BansilalRamnathAgrawal Charitable Trust's

**VISHWAKARMA INSTITUTE OF INFORMATION TECHNOLOGY**

Department of Engineering &Applied Sciences

# F. Y. B. Tech.

Course Material (A Brief Reference Version for Students)

## Course: Python for Engineers

UNIT-III : **Numpy and Matplotlib**

| **Unit III – Numpy and Matplotlib** |
| --- |
| What is Numpy? How to install Numpy, Arrays, Array indexing, Array Vs Listing Data types, Array math, Broadcasting. Matplotlib -Plotting, subplots and images. |

# 1. What is NumPy?

- NumPy is a python library used for working with arrays.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.
- NumPy stands for Numerical Python.

# 2. Why Use NumPy ?

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster that traditional Python lists.
- The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.
- Data Science is a branch of computer science where we study how to store, use and Analyze data for deriving information from it.

# 3. Where is NumPy used?

Python NumPy arrays provide tools for integrating C, C++, etc. It is also useful in linear algebra, random number capability etc. NumPy array can also be used as an efficient multi-dimensional container for generic data.

## 4. What are NumPy Arrays?

NumPy is a Python package that stands for 'Numerical Python'. It is the core library for scientific computing, which contains a powerful n-dimensional array object.

## 5. Why is NumPy Faster Than Lists?

- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- This behavior is called locality of reference in computer science.
- This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

## 6. Which Language is NumPy written in?

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

## 7. What is Python NumPy Array?

Numpy array is a powerful N-dimensional array object which is in the form of rows and columns. We can initialize NumPy arrays from nested Python lists and access it elements.

## 8. Installation of NumPy

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

Install it using this command:

C:\Users\Your Name>pip install numpy

If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc.

## 9. Import NumPy

Once NumPy is installed, import it in your applications by adding the import keyword:

import numpy

Now Numpy is imported and ready to use.

**Example:**

```
import numpy

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

print(arr)
```

## 10. NumPy as np

NumPy is usually imported under the np alias.

alias: In Python alias are an alternate name for referring to the same thing.

Create an alias with the as keyword while importing:

```
import numpy as np
```

Now the NumPy package can be referred to as np instead of numpy.

**Example:**

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

## 11. Python Arrays

Python does not have built-in support for Arrays, but Python Lists can be used instead.

### 11.1 Arrays

Arrays are used to store multiple values in one single variable:

**Example:**

Create an array containing car names:

```
cars = ["Ford", "Volvo", "BMW"]
```

### 11.2 What is an Array?

An array is a special variable, which can hold more than one value at a time.

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

### 11.3 Access the Elements of an Array

You refer to an array element by referring to the index number.

Example:

Get the value of the first array item:

```
x = cars[0]
```

Example:

Modify the value of the first array item:

```
cars[0] = "Toyota"
```

### 11.4 The Length of an Array

Use the len() method to return the length of an array (the number of elements in an array).

Example:
Return the number of elements in the cars array:

```
x = len(cars)
```

### 11.5 Looping Array Elements

You can use the for in loop to loop through all the elements of an array.

Example

Print each item in the cars array:

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

### 11.6 Adding Array Elements

You can use the append() method to add an element to an array.

Example

Add one more element to the cars array:

cars.append("Honda")

### 11.7 Removing Array Elements

You can use the pop() method to remove an element from the array.

Example
Delete the second element of the cars array:
cars.pop(1)

You can also use the remove() method to remove an element from the array.

Example

Delete the element that has the value "Volvo":

## 12. NumPy Array Indexing

### 12.1 Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

**Example:**
Get the first element from the following array:

```
import numpy as np

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

print(arr[0])
```

**Example:**
Get the Second element from the following array:

```
import numpy as np
```

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

print(arr[1])
```

**Example:**
Get the third and fourth element from the following array and add them:

```
import numpy as np

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

print(arr[2] + arr[3])
```

## 12.2 Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

**Example:**

Access the 2nd element on 1st dim:

```
import numpy as np

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

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

**Example:**
Access the 5th element on 2nd dim:

```
import numpy as np

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

print('5th element on 2nd dim: ', arr[1, 4])
```

## 12.3 Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

**Example:**

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

```
import numpy as np

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

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

Explaination:

arr[0, 1, 2] prints the value 6.

And this is why:

The first number represents the first dimension, which contains two arrays:
[[1, 2, 3], [4, 5, 6]]
and:
[[7, 8, 9], [10, 11, 12]]
Since we selected 0, we are left with the first array:
[[1, 2, 3], [4, 5, 6]]

The second number represents the second dimension, which also contains two arrays:
[1, 2, 3]
and:
[4, 5, 6]
Since we selected 1, we are left with the second array:
[4, 5, 6]

The third number represents the third dimension, which contains three values:
4
5
6
Since we selected 2, we end up with the third value:
6

## 12.4 Negative Indexing

Use negative indexing to access an array from the end.

**Example:**
Print the last element from the 2nd dim:

```
import numpy as np

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

print('Last element from 2nd dim: ', arr[1, -1])
```

## 13. List vs Array

Let’s conclude the list vs array. Inserting parts in the middle of the list is exhausting since arrays are contiguous in memory. Lists allow straightforward insertion into lists. An array is a method of organizing data in a memory device. A list is a data structure that supports several operations. An array is a collection of homogenous parts, while a list consists of heterogeneous elements. Array memory is static and continuous. List memory is dynamic and random. Users don’t need to confine track of next memory with arrays. With lists, a user has to track of next location.

Lists and arrays are used in Python to store data(any data type- strings, integers etc), both can be indexed and iterated also. Difference between lists and arrays are the functions that you can perform on them like for example when you want to divide an array by 4, the result will be printed on request but in case of a list, python will throw an error message. Here's how it works:

Arrays need to be declared whereas lists do not need declaration because they are a part of Python's syntax. This is the reason lists are more often used than arrays. But in case you want to perform some arithmetic function to your list, one should go with arrays instead.

If you want to store a large amount of data, then you should consider arrays because they can store data very compactly and efficiently.

## 14. What’s the Difference?

Now that we know their definitions and features, we can talk about the differences between lists and arrays in Python:

- **Arrays need to be declared. Lists don’t**, since they are built into Python. In the examples above, you saw that lists are created by simply enclosing a sequence of elements into square brackets. Creating an array, on the other hand, requires a specific function from either the *array* module (i.e., *array.array()*) or *NumPy* package (i.e., *numpy.array()*). Because of this, lists are used more often than arrays.
- **Arrays can store data very compactly** and are more efficient for storing large amounts of data.
- **Arrays are great for numerical operations**; lists cannot directly handle math operations. For example, you can divide each element of an array by the same number with just one line of code. If you try the same with a list, you’ll get an error.

