In [1]:
```
import numpy as np
```

Combining array

Arrays can be combined in various ways. This process in NumPy is referred to as `stacking` . Stacking can take various forms, including `horizontal` , `vertical` , and `depth-wise` stacking. To demonstrate this, we will use the following two arrays

- `stack` means 1 ke uppar 1 rakhna ya saath rakhna

Stacking

Join a sequence of arrays along a new axix.

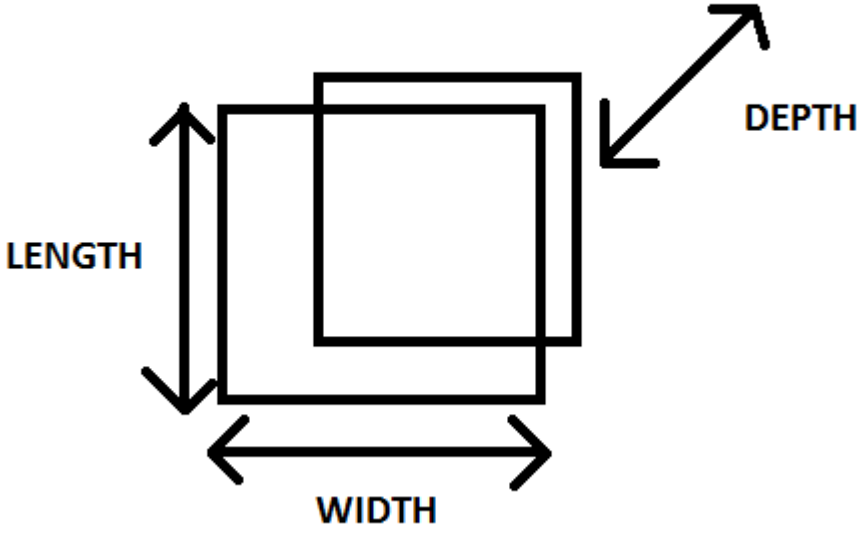
In [2]:
```
# create two arrays

a = np.arange(9).reshape(3,3)
b = np.arange(9,18).reshape(3,3)

print(a)
print("=============")
print(b)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
=============
[[ 9 10 11]
 [12 13 14]
 [15 16 17]]
```

