of the original string is converted to the Upper Case.
5	title()	It returns the title cased version of the string, i.e., the first letter of each word of the string is converted into the upper case.
6	lower()	It returns a copy of the string in which all the letters are converted into the lower case.
7	upper()	It returns a copy of the string in which all the letters are converted into the upper case.
9	split()	It returns a list of words in the string.
9	splitlines()	It returns the list of lines in the string, breaking at line boundaries.
10	strip()	Returns a copy of the string with the leading and trailing white spaces removed.
11	join()	It returns a string which is the concatenation of all the strings specified in the given sequence.
12	replace()	It returns a copy of the string by replacing all occurrences of a particular substring with the specified one.
13	decode()	It is used to decode the specified string element-wise using the specified codec.
14	encode()	It is used to encode the decoded string element-wise.

Double-click (or enter) to edit

```
#numpy.char.encode() and decode() method example
import numpy as np
enstr = np.char.encode("welcome to javatpoint", 'cp500')
dstr =np.char.decode(enstr, 'cp500')
print(enstr)
print(dstr)
```

```
b'\xa6\x85\x93\x83\x96\x94\x85@\xa3\x96@\x91\x81\xa5\x81\xa3\x97\x96\x89\x95\xa3'
welcome to javatpoint
```

NumPy Mathematical Functions

Numpy contains a large number of mathematical functions which can be used to perform various mathematical operations. The mathematical functions include trigonometric functions, arithmetic functions, and functions for handling complex numbers. Let's discuss the mathematical functions.

Trigonometric functions Numpy contains the trigonometric functions which are used to calculate the sine, cosine, and tangent of the different angles in radian.

The sin, cos, and tan functions return the trigonometric ratio for the specified angles. Consider the following example.

Example

```
import numpy as np
arr = np.array([0, 30, 60, 90, 120, 150, 180])
print("\nThe sin value of the angles",end = " ")
print(np.sin(arr * np.pi/180))
print("\nThe cosine value of the angles",end = " ")
print(np.cos(arr * np.pi/180))
print("\nThe tangent value of the angles",end = " ")
print(np.tan(arr * np.pi/180))
```

```
The sin value of the angles [0.00000000e+00 5.00000000e-01 8.66025404e-01 1.00000000e
 8.66025404e-01 5.00000000e-01 1.22464680e-16]

The cosine value of the angles [ 1.00000000e+00  8.66025404e-01  5.00000000e-01  6.12
 -5.00000000e-01 -8.66025404e-01 -1.00000000e+00]

The tangent value of the angles [ 0.00000000e+00  5.77350269e-01  1.73205081e+00  1.6
 -1.73205081e+00 -5.77350269e-01 -1.22464680e-16]
```

The numpy.floor() function

This function is used to return the floor value of the input data which is the largest integer not greater than the input value. Consider the following example.

Example

```
import numpy as np
arr = np.array([12.202, 90.23120, 123.020, 23.202])
print(np.floor(arr))
```

```
[ 12.  90. 123.  23.]
```

Numpy statistical functions

Numpy provides various statistical functions which are used to perform some statistical data analysis. In this section of the tutorial, we will discuss the statistical functions provided by the numpy.

Finding the minimum and maximum elements from the array The numpy.amin() and numpy.amax() functions are used to find the minimum and maximum of the array elements along the specified axis respectively.

Consider the following example.

Example

```
import numpy as np
```

```
a = np.array([[2,10,20],[80,43,31],[22,43,10]])

print("The original array:\n")
print(a)


print("\nThe minimum element among the array:",np.amin(a))
print("The maximum element among the array:",np.amax(a))

print("\nThe minimum element among the rows of array",np.amin(a,0))
print("The maximum element among the rows of array",np.amax(a,0))

print("\nThe minimum element among the columns of array",np.amin(a,1))
print("The maximum element among the columns of array",np.amax(a,1))
```

```
The original array:

[[ 2 10 20]
 [80 43 31]
 [22 43 10]]

The minimum element among the array: 2
The maximum element among the array: 80

The minimum element among the rows of array [ 2 10 10]
The maximum element among the rows of array [80 43 31]

The minimum element among the columns of array [ 2 31 10]
The maximum element among the columns of array [20 80 43]
```

numpy.where() function

This function is used to return the indices of all the elements which satisfies a particular condition.

Consider the following example.