```
array =np.array([3, 6, 9, 12])
division =array/3
print(division)
print(type(division))
```

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

```
list=[3, 6, 9, 12]
```

```
division =list/3
---------------------------------------------------------------------------
TypeErrorTraceback (most recent call last)
 in ()
     1 list = [3, 6, 9, 12]


----> 2 division = list/3


TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

Of course, it's possible to do a mathematical operation with a list, but it's much less efficient:

- If you need to store a relatively short sequence of items and you don't plan to do any mathematical operations with it, a list is the preferred choice. This data structure will allow you to store an ordered, mutable, and indexed sequence of items without importing any additional modules or packages.
- If you have a very long sequence of items, consider using an array. This structure offers more efficient data storage.
- If you plan to do any numerical operations with your combination of items, use an array. Data analytics and data science rely heavily on (mostly NumPy) arrays.

## 15.Python Lists vs. Numpy Arrays - What is the difference?

We use python NumPy array instead of a list because of the below three reasons:

1. Less Memory
2. Fast
3. Convenient

The very first reason to choose python NumPy array is that it occupies less memory as compared to list. Then, it is pretty fast in terms of execution and at the same time, it is very convenient to work with NumPy. So these are the major advantages that Python NumPy array has over list. Don't worry, I am going to prove the above points one by one practically in PyCharm. Consider the below

Example:

```
import numpy as np

import time
import sys
S= range(1000)
print(sys.getsizeof(5)*len(S))

D= np.arange(1000)
print(D.size*D.itemsize)
```

O/P – 14000

4000

The above output shows that the memory allocated by list (denoted by S) is 14000 whereas the memory allocated by the NumPy array is just 4000. From this, you can conclude that there is a major difference between the two and this makes Python NumPy array as the preferred choice over list.

Example:

```
import numpy as np
import sys
S= range(5)   #INTEGER
# printing size of each element of the list
print("Size of each element of list in bytes: ",sys.getsizeof(S))
 # printing size of the whole list
 print("Size of the whole list in bytes: ",sys.getsizeof(S)*len(S))
  D= np.arange(5)
 # printing size of each element of the Numpy array
print("Size of each element of the Numpy array in bytes: ",D.itemsize)
  # printing size of the whole Numpy array
 print("Size of the whole Numpy array in bytes, D.itemsize * len(D))
```

**Output:**

```
Size of each element of list in bytes:  48
Size of the whole list in bytes:  240
Size of each element of the Numpy array in bytes:  8
Size of the whole Numpy array in bytes:  40
```

Example:

```
import numpy as np
import sys
S= ['1.1','2.1','3.1','4.1','5.1']
# printing size of each element of the list
print("Size of each element of list in bytes: ",sys.getsizeof(S))
 # printing size of the whole list
 print("Size of the whole list in bytes: ",sys.getsizeof(S)*len(S))

D= np.arange(['1.1','2.1','3.1','4.1','5.1'])
```

```
 # printing size of each element of the Numpy array
print("Size of each element of the Numpy array in bytes: ",D.itemsize)   # printing size of the whole Numpy array
print("Size of the whole Numpy array in bytes, D.itemsize * D.itemsize)
```

**Output:**

size of each element in bytes 112
size of List in bytes 560
size of each element in bytes 12
size of Nympy array in bytes 60

**np.size() will give us how many elements are present in total. For a (3,4) array, it will be 12.**

Next, let's talk how python NumPy array is faster and more convenient when compared to list.

```
import time
import sys

SIZE = 1000000

L1= range(SIZE)
L2= range(SIZE)
A1= np.arange(SIZE)
A2=np.arange(SIZE)

start= time.time()
result=[(x,y) for x,y in zip(L1,L2)]
print((time.time()-start)*1000)

start=time.time()
result= A1+A2
print((time.time()-start)*1000)
```

O/P – 380.9998035430908

49.99995231628418

n the above code, we have defined two lists and two numpy arrays. Then, we have compared the time taken in order to find the sum of lists and sum of numpy arrays both. If you see the output of the above program, there is a significant change in the two values. List took 380ms whereas the numpy array took almost 49ms. Hence, numpy array is faster than list. Now, if you noticed we had run a 'for' loop for a list which returns the concatenation of both the lists whereas for numpy arrays, we have just added the two array by simply printing A1+A2. That's why working with numpy is much easier and convenient when compared to the lists.

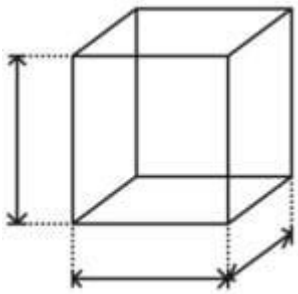
Therefore, the above examples proves the point as to why you should go for python numpy array rather than a list!

Moving forward in python numpy tutorial, let's focus on some of its operations.

You may go through this recording of Python NumPy tutorial where our instructor has explained the topics in a detailed manner with examples that will help you to understand this concept better.

# 16. Python NumPy Operations

## 16.1 ndim:



You can find the dimension of the array, whether it is a two-dimensional array or a single dimensional array. So, let us see this practically how we can find the dimensions. In the below code, with the help of 'ndim' function, I can find whether the array is of single dimension or multi dimension.

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

Output – 2

Since the output is 2, it is a two-dimensional array (multi dimension).

## 16.2 itemsize:


You can calculate the byte size of each element. In the below code, I have defined a single dimensional array and with the help of 'itemsize' function, we can find the size of each element.

```
import numpy as np
a = np.array([(1,2,3)])
print(a.itemsize)
```

Output – 8

So every element occupies 4 byte in the above numpy array.