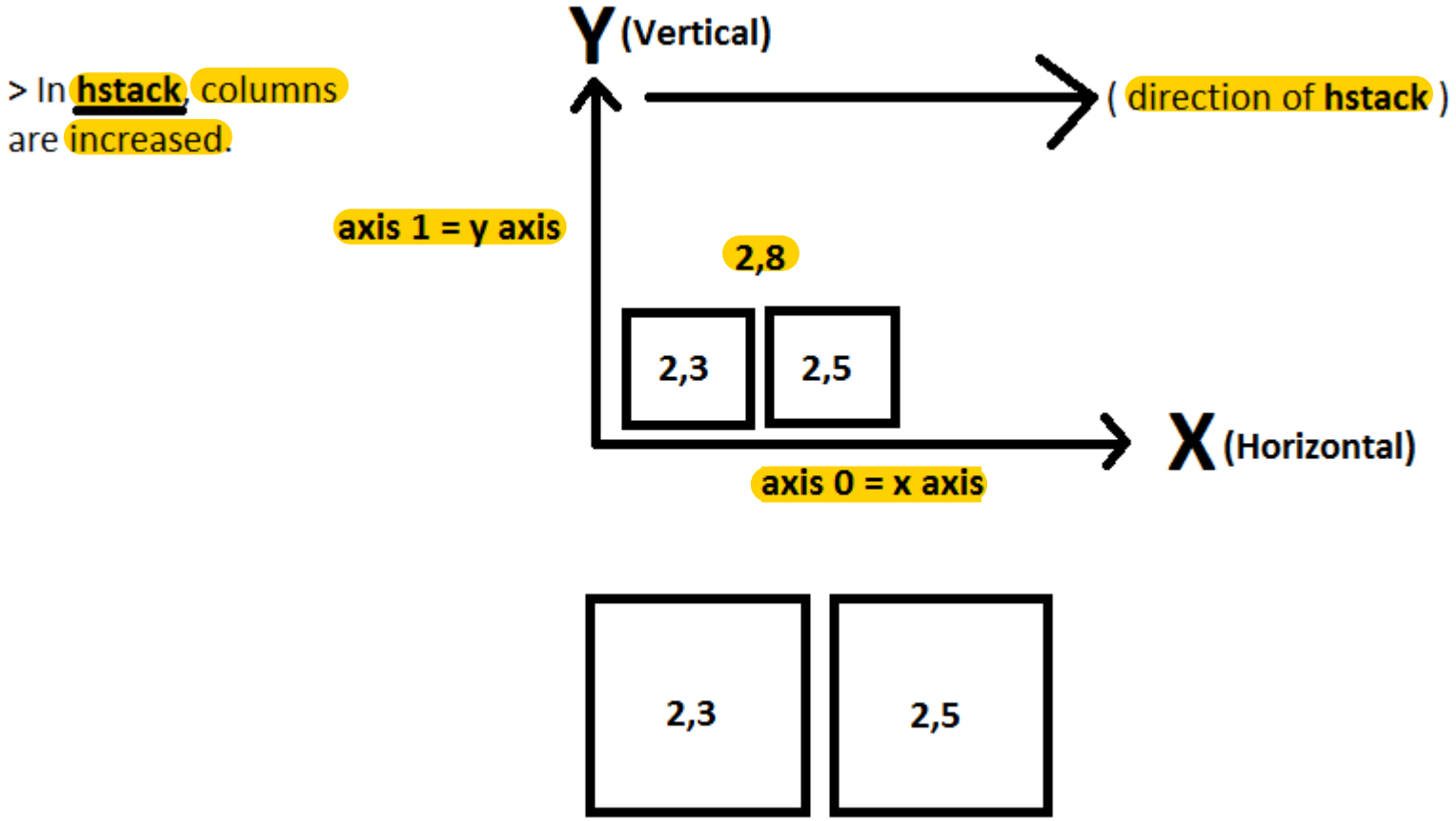
In [4]:
```
# Simply stacking array a and b:
# stack ka matlab ye hai ke 1 array rakhi hui hai, uske peechay ya uppar doosri array ko rakhdena
# samjahna ye hai ke jab ye stack hongi t 3d bnjayegi, How?
np.stack((a,b))
```

Out[4]:
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]]])
```

- 1st array hai mere pass 2 by 2 ki, or 2nd array hai 3 by 3 ki, jab mai (3,3) wali array ko (2,2) par lakar rakhunga to ye ab 3d hogyi 1 dimension iski width hogi, 2nd dimension iski length hogi or 3rd dimension iski depth hogyi, ab ye eik ke peechay eik stack hote jarhi hain



Horizontal Stacking



Combines two arrays in a manner where the columns of the second array are placed to the right of those in the first array. The function actually stacks the two items provided in a two-element tuple. The result is a new array with data copied from the two that are specified:

In [5]:
```
arr1 = np.arange(16).reshape(8,2)
arr1
```

Out[5]:
```
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11],
       [12, 13],
       [14, 15]])
```

In [7]:
```
arr2 = np.arange(16,56).reshape(8,5)
arr2
```

Out[7]:
```
array([[16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25],
       [26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35],
       [36, 37, 38, 39, 40],
       [41, 42, 43, 44, 45],
       [46, 47, 48, 49, 50],
       [51, 52, 53, 54, 55]])
```

In [9]:
```
# dono haath phelana parenge