Example

```
import numpy as np

b = np.array([12, 90, 380, 12, 211])

print(np.where(b>12))

c = np.array([[20, 24],[21, 23]])

print(np.where(c>20))
```

```
(array([1, 2, 4]),)
(array([0, 1, 1]), array([1, 0, 1]))
```

The Difference Between Copy and View

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

COPY:

Example Make a copy, change the original array, and display both arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42
print(arr)
print(x)
```

```
[42  2  3  4  5]
[1 2 3 4 5]
```

The copy SHOULD NOT be affected by the changes made to the original array.

VIEW:

Example Make a view, change the original array, and display both arrays:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42

print(arr)
print(x)
```

```
[42  2  3  4  5]
[42  2  3  4  5]
```

The view SHOULD be affected by the changes made to the original array.

Random Numbers in NumPy

What is a Random Number? Random number does NOT mean a different number every time. Random means something that can not be predicted logically.

Generate Random Number

NumPy offers the random module to work with random numbers.

Example Generate a random integer from 0 to 100:

```
from numpy import random

x = random.randint(100)

print(x)
```

```
     7
```

Generate Random Float

The random module's rand() method returns a random float between 0 and 1.

Example Generate a random float from 0 to 1:

```
from numpy import random

x = random.rand()

print(x)
```

```
     0.4774790600751919
```

The randint() function can be used to simulate a lucky draw situation

```
import random
number = random.randint(1, 20)
```

```
attempts = 0  # count no of attempts to guess the number
guess = 0
while guess != number:
    guess = int(input('Guess a number: '))
    attempts += 1
    if guess == number:
        print( "Correct! You used", attempts, "attempts!")
        break
    elif guess < number:
        print ('Go higher!')
    else:
        print ('Go lower!')
```

```
 Guess a number: 19
 Go lower!
 Guess a number: 15
 Go higher!
 Guess a number: 16
 Go higher!
 Guess a number: 17
 Correct! You used 4 attempts!
```

```
#Example
import numpy as np
import numpy.matlib
print(numpy.matlib.empty((3,3)))
```

```
 [[9.88e-324 4.94e-323 9.88e-323]
  [3.95e-322 2.12e-322 1.53e-322]
  [1.09e-322 2.12e-322 4.94e-323]]
```

numpy.matlib.zeros() function

numpy.matlib.ones() function

numpy.matlib.eye() function

This function returns a matrix with the diagonal elements initialized to 1 and zero elsewhere.

numpy.matlib.identity() function

This function is used to return an identity matrix of the given size. An identity matrix is the one with diagonal elements initializes to 1 and all other elements to zero.

numpy.matlib.rand() function

This function is used to generate a matrix where all the entries are initialized with random values.

#What is Data Distribution? Data Distribution is a list of all possible values, and how often each value occurs.

Such lists are important when working with statistics and data science.

The random module offer methods that returns randomly generated data distributions.

Random Distribution

A random distribution is a set of random numbers that follow a certain probability density function.

Probability Density Function: A function that describes a continuous probability. i.e. probability of all values in an array.

We can generate random numbers based on defined probabilities using the choice() method of the random module.

The choice() method allows us to specify the probability for each value.

The probability is set by a number between 0 and 1, where 0 means that the value will never occur and 1 means that the value will always occur.

Example

Generate a 1-D array containing 100 values, where each value has to be 3, 5, 7 or 9.

The probability for the value to be 3 is set to be 0.1

The probability for the value to be 5 is set to be 0.3

The probability for the value to be 7 is set to be 0.6

The probability for the value to be 9 is set to be 0

```
from numpy import random

x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(100))

print(x)
```

```
[5 3 7 5 7 7 7 7 7 7 7 7 5 3 7 5 7 7 7 7 7 7 7 7 3 5 5 7 5 7 7 7 7 3 7 7 7
 7 7 5 7 7 7 7 7 5 7 7 3 5 5 3 7 7 7 7 7 7 7 7 7 7 3 5 5 3 5 5 7 3 3 5 3 7
 5 3 7 5 5 7 5 5 7 7 7 7 5 5 7 7 7 7 7 7 5 7 3 7 7 7]
```

The sum of all probability numbers should be 1.

Even if you run the example above 100 times, the value 9 will never occur.

You can return arrays of any shape and size by specifying the shape in the size parameter.

Example

Same example as above, but return a 2-D array with 3 rows, each containing 5 values.