### 16.3 dtype:

You can find the data type of the elements that are stored in an array. So, if you want to know the data type of a particular element, you can use 'dtype' function which will print the datatype along with the size. In the below code, I have defined an array where I have used the same function.

```
1  import numpy as np
2  a = np.array([(1,2,3)])
3  print(a.dtype)
```

Output – int32

As you can see, the data type of the array is integer 32 bits. Similarly, you can find the size and shape of the array using 'size' and 'shape' function respectively.

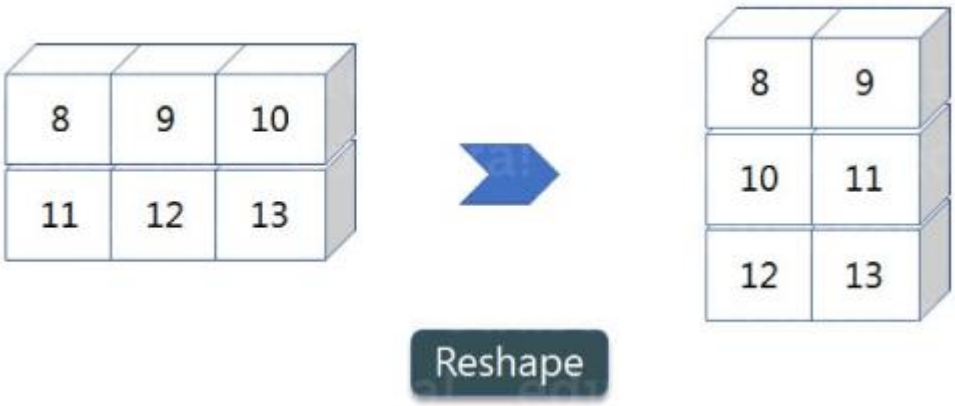
```
1  import numpy as np
2  a = np.array([(1,2,3,4,5,6)])
3  print(a.size)
4  print(a.shape)
```

Output – 6 (1,6)

Next, let us move forward and see what are the other operations that you can perform with python numpy module. We can also perform reshape as well as slicing operation using python numpy operation. But, what exactly is reshape and slicing? So let me explain this one by one in this python numpy tutorial.

### 16.4 reshape:

Reshape is when you change the number of rows and columns which gives a new view to an object. Now, let us take an example to reshape the below array:



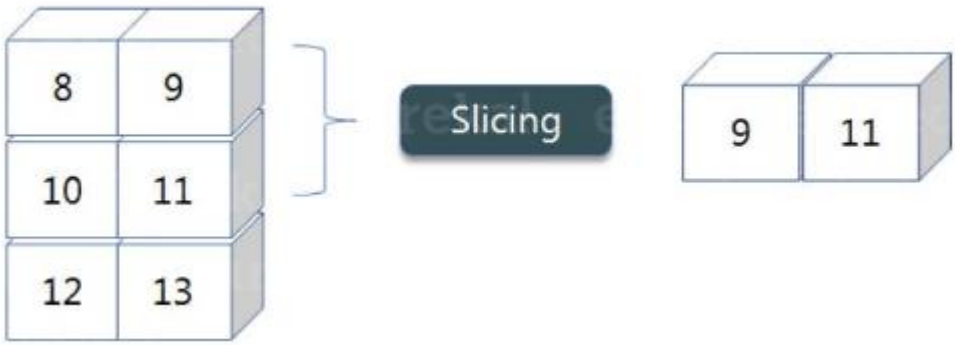

As you can see in the above image, we have 3 columns and 2 rows which has converted into 2 columns and 3 rows. Let me show you practically how it's done.

```
import numpy as np
a = np.array([(8,9,10),(11,12,13)])
print(a)
a=a.reshape(3,2)
print(a)
```

Output – [[ 8 9 10] [11 12 13]] [[ 8 9] [10 11] [12 13]]

### 16.5 slicing:

As you can see the 'reshape' function has showed its magic. Now, let's take another operation i.e Slicing. Slicing is basically extracting particular set of elements from an array. This slicing operation is pretty much similar to the one which is there in the list as well. Consider the following example:




Before getting into the above example, let's see a simple one. We have an array and we need a particular element (say 3) out of a given array. Let's consider the below example:

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

Output – 3

Here, the array(1,2,3,4) is your index 0 and (3,4,5,6) is index 1 of the python numpy array. Therefore, we have printed the second element from the zeroth index.
Taking one step forward, let's say we need the 2nd element from the zeroth and first index of the array. Let's see how you can perform this operation:

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

Output – [3 5]

Here colon represents all the rows, including zero. Now to get the 2nd element, we'll call index 2 from both of the rows which gives us the value 3 and 5 respectively.

Next, just to remove the confusion, let's say we have one more row and we don't want to get its 2nd element printed just as the image above. What we can do in such case?
Consider the below code:

```
import numpy as np
a=np.array([(8,9),(10,11),(12,13)])
print(a[0:2,1])
```

Output – [9 11]

As you can see in the above code, only 9 and 11 gets printed. Now when I have written 0:2, this does not include the second index of the third row of an array. Therefore, only 9 and 11 gets printed else you will get all the elements i.e [9 11 13].

### 16.6 linspace

This is another operation in python numpy which returns evenly spaced numbers over a specified interval. Consider the below example:

```
import numpy as np
a=np.linspace(1,3,10)
print(a)
```

Output – [ 1. 1.22222222 1.44444444 1.66666667 1.88888889 2.11111111 2.33333333 2.55555556 2.77777778 3. ]

As you can see in the result, it has printed 10 values between 1 to 3.