np.hstack((arr1,arr2))
```

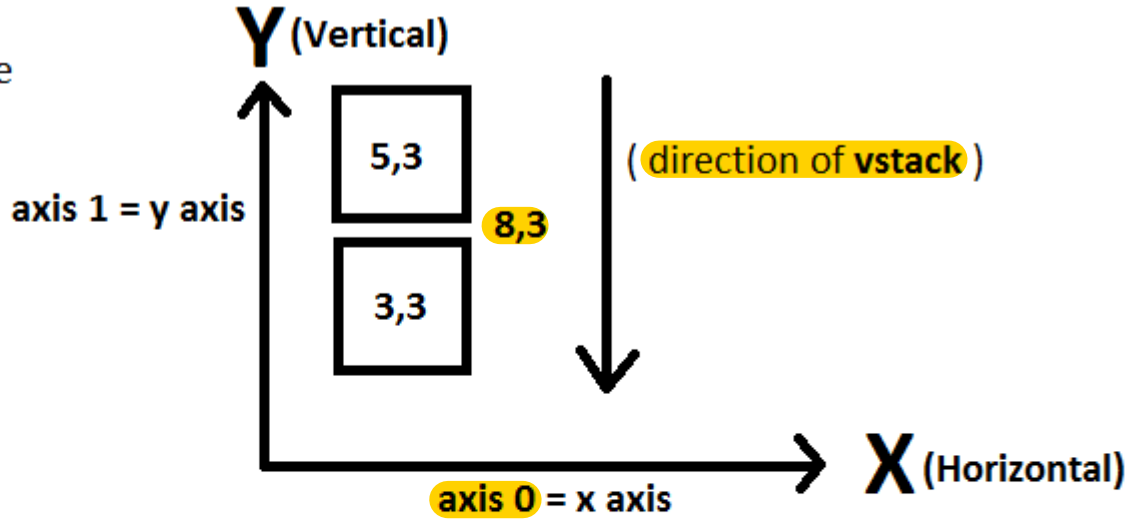
Out[9]:
```
array([[ 0,  1, 16, 17, 18, 19, 20],
       [ 2,  3, 21, 22, 23, 24, 25],
       [ 4,  5, 26, 27, 28, 29, 30],
       [ 6,  7, 31, 32, 33, 34, 35],
       [ 8,  9, 36, 37, 38, 39, 40],
       [10, 11, 41, 42, 43, 44, 45],
       [12, 13, 46, 47, 48, 49, 50],
       [14, 15, 51, 52, 53, 54, 55]])
```

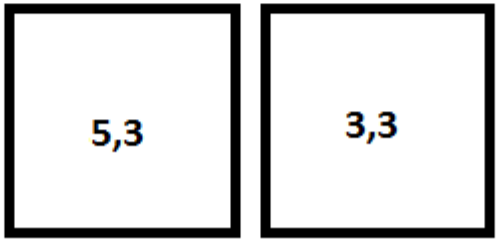
In [10]:
```
# axis 1 hamare pass y axis hota hai >>>> similar to hstack

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

Out[10]:
```
array([[ 0,  1, 16, 17, 18, 19, 20],
       [ 2,  3, 21, 22, 23, 24, 25],
       [ 4,  5, 26, 27, 28, 29, 30],
       [ 6,  7, 31, 32, 33, 34, 35],
       [ 8,  9, 36, 37, 38, 39, 40],
       [10, 11, 41, 42, 43, 44, 45],
       [12, 13, 46, 47, 48, 49, 50],
       [14, 15, 51, 52, 53, 54, 55]])
```

Vertical Stacking

> In **vstack**, rows are increased.





Vertical stacking returns a new array with the contents of the second array as appended rows of the first array.