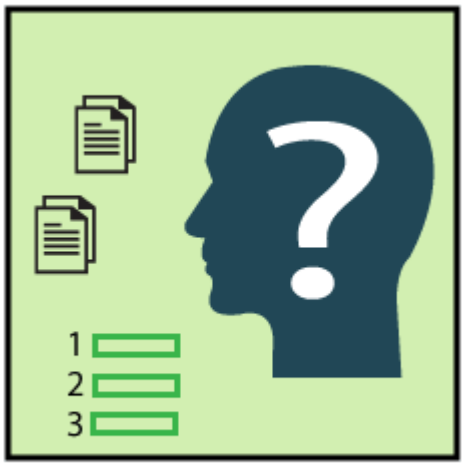
```
from numpy import random

x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(3, 5))

print(x)
```

```
[[7 7 3 5 5]
 [5 7 3 5 5]
 [7 7 7 5 5]]
```

Matplotlib (Python Plotting Library)

Human minds are more adaptive for the visual representation of data rather than textual data. We can easily understand things when they are visualized. It is better to represent the data through the graph where we can analyze the data more efficiently and make the specific decision according to data analysis. Before learning the matplotlib, we need to understand data visualization and why data visualization is important.

Data Visualization

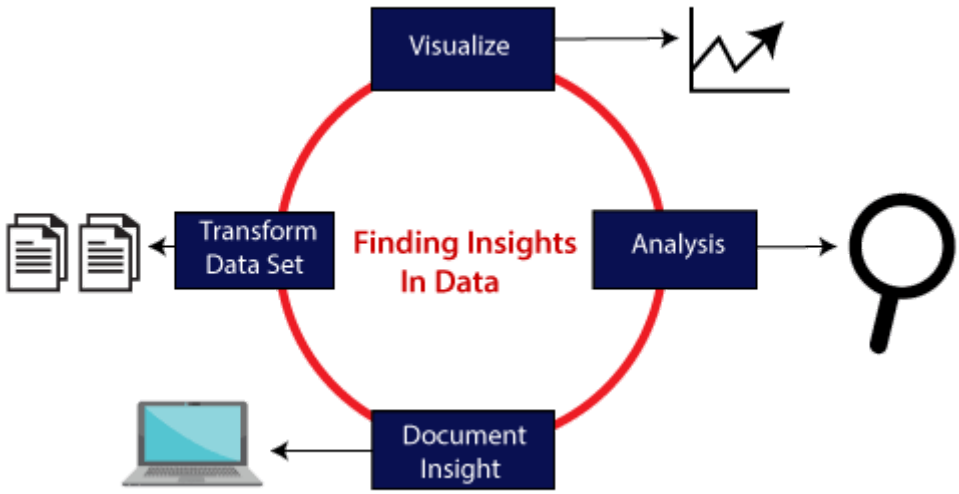


Graphics provides an excellent approach for exploring the data, which is essential for presenting results. Data visualization is a new term. It expresses the idea that involves more than just representing data in the graphical form (instead of using textual form).

This can be very helpful when discovering and getting to know a dataset and can help with classifying patterns, corrupt data, outliers, and much more. With a little domain knowledge, data visualizations can be used to express and demonstrate key relationships in plots and charts. The static does indeed focus on quantitative description and estimations of data. It provides an important set of tools for gaining a qualitative understanding.

There are five key plots that are used for data visualization.

There are five phases which are essential to make the decision for the organization:



Visualize: We analyze the raw data, which means it makes complex data more accessible, understandable, and more usable. Tabular data representation is used where the user will look up a specific measurement, while the chart of several types is used to show patterns or relationships in the data for one or more variables.

Analysis: Data analysis is defined as cleaning, inspecting, transforming, and modeling data to derive useful information. Whenever we make a decision for the business or in daily life, is by past experience. What will happen to choose a particular decision, it is nothing but analyzing our past. That may be affected in the future, so the proper analysis is necessary for better decisions for any business or organization.

Document Insight: Document insight is the process where the useful data or information is organized in the document in the standard format.

Transform Data Set: Standard data is used to make the decision more effectively.

Why need data visualization?

Data visualization can perform below tasks:

It identifies areas that need improvement and attention. It clarifies the factors. It helps to understand which product to place where. Predict sales volumes.

Matplotlib is a Python library which is defined as a multi-platform data visualization library built on Numpy array.

It can be used in python scripts, shell, web application, and other graphical user interface toolkit.