### 16.7 max/ min

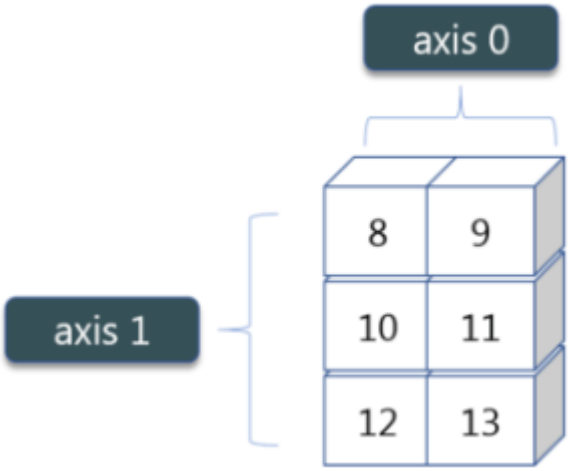
Next, we have some more operations in numpy such as to find the minimum, maximum as well the sum of the numpy array. Let's go ahead in python numpy tutorial and execute it practically.

```
import numpy as np

a= np.array([1,2,3])
print(a.min())
print(a.max())
print(a.sum())
```

Output – 1 3 6

You must be finding these pretty basic, but with the help of this knowledge you can perform a lot bigger tasks as well. Now, lets understand the concept of **axis** in python numpy.




As you can see in the figure, we have a numpy array 2*3. Here the rows are called as axis 1 and the columns are called as axis 0. Now you must be wondering what is the use of these axis?

Suppose you want to calculate the sum of all the columns, then you can make use of axis. Let me show you practically, how you can implement axis in your PyCharm:

```
a= np.array([(1,2,3),(3,4,5)])
print(a.sum(axis=0))
```

Output – [4 6 8]

Therefore, the sum of all the columns are added where 1+3=4, 2+4=6 and 3+5=8. Similarly, if you replace the axis by 1, then it will print [6 12] where all the rows get added.

## 16.8 Square Root & Standard Deviation

There are various mathematical functions that can be performed using python numpy. You can find the square root, standard deviation of the array. So, let's implement these operations:

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

Output – [[ 1. 1.41421356 1.73205081]
[ 1.73205081 2. 2.23606798]]
1.29099444874

As you can see the output above, the square root of all the elements are printed. Also, the standard deviation is printed for the above array i.e how much each element varies from the mean value of the python numpy array.

## 16.9 Addition Operation

You can perform more operations on numpy array i.e addition, subtraction,multiplication and division of the two matrices. Let me go ahead in python numpy tutorial, and show it to you practically:

```
import numpy as np
x= np.array([(1,2,3),(3,4,5)])
y= np.array([(1,2,3),(3,4,5)])
print(x+y)
```

Output – [[ 2 4 6] [ 6 8 10]]

This is extremely simple! Right? Similarly, we can perform other operations such as subtraction, multiplication and division. Consider the below example:

```
import numpy as np
x= np.array([(1,2,3),(3,4,5)])
y= np.array([(1,2,3),(3,4,5)])
print(x-y)
print(x*y)
print(x/y)
```

Output – [[0 0 0] [0 0 0]]
[[ 1 4 9] [ 9 16 25]]
[[ 1. 1. 1.] [ 1. 1. 1.]]

## 16.10 Vertical & Horizontal Stacking

Next, if you want to concatenate two arrays and not just add them, you can perform it using two ways – *vertical stacking* and *horizontal stacking*. Let me show it one by one in this python numpy tutorial.

```
import numpy as np
x= np.array([(1,2,3),(3,4,5)])
y= np.array([(1,2,3),(3,4,5)])
print(np.vstack((x,y)))
print(np.hstack((x,y)))
```

Output – [[1 2 3] [3 4 5] [1 2 3] [3 4 5]]
[[1 2 3 1 2 3] [3 4 5 3 4 5]]

### 16.11 ravel

There is one more operation where you can convert one numpy array into a single column i.e *ravel*. Let me show how it is implemented practically:

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

Output – [ 1 2 3 3 4 5]

## 17. Python Special Functions

### 17.1 sin() function

```
import numpy as np
import math

in_array =[0, math.pi /2, np.pi /3, np.pi]
print("Input array : \n", in_array)

Sin_Values =np.sin(in_array)
print("\nSine values : \n", Sin_Values)
```

Run on IDE

**Output :**

```
Input array :
 [0, 1.5707963267948966, 1.0471975511965976, 3.141592653589793]


Sine values :
[  0.00000000e+00   1.00000000e+00   8.66025404e-01   1.22464680e-16]
```

### 12.2. numpy.cos(x[, out]) = ufunc ‘cos’) :

This mathematical function helps user to calculate trignmetric cosine for all x(being the array elements).

**cos() function**

```
import numpy as np
import math

in_array =[0, math.pi /2, np.pi /3, np.pi]
print("Input array : \n", in_array)

cos_Values =np.cos(in_array)
print("\nCosine values : \n", cos_Values)
```

Run on IDE

**Output :**

```
Input array :
 [0, 1.5707963267948966, 1.0471975511965976, 3.141592653589793]


Cosine values :
[  1.00000000e+00   6.12323400e-17   5.00000000e-01  -1.00000000e+00]
```

| FUNCTION | DESCRIPTION |
|---|---|
| **tan()** | Compute tangent element-wise. |
| **arcsin()** | Inverse sine, element-wise. |
| **arccos()** | Trigonometric inverse cosine, element-wise. |
| **arctan()** | Trigonometric inverse tangent, element-wise. |
| **arctan2()** | Element-wise arc tangent of x1/x2 choosing the quadrant correctly. |
| **degrees()** | Convert angles from radians to degrees. |
| **rad2deg()** | Convert angles from radians to degrees. |
| **deg2rad** | Convert angles from degrees to radians. |

| | |
|---|---|
| **radians()** | Convert angles from degrees to radians. |
| **hypot()** | Given the “legs” of a right triangle, return its hypotenuse. |

### 12.3.Functions for Rounding

**numpy.around(arr, decimals = 0, out = None) :**

This mathematical function helps user to evenly round array elements to the given number of decimals.