In [11]:
```
arr3 = np.arange(20).reshape(2,10)
arr3
```

Out[11]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

In [12]:
```
arr4 = np.arange(20,60).reshape(4,10)
arr4
```

Out[12]:
```
array([[20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

In [13]:
```
np.vstack((arr3,arr4))
```

Out[13]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

In [14]:
```
np.concatenate((arr3,arr4),axis=0)
```

Out[14]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

Columns Stacking

DYS

Row Stacking

DYS

Depth Stacking

takes a list of arrays and arranges them in order along an additional axis referred to as the depth:

- dstack stacks each independent column of a and b

In [3]:
```
arr5 = np.arange(9).reshape(3,3)
arr5
```

Out[3]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [5]:
```
arr6 = np.arange(9,18).reshape(3,3)
arr6
```

Out[5]:
```
array([[ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]])
```

In [8]:
```
# jab mai ne kaha isko ke depth stack kardo arr5 or arr6 ko to depth stack bilkul wese hee hoga jese humne si
rf khaali ..
# ..stack mai kia tha, matalab bilkul wese hee nhi hoga lekin banegi ismai 3d hee, jistarah stacking mai 3d b
an rhi thi, 1 ke..
# ..peechay 1 chale gyi thi, dstack mai bhi 3d hee banegi qk depth wise ye 3rd dimension main array ko ye add
karega, matrix..
# ko ye 3eesri dimension mai add karega LEFT, RIGHT, UP and DOWN add nhi karega balke ye backside pe add kare
ga, jab ye ..
# ..backside pe add karega to cube ki tarah 3eesri dimension banna shuru hojayegi, lekin ye to stacking mai h
orha tha,..
#stacking mai wo poori image uthake uske peechay ese hee rakh deta tha, lekin jab depth stacking karenge to i
ska thora format..
# change hai
# ye input arrays ki thori si dimension bhi change karta hai, isne stack ki tarah add nhi kardya balke isne d
imension bhi .
# .. change kardi, sab sai pehle ye baat ke both input arrays 3 by 3 shape ki hain to number of elements hong
e apke pass 18
# ab 18 ka 3d stack kia kia bansakta hai? >>> 3,2,3 =>18 OR >>> 2,3,3 =>18 OR >>> 3,3,2 =>18 to ye 3 possibil
ity hain ye apni..
# marzi sai ab sawaal ye hai ke isne yahan kia hai?? to depth stacking mai 3d mai hee ans ayega

dstacked = np.dstack((arr5,arr6))
print(dstacked)
print(dstacked.shape)
```

```
[[[ 0  9]
  [ 1 10]
  [ 2 11]]

 [[ 3 12]
  [ 4 13]
  [ 5 14]]

 [[ 6 15]
  [ 7 16]
  [ 8 17]]]
(3, 3, 2)
```

Splitting arrays

Arrays can also be split into multiple arrays along the `horizontal`, `vertical` and `depth` axis using the `np.hsplit()`, `np.vsplit()`, and `np.dsplit()` functions. We will only look at the np.hsplit() function as the other work similarly.

In [10]:
```
# if I split into 2 equal parts it will reusult in an error bcoz 9 values can't be splitted into 2

x = np.arange(9.0)
print(x)
print("=============")
print(np.split(x, 3))
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8.]
=============
[array([0., 1., 2.]), array([3., 4., 5.]), array([6., 7., 8.])]
```

In [11]:
```
a = np.arange(12).reshape(3,4)
a
```

Out[11]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [15]:
```
# horizontal split the 2d array into 4 array columns
# horizontal kaatne mai jawab columns mai arhe hain

print(np.hsplit(a,4))
print("========================")
print(np.hsplit(a,2))
```

```
[array([[0],
       [4],
       [8]]), array([[1],
       [5],
       [9]]), array([[ 2],
       [ 6],
       [10]]), array([[ 3],
       [ 7],
       [11]])]
========================
[array([[0, 1],
       [4, 5],
       [8, 9]]), array([[ 2,  3],
       [ 6,  7],
       [10, 11]])]
```

In [17]:
```
print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [16]:
```
# Split at columns 1 and 3

np.hsplit(a, [1,3])
```

Out[16]:
```
[array([[0],
        [4],
        [8]]),
 array([[ 1,  2],
        [ 5,  6],
        [ 9, 10]]),
 array([[ 3],
        [ 7],
        [11]])]
```