The General Concept of Matplotlib

A Matplotlib figure can be categorized into various parts as below:

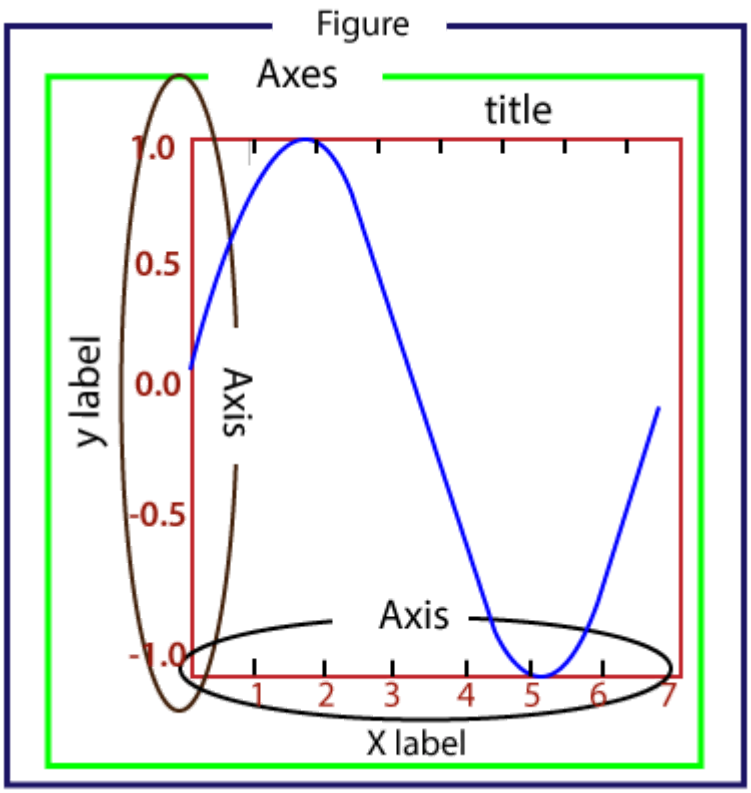


Figure: It is a whole figure which may hold one or more axes (plots). We can think of a Figure as a canvas that holds plots.

Axes: A Figure can contain several Axes. It consists of two or three (in the case of 3D) Axis objects. Each Axes is comprised of a title, an x-label, and a y-label.

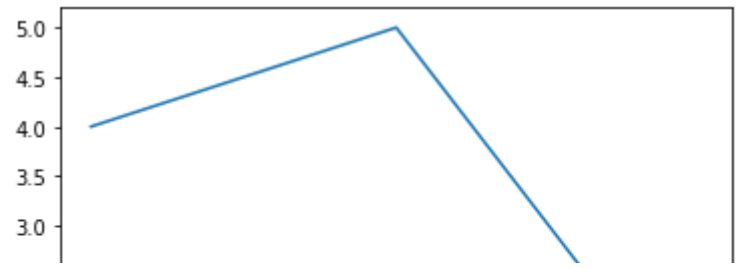
Axis: Axises are the number of line like objects and responsible for generating the graph limits.

Artist: An artist is the all which we see on the graph like Text objects, Line2D objects, and collection objects. Most Artists are tied to Axes.

Basic Example of plotting Graph

Here is the basic example of generating a simple graph; the program is following:

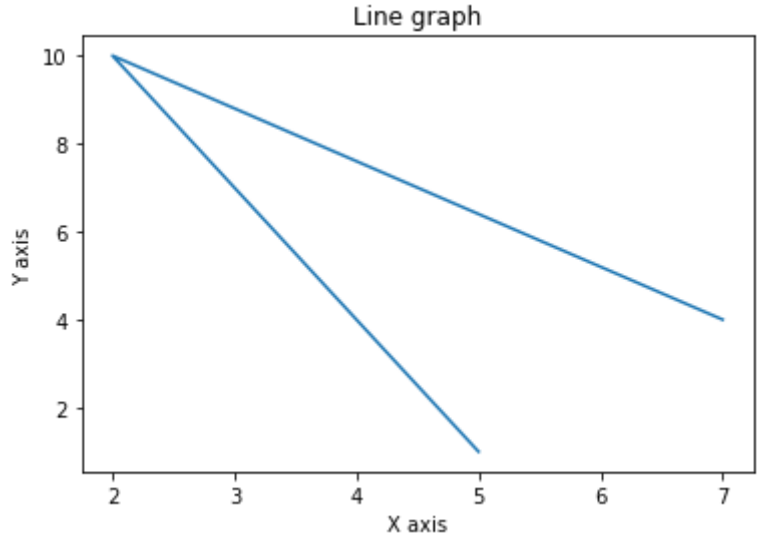
```
from matplotlib import pyplot as plt
#ploting our canvas
plt.plot([1,2,3],[4,5,1])
#display the graph
plt.show()
```



It takes only three lines to plot a simple graph using the Python matplotlib. We can add titles, labels to our chart which are created by Python matplotlib library to make it more meaningful. The example is the following:

```
1.00  1.25  1.50  1.75  2.00  2.25  2.50  2.75  3.00
```

```
from matplotlib import pyplot as plt

x = [5, 2, 7]
y = [1, 10, 4]
plt.plot(x, y)
plt.title('Line graph')
plt.ylabel('Y axis')
plt.xlabel('X axis')
plt.show()
```



The graph is more understandable from the previous graph.