**around() function**

```
import numpy as np

in_array =[.5, 1.5, 2.5, 3.5, 4.5, 10.1]
print("Input array : \n", in_array)

round_off_values =np.around(in_array)
print("\nRoundedvalues : \n", round_off_values)

in_array =[.53, 1.54, .71]
print("\nInputarray : \n", in_array)

round_off_values =np.around(in_array)
print("\nRoundedvalues : \n", round_off_values)

in_array =[.5538, 1.33354, .71445]
print("\nInputarray : \n", in_array)

round_off_values =np.around(in_array, decimals =3)
print("\nRounded values : \n", round_off_values)
```

Run on IDE

**Output :**

```
Input array :
 [0.5, 1.5, 2.5, 3.5, 4.5, 10.1]


Rounded values :
[ 0.  2.  2.  4.  4. 10.]


Input array :
 [0.53, 1.54, 0.71]
```

```
Rounded values :
[ 1.  2.  1.]

Input array :
[0.5538, 1.33354, 0.71445]

Rounded values :
[ 0.554  1.334  0.714]
```

```
Input array :
[0.5538, 1.33354, 0.71445]

Rounded values :
[ 0.554  1.334  0.714]
```

### 12.4.Exponents and logarithms Functions

**numpy.exp(array, out = None, where = True, casting = 'same_kind', order = 'K', dtype = None) :**

This mathematical function helps user to calculate exponential of all the elements in the input array.

**exp() function**

```
importnumpy as np

in_array =[1, 3, 5]
print("Input array : ", in_array)

out_array =np.exp(in_array)
print("Output array : ", out_array)
```

Run on IDE

**Output :**

```
Input array :  [1, 3, 5]
Output array :  [   2.71828183   20.08553692  148.4131591 ]
```

**12.4.numpy.log(x[, out] = ufunc 'log1p') :**

This mathematical function helps user to calculate Natural logarithm **of x** where x belongs to all the input array elements.
Natural logarithm log is the inverse of the exp()**,** so that log(exp(x)) = x. The natural logarithm is log in base e.

**log() function**

```
importnumpy as np

in_array =[1, 3, 5, 2**8]
print("Input array : ", in_array)

out_array =np.log(in_array)
print("Output array : ", out_array)


print("\nnp.log(4**4) : ", np.log(4**4))
print("np.log(2**8) : ", np.log(2**8))
```

Run on IDE

**Output :**

```
Input array :  [1, 3, 5, 256]

Output array :  [ 0.          1.09861229  1.60943791  5.54517744]


np.log(4**4) :  5.54517744448

np.log(2**8) :  5.54517744448
```

**12.5.Arithmetic Functions**

**numpy.reciprocal(x, /, out=None, *, where=True) :**

This mathematical function is used to calculate reciprocal of all the elements in the input array.
**Note:** For integer arguments with absolute value larger than 1, the result is always zero because of the way Python handles integer division. For integer zero the result is an overflow.

**Python3 code demonstrate reciprocal() function**

```
# importing numpy
importnumpy as np

in_num =2.0
print("Input  number : ", in_num)

out_num =np.reciprocal(in_num)
print("Output number : ", out_num)
```

Run on IDE

**Output :**

```
Input  number :  2.0
Output number :  0.5
```

**numpy.divide(arr1, arr2, out = None, where = True, casting = 'same_kind', order = 'K', dtype = None) :**

Array element from first array is divided by elements from second element (all happens element-wise). Both arr1 and arr2 must have same shape and element in arr2 must not be zero; otherwise it will raise an error.

**divide() function**

```
importnumpy as np

# input_array
arr1 =[2, 27, 2, 21, 23]
arr2 =[2, 3, 4, 5, 6]
print("arr1        : ", arr1)
print("arr2        : ", arr2)

# output_array
out =np.divide(arr1, arr2)
print("\nOutput array : \n", out)
```

Run on IDE

**Output :**

```
arr1        :  [2, 27, 2, 21, 23]
arr2        :  [2, 3, 4, 5, 6]


Output array :
 [ 1.         9.         0.5        4.2        3.83333333]
```

| FUNCTION | DESCRIPTION |
|---|---|
| **add()** | Add arguments element-wise. |
| **positive()** | Numerical positive, element-wise. |

| | |
|---|---|
| **negative()** | Numerical negative, element-wise. |
| **multiply()** | Multiply arguments element-wise. |
| **power()** | First array elements raised to powers from second array, element-wise. |
| **subtract()** | Subtract arguments, element-wise. |
| **true_divide()** | Returns a true division of the inputs, element-wise. |
| **floor_divide()** | Return the largest integer smaller or equal to the division of the inputs. |
| **float_power()** | First array elements raised to powers from second array, element-wise. |
| **mod()** | Return the element-wise remainder of division. |
| **remainder()** | Return element-wise remainder of division. |
| **divmod()** | Return element-wise quotient and remainder simultaneously. |

| FUNCTION | DESCRIPTION |
|---|---|
| **convolve()** | Returns the discrete, linear convolution of two one-dimensional sequences. |
| **sqrt()** | Return the non-negative square-root of an array, element-wise. |

| | |
|---|---|
| **square()** | Return the element-wise square of the input. |
| **absolute()** | Calculate the absolute value element-wise. |
| **fabs()** | Compute the absolute values element-wise. |
| **sign()** | Returns an element-wise indication of the sign of a number. |
| **interp()** | One-dimensional linear interpolation. |
| **maximum()** | Element-wise maximum of array elements. |
| **minimum()** | Element-wise minimum of array elements. |
| **real_if_close()** | If complex input returns a real array if complex parts are close to zero. |
| **nan_to_num()** | Replace NaN with zero and infinity with large finite numbers. |
| **heaviside()** | Compute the Heaviside step function. |

## 13. Python | Broadcasting with NumPy Arrays

The term ***broadcasting*** refers to how numpy treats arrays with different Dimension during arithmetic operations which lead to certain constraints, the smaller array is broadcast across the larger array so that they have compatible shapes.

Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python as we know that Numpy implemented in C. It does this without making needless copies of data and which leads to efficient algorithm implementations. There are cases where broadcasting is a bad idea because it leads to inefficient use of memory that slow down the computation.