Working with Pyplot

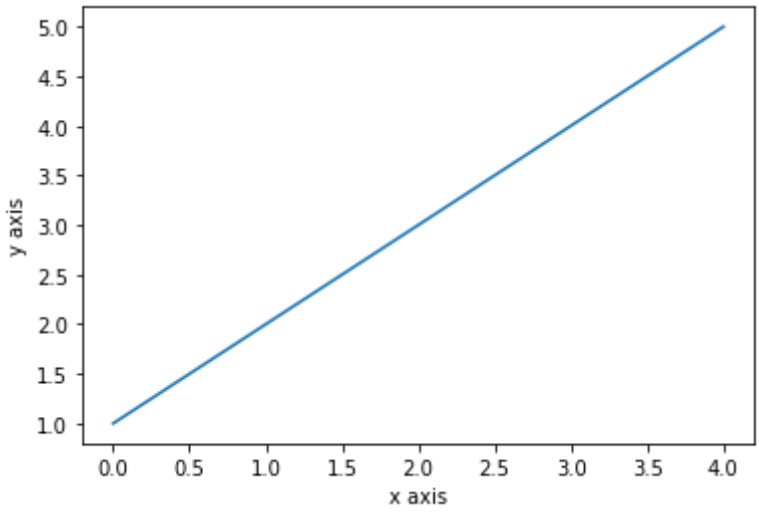
The matplotlib.pyplot is the collection command style functions that make matplotlib feel like working with MATLAB. The pyplot functions are used to make some changes to figure such as create a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot including labels, etc.

It is good to use when we want to plot something quickly without instantiating any figure or Axes.

While working with matplotlib.pyplot, some states are stored across function calls so that it keeps track of the things like current figure and plotting area, and these plotting functions are directed to the current axes.

The pyplot module provide the plot() function which is frequently use to plot a graph. Let's have a look on the simple example:

```
from matplotlib import pyplot as plt
plt.plot([1,2,3,4,5])
plt.ylabel("y axis")
plt.xlabel('x axis')
plt.show()
```

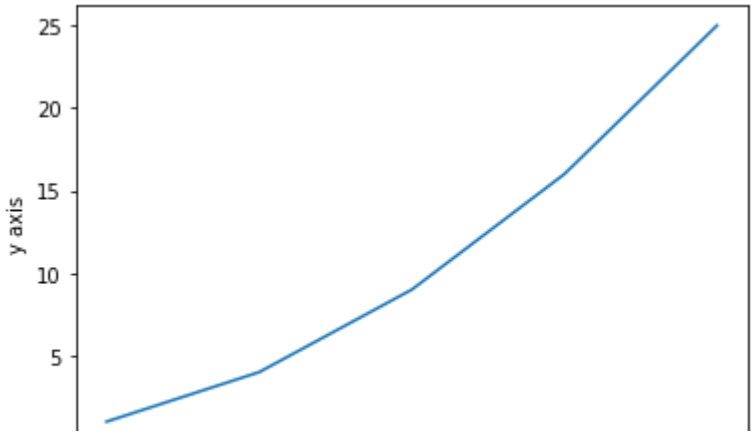


In the above program, it plots the graph x-axis ranges from 0-4 and the y-axis from 1-5. If we provide a single list to the plot(), matplotlib assumes it is a sequence of y values, and automatically generates the x values.

Since we know that python index starts at 0, the default x vector has the same length as y but starts at 0. Hence the x data are [0, 1, 2, 3, 4].