**Example :**

```
import numpy as np

A =np.array([5, 7, 3, 1])
B =np.array([90, 50, 0, 30])

# array are compatible because of same Dimension
```

```
c =a *b
print(c)
```

**Let's see a naive way of producing this computation with Numpy:**

```
macros =array([[0.8, 2.9, 3.9], [52.4, 23.6, 36.5], [55.2, 31.7, 23.9], [14.4, 11, 4.9]])

# Create a new array filled with zeros,
# of the same shape as macros.

result =zeros_like(macros)

cal_per_macro =array([3, 3, 8])

# Now multiply each row of macros by
# cal_per_macro. In Numpy, `*` is
# element-wise multiplication between two arrays.

for in range(macros.shape[0]):
    result[i, :] =macros[i, :] *cal_per_macro

result
```

**output:**

```
array([[   2.4,    8.7,   31.2 ],
[  157.2,   70.8,  292 ],
     [   165.6,  95.1,   191.2],
     [   43.2,   33,    39.2]])
```

**13.1. Broadcasting Rules:**

**Broadcasting** stretches the value or the arrays to the required shape and then performs an arithmetic operation. but NumPy doesn't perform this with stretching and replicating. It directly performs this. To perform broadcasting we must follow some rules and those rules are known as Broadcasting rules.

Broadcasting two arrays together follow these rules:

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
- Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

**Rule1(Example):**

**Adding a two-dimensional array to a one-dimensional array:**

```
import numpy as np
a=np.ones((2,3))
print(a)
b=np.arange(3)
print(b)
C=a+b
print(C)
```

Output:
**[[1. 1. 1.] [1. 1. 1.]]**
**[0 1 2]**
**[[1. 2. 3.] [1. 2. 3.]]**

**Rule 2(Example):**

```
import numpy as np
a=np.arange(3).reshape((3,1))
print(a)
b=np.arange(3)
print(b)
C=a+b
print(C)
```

Output:
**[[0]**
**[1]**
**[2]]**

**[0 1 2]**

**[[0 1 2]**
**[1 2 3]**
**[2 3 4]]**

**Rule 3(Example):**

**Which the two arrays are not compatible**

```
import numpy as np
a=np.ones((3,2))
print(a)
b=np.arange(3)
print(b)
C=a+b
print(C)
```

Output:

**[[1. 1.]**
**[1. 1.]**
**[1. 1.]]**

**[0 1 2]**

**ValueError: operands could not be broadcast together with shapes (3,2) (3,)**

**Example :** Single Dimension array

```
import numpy as np
a =np.array([17, 11, 19]) # 1x3 Dimension array
print(a)
b =3
print(b)

# Broadcasting happened beacuse of
# miss match in array Dimension.
c =a +b
print(c)
```

**Output:**

```
[17 11 19]
3
[20 14 22]
```

**Example :** Two Dimensional Array

```
import numpy as np
A =np.array([[11, 22, 33], [10, 20, 30]])
print(A)

b =4
print(b)

C =A +b
print(C)
```

**Output:**

```
[[11 22 33]
 [10 20 30]]
 4
[[15 26 37]
 [14 24 34]]
```

**Example :**

```
Import numpy as np

v =np.array([12, 24, 36])

w =np.array([45, 55])
```

```
# To compute an outer product we first
# reshape v to a column vector of shape 3x1
# then broadcast it against w to yield an output
# of shape 3x2 which is the outer product of v and w
print(np.reshape(v, (3, 1)) *w)

X =np.array([[12, 22, 33], [45, 55, 66]])

# x has shape  2x3 and v has shape (3, )
# so they broadcast to 2x3,
print(X +v)

# Add a vector to each column of a matrix X has
# shape 2x3 and w has shape (2, ) If we transpose X
# then it has shape 3x2 and can be broadcast against w
# to yield a result of shape 3x2.

# Transposing this yields the final result
# of shape  2x3 which is the matrix.
print((x.T +w).T)

# Another solution is to reshape w to be a column
# vector of shape 2X1 we can then broadcast it
# directly against X to produce the same output.
print(x +np.reshape(w, (2, 1)))

# Multiply a matrix by a constant, X has shape  2x3.
# Numpy treats scalars as arrays of shape();
# these can be broadcast together to shape 2x3.
print(x *2)
```

**Output:**

```
[[ 4  5]
 [ 8 10]
 [12 15]]

[[2 4 6]
 [5 7 9]]

[[ 5  6  7]
 [ 9 10 11]]

[[ 5  6  7]
 [ 9 10 11]]
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

# 14. Matplotlib

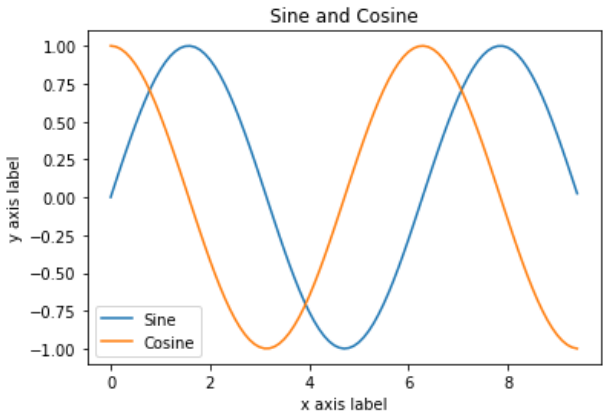
Matplotlib is a library in Python and it is numerical – mathematical extension for NumPy library. Pyplot is a state-based interface to **a** Matplotlib module which provides a MATLAB-like interface.

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

# Computes x and y coordinates for
# points on sine and cosine curves
x =np.arange(0, 3*np.pi, 0.1)
y_sin =np.sin(x)
y_cos =np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

**Output:**



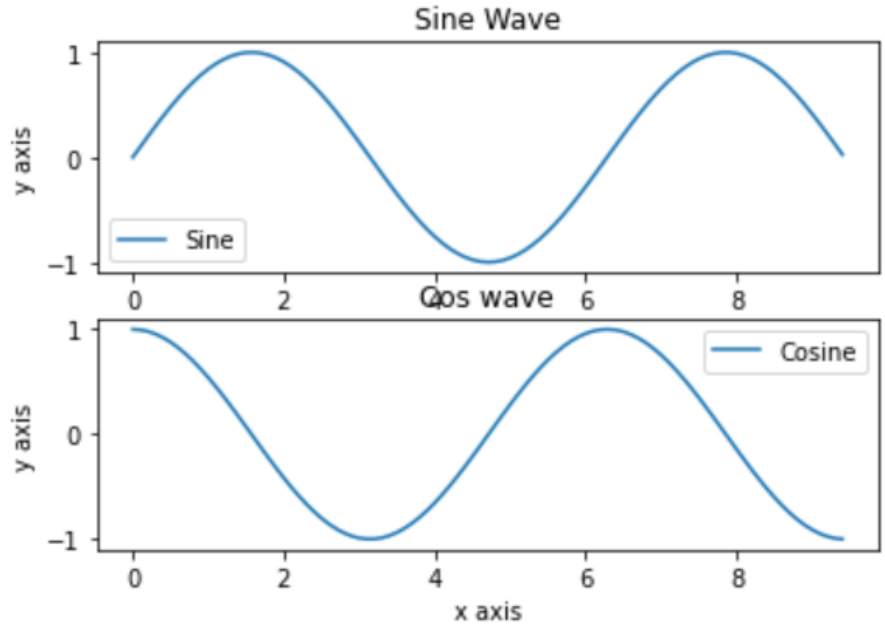

The **Matplotlib** subplot() function can be called to plot two or more plots in one figure.

Matplotlib supports all kind of subplots including 2x1 vertical, 2x1 horizontal or a 2x2 grid.

### 14.1. Horizontal subplot

Use the code below to create a horizontal subplot

```
import numpy as np
#import matplotlib.pyplot as plt
from pylab import*
x=np.arange(0,3*np.pi,0.1)
print(x)
y_sin=np.sin(x)
y_cos=np.cos(x)
subplot(2,1,1)
plot(x,y_sin)
xlabel("x axis")
ylabel("y axis")
title("Sine Wave")
legend(['Sine'])
subplot(2,1,2)
plot(x,y_cos)
xlabel("x axis")
ylabel("y axis")
title("Cos wave")
legend(['Cosine'])
show()
```



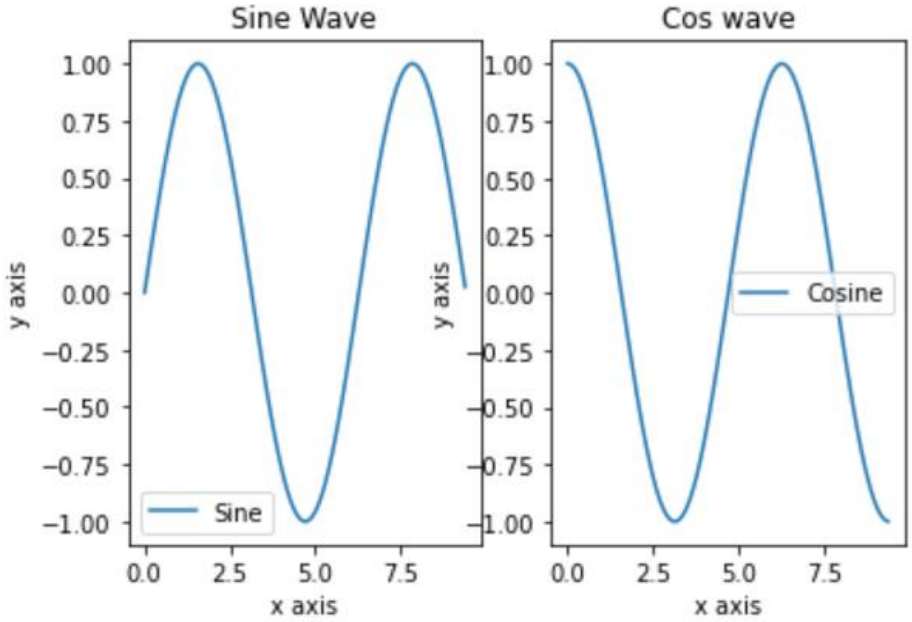

matplotlib subplot

### 14.2 Vertical subplot

By changing the subplot parameters we can create a vertical plot

```
from pylab import *
import numpy as np
#import matplotlib.pyplot as plt
```

```
from pylab import*
x=np.arange(0,3*np.pi,0.1)
print(x)
y_sin=np.sin(x)
y_cos=np.cos(x)
subplot(1,2,1)
plot(x,y_sin)
xlabel("x axis")
ylabel("y axis")
title("Sine Wave")
legend(['Sine'])
subplot(1,2,2)
plot(x,y_cos)
xlabel("x axis")
ylabel("y axis")
title("Cos wave")
legend(['Cosine'])
show()
```



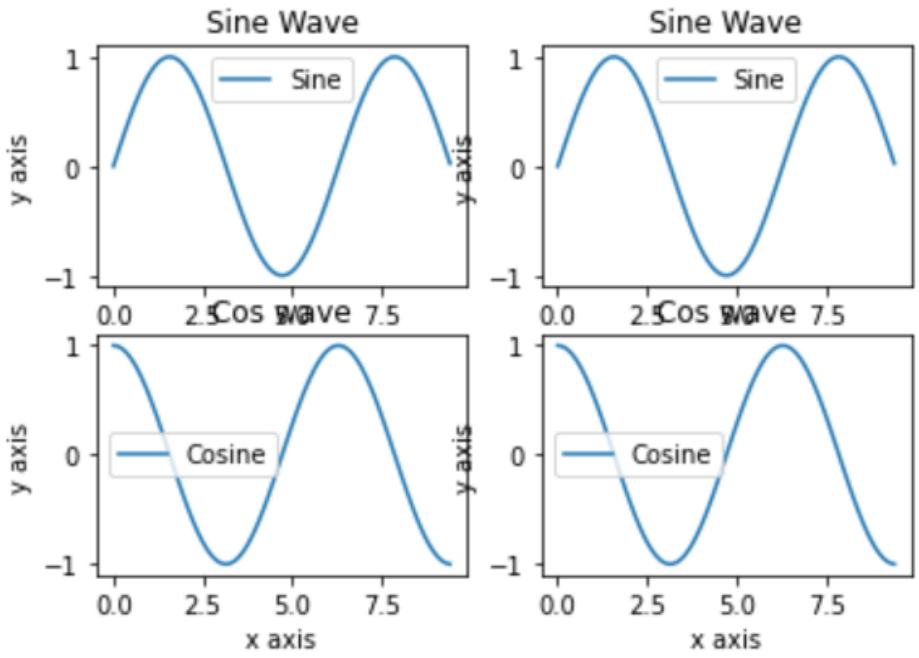

matplotlib subplot vertical

### 14.3. Subplot grid

To create a 2x2 grid of plots, you can use this code:

```
import numpy as np
#import matplotlib.pyplot as plt
from pylab import*
x=np.arange(0,3*np.pi,0.1)
print(x)
y_sin=np.sin(x)
y_cos=np.cos(x)
```

```
subplot(2,2,1)
plot(x,y_sin)
xlabel("x axis")
ylabel("y axis")
title("Sine Wave")
legend(['Sine'])
subplot(2,2,2)
plot(x,y_sin)
xlabel("x axis")
ylabel("y axis")
title("Sine Wave")
legend(['Sine'])
subplot(2,2,3)
plot(x,y_cos)
xlabel("x axis")
ylabel("y axis")
title("Cos wave")
legend(['Cosine'])
subplot(2,2,4)
plot(x,y_cos)
xlabel("x axis")
ylabel("y axis")
title("Cos wave")
legend(['Cosine'])
show()
```




Matplotlib.pyplot.subplots() in Python

**14.4.Display Image**

The image module in Matplotlib package provides functionalities required for loading, rescaling and displaying image.

Loading image data is supported by the Pillow library. Natively, Matplotlib only supports PNG images. The commands shown below fall back on Pillow if the native read fails.

**Method 1:**

The image used in this example is a PNG file, but keep that Pillow requirement in mind for your own data. The **imread()** function is used to read image data in an **ndarray** object of float32 dtype.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
img=mpimg.imread('mtplogo.png')
```

Assuming that following image named as **mtplogo.png** is present in the current working directory.


Any array containing image data can be saved to a disk file by executing the **imsave()** function. Here a vertically flipped version of the original png file is saved by giving origin parameter as lower.

```
plt.imsave("logo.png",img,cmap='gray', origin ='lower')
```

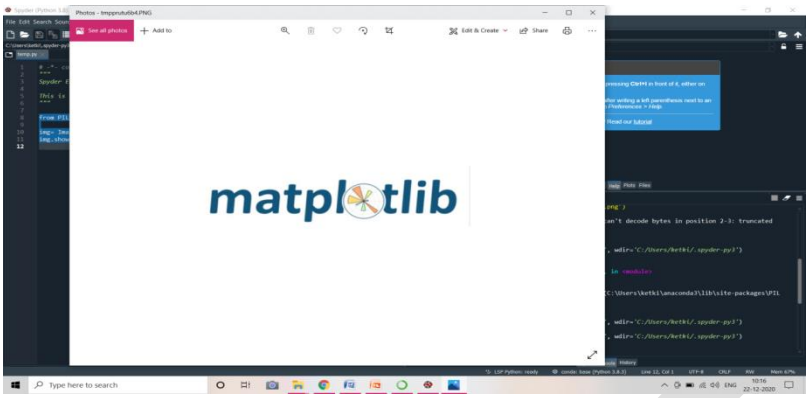
The new image appears as below if opened in any image viewer.

To draw the image on Matplotlib viewer, execute the **imshow()** function.

```
imgplot=plt.imshow(img)
```

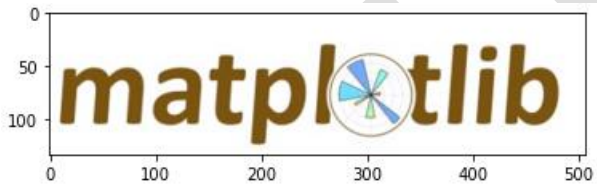
**Method 2:**

```
from PIL import Image
img=Image.open(r"C:\Users\ketki\Desktop\mtplogo.png")
img.show()
```



**Method 3:**

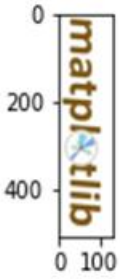
```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('C:/Users/ketki/Desktop/mtplogo.png')
imgplot=plt.imshow(img)
```




**Operation on image: (With open Cv)**

Example 1:

```
import cv2
from pylab import *
import matplotlib.pyplot as plt
img = cv2.imread('C:/Users/ketki/Desktop/mtplogo.png')
subplot(2,2,1)
imgplot=plt.imshow(img)
img_rotate_90_clockwise = cv2.rotate(img, cv2.ROTATE_90_CLOCKWISE)
cv2.imwrite('data/dst/lena_cv_rotate_90_clockwise.jpg', img_rotate_90_clockwise)
subplot(2,2,2)
imgplot=plt.imshow(img_rotate_90_clockwise)
```

Example 2:

```
import cv2
from pylab import *
import matplotlib.pyplot as plt
img = cv2.imread('C:/Users/ketki/Desktop/mtplogo.png')
subplot(2,2,1)
imgplot=plt.imshow(img)
img_rotate_90_clockwise = cv2.rotate(img, cv2.ROTATE_90_CLOCKWISE)
cv2.imwrite('data/dst/lena_cv_rotate_90_clockwise.jpg', img_rotate_90_clockwise)
subplot(2,2,2)
imgplot=plt.imshow(img_rotate_90_clockwise)
img_rotate_90_counterclockwise = cv2.rotate(img,
cv2.ROTATE_90_COUNTERCLOCKWISE)
cv2.imwrite('data/dst/lena_cv_rotate_90_counterclockwise.jpg',
img_rotate_90_counterclockwise)
subplot(2,2,3)
imgplot=plt.imshow(img_rotate_90_counterclockwise)
img_rotate_180 = cv2.rotate(img, cv2.ROTATE_180)
cv2.imwrite('data/dst/lena_cv_rotate_180.jpg', img_rotate_180)
```