We can pass the arbitrary number of arguments to the plot(). For example, to plot x versus y, we can do this following way:

```
from matplotlib import pyplot as plt
plt.plot([1,2,3,4,5],[1,4,9,16,25])
plt.ylabel("y axis")
plt.xlabel('x axis')
plt.show()
```



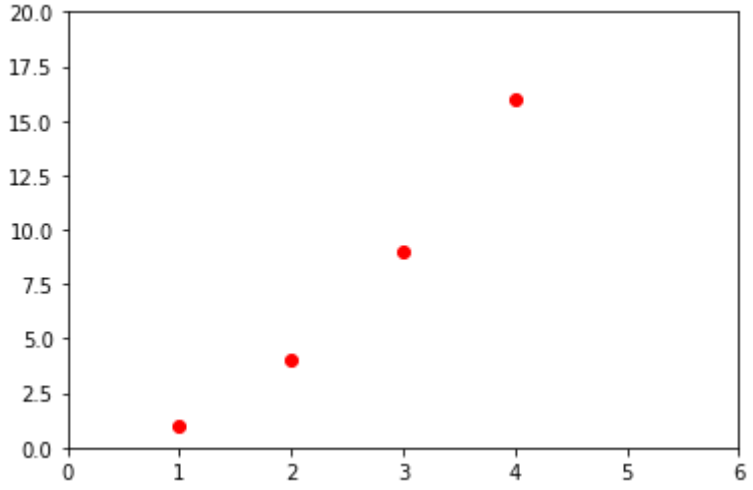
Formatting the style of the plot

There is an optional third argument, which is a format string that indicates the color and line type of the plot.

The default format string is 'b-'which is the solid blue as we can observe in the above plotted graph.

Let's consider the following example where we plot the graph with the red circle.

```
from matplotlib import pyplot as plt
plt.plot([1, 2, 3, 4,5], [1, 4, 9, 16,25], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```



Example format String

'b'	Using for the blue marker with default shape.
'ro'	Red circle
'-g'	Green solid line
'--'	A dashed line with the default color
'^k:'	Black triangle up markers connected by a dotted line

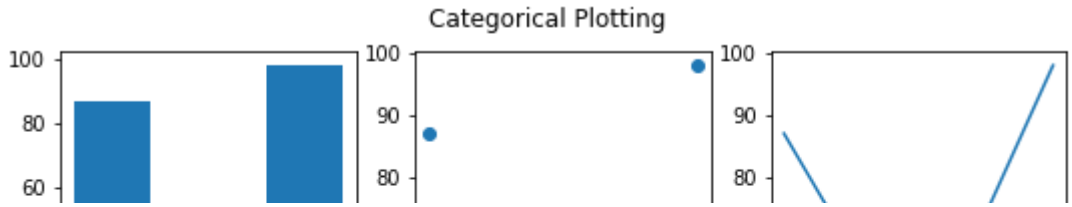
The matplotlib supports the following color abbreviation:

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

Plotting with categorical variables

Matplotlib allows us to pass categorical variables directly to many plotting functions: consider the following example

```
from matplotlib import pyplot
names = ['Abhishek', 'Himanshu', 'Devansh']
marks= [87,50,98]

plt.figure(figsize=(9,3))

plt.subplot(131)
plt.bar(names, marks)
plt.subplot(132)
plt.scatter(names, marks)
plt.subplot(133)
plt.plot(names, marks)
plt.suptitle('Categorical Plotting')
plt.show()
```



What is subplot()

The Matplotlib subplot() function is defined as to plot two or more plots in one figure. We can use this method to separate two graphs which plotted in the same axis Matplotlib supports all kinds of subplots, including 2x1 vertical, 2x1 horizontal, or a 2x2 grid.

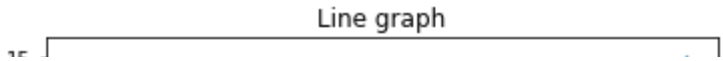
It accepts the three arguments: they are nrows, ncols, and index. It denote the number of rows, number of columns and the index.

The subplot() function can be called in the following way:

Creating different types of graph

1. Line graph The line graph is one of charts which shows information as a series of the line. The graph is plotted by the plot() function. The line graph is simple to plot; let's consider the following example:

```
from matplotlib import pyplot as plt

x = [4,8,9]
y = [10,12,15]

plt.plot(x,y)

plt.title("Line graph")
plt.ylabel('Y axis')
plt.xlabel('X axis')
plt.show()
```



We can customize the graph by importing the style module. The style module will be built into a matplotlib installation. It contains the various functions to make the plot more attractive. In the below program, we are using the style module:

```
from matplotlib import pyplot as plt
from matplotlib import style

style.use('ggplot')
x = [16, 8, 10]
y = [8, 16, 6]
x2 = [8, 15, 11]
y2 = [6, 15, 7]
plt.plot(x, y, 'r', label='line one', linewidth=5)
plt.plot(x2, y2, 'm', label='line two', linewidth=5)
plt.title('Epic Info')
fig = plt.figure()
plt.ylabel('Y axis')
plt.xlabel('X axis')
plt.legend()
plt.grid(True, color='k')
```

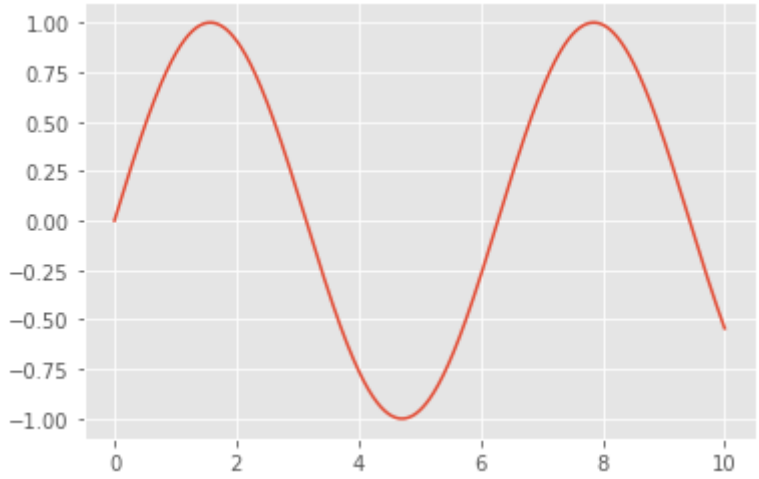
```
WARNING:matplotlib.legend:No handles with labels found to put in legend.
```

In Matplotlib, the figure (an instance of class plt.Figure) can be supposed of as a single container that consists of all the objects denoting axes, graphics, text, and labels.

Example-3

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

fig = plt.figure()
ax = plt.axes()

x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x))
```

```
[<matplotlib.lines.Line2D at 0x7fc405c051d0>]
```



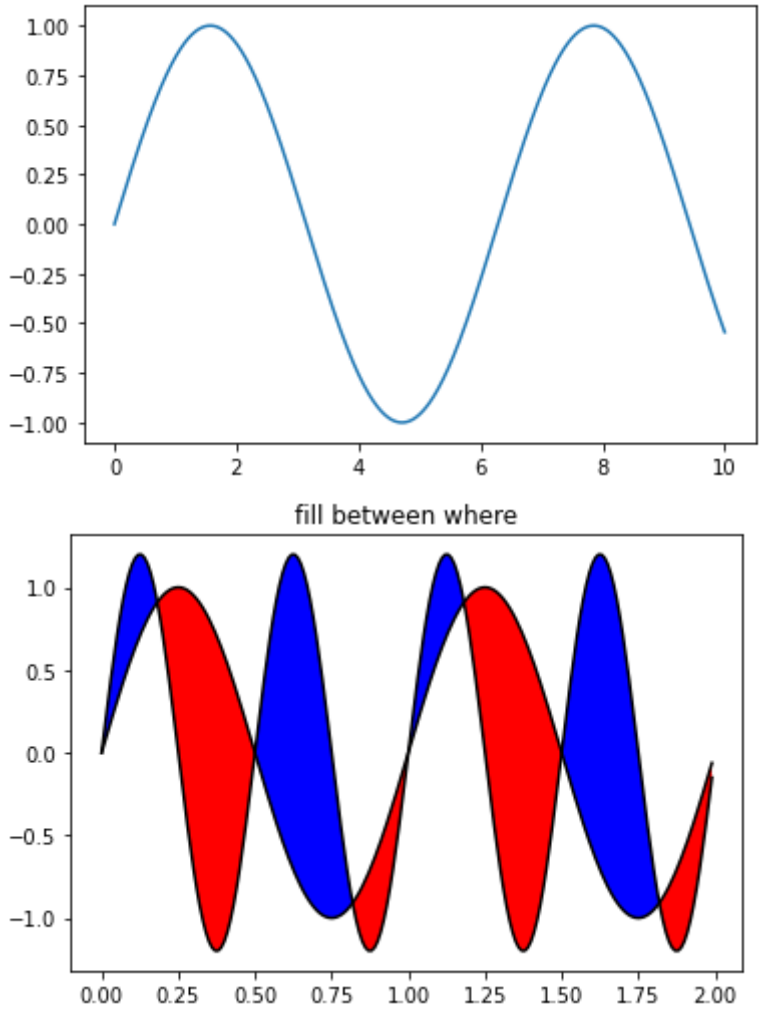
The matplotlib provides the fill_between() function which is used to fill area around the lines based on the user defined logic.

Example-4

```
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x))
```

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0.0, 2, 0.01)
y1 = np.sin(2 * np.pi * x)
y2 = 1.2 * np.sin(4 * np.pi * x)
fig, ax = plt.subplots(1, sharex=True)
ax.plot(x, y1, x, y2, color='black')
ax.fill_between(x, y1, y2, where=y2 >= y1, facecolor='blue', interpolate=True)
ax.fill_between(x, y1, y2, where=y2 <= y1, facecolor='red', interpolate=True)
ax.set_title('fill between where')
```

```
Text(0.5, 1.0, 'fill between where')
```

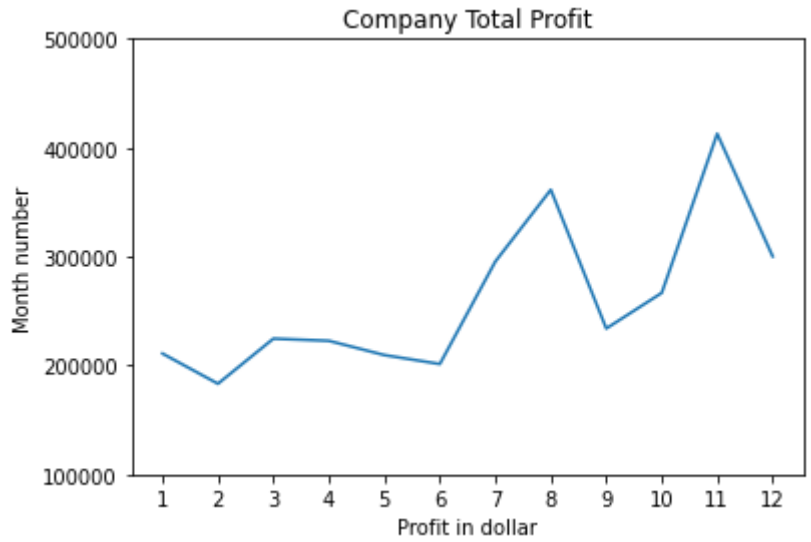



```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv("/content/drive/My Drive/Colab Notebooks/company_sales_data.csv")
profitList = df ['total_profit'].tolist()
monthList  = df ['month_number'].tolist()
plt.plot(monthList, profitList, label = 'Month-wise Profit data of last year')
plt.xlabel('Profit in dollar')
plt.ylabel('Month number')
```

```
plt.xticks(monthList)
plt.title('Company Total Profit')
plt.yticks([100000, 200000, 300000, 400000, 500000])
plt.show()
```



Take Dataset **company_sales_data** From Github https://github.com/makhan010385/DataSet

Exercise 1

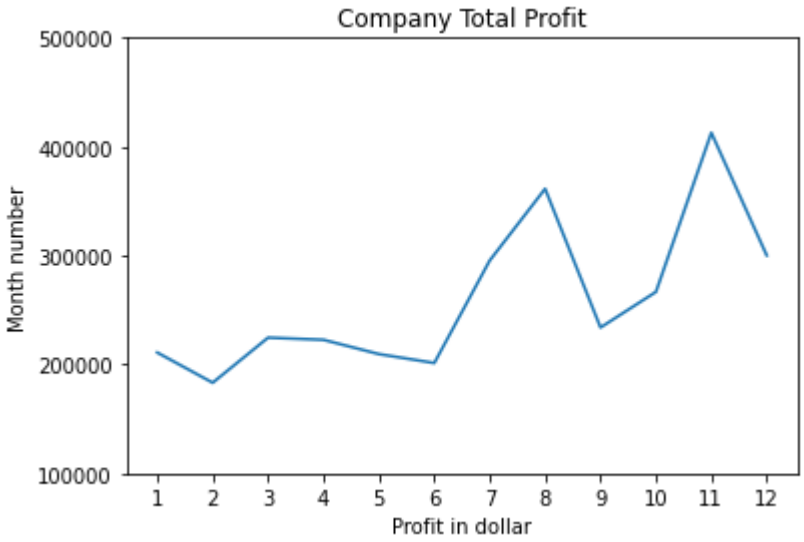
Read Total profit of all months and show it using a line plot

Total profit data provided for each month. Generated line plot must include the following properties: –

• X label name = Total Profit

• Y label name = Month Number

The line plot graph should look like this.



Exercise 2: Get total Unit of all months and show line plot with the following Style properties

Generated line plot must include following Style properties:

.Line Style dotted and Line-color should be red

.Show legend at the lower right location.

.X label name = Month

.Y label name = Sold units number

.Add a circle marker.

.Line marker color as read

.Line width should be 3