Useful numerical methods of NumPy arrays

In [20]:
```
# demonstrate some of the properties of NumPy arrays
m = np.arange(10,19).reshape(3,3)
print(m)
print("{0} min of the entire matrix".format(m.min()))
print("{0} max of the entire matrix".format(m.max()))
print("{0} position of the min value".format(m.argmin()))     # index of minValue, yaani wo minValue kaha par hi hai, value nhi
print("{0} position of the max value".format(m.argmax()))     # index of maxValue
print("{0} mins down each column".format(m.min(axis=0)))      # to har col mai min value 10, 11, 12 hai
print("{0} mins across each row".format(m.min(axis=1)))       # min value of every row
print("{0} maxs down each column".format(m.max(axis=0)))
```

```
[[10 11 12]
 [13 14 15]
 [16 17 18]]
10 min of the entire matrix
18 max of the entire matrix
0 position of the min value
8 position of the max value
[10 11 12] mins down each column
[10 13 16] mins across each row
[16 17 18] maxs down each column
```

Some statistical function

In [21]:
```
my_array = np.arange(36).reshape(6,6)
my_array_array
```

Out[21]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

In [22]:
```
print(f"The mean of the array is {my_array.mean()}")
print(f"The standar deviation of the array is {my_array.std()}")
print(f"The variation of the array is {my_array.var()}")
```

```
The mean of the array is 17.5
The standar deviation of the array is 10.388294694831615
The variation of the array is 107.91666666666667
```

In [23]:
```
print(f"The sum of the array is {my_array.sum()}")
print(f"The product of the array is {my_array.prod()}")
```

```
The sum of the array is 630
The product of the array is 0
```

In [24]:
```
print(f"The cumulative sum of the array is {my_array.cumsum()}")
print(f"The cumulative product of the array is {my_array.cumprod()}")
```

```
The cumulative sum of the array is [  0   1   3   6  10  15  21  28  36  45  55  66  78  91 105 120 136 153
 171 190 210 231 253 276 300 325 351 378 406 435 465 496 528 561 595 630]
The cumulative product of the array is [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0]
```

Applying logical operators on arrays

In [25]:
```
# Logical ka ans hamare pass hamesha True ya False mai ata hai
a = np.arange(10)
a
```

Out[25]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [26]:
```
# .any() returns True if any if any of the values is less than 5

(a < 5).any()   # poori array mai agar 1 bhi less than 5 hooga to ye True return kardega
```

Out[26]:
```
True
```

In [27]:
```
# .all() return True if and only if all the values are less than 5
(a < 5).all()
```

Out[27]:
```
False
```

Boolean Indexing

In [2]:
```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
names
```

Out[2]:
```
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

In [3]:
```
# 1st row Bob ki hai, to har row ka 1 naam banadiya humne
data =np.array(np.random.randn(7, 4)*10,dtype=np.int)
data
```

Out[3]:
```
array([[ -8,   8,   7,   2],
       [ 11, -17,   1, -10],
       [ 17,   0,  -4,   8],
       [  3,  -1,  -2,  -6],
       [-21,  -6,  11,  -2],
       [-15,  17, -13,   4],
       [  9, -11, -14, -16]])
```

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'.

Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized.

Thus, comparing names with the string 'Bob' yields a boolean array:

```
'Bob'   [[ 9, -2,  9, -7],
'Joe'    [12, 14,  0, 14],
'Will'   [ 0,  2,  2,  2],
'Bob'    [-1, -3, -3,  8],
'Will'   [13,  2, -2, -1],
'Joe'    [ 5,  7,  7, -8],
'Joe'    [ 2,  6, 26, -8]])
```

First consdier this, if we wnat to select (fancy selection) different rows from array we can pass a list of row indices.

But a big but, if we want to select different rows based on some criteria like here we want all rows corresponding to name 'Bob'

we will do booean indexing

In [4]:
```
# slicing mai start sai end tak mai mangliya karta tha lekin fancy indexing mai mere farmayishen shuru hogyin
hain, yahan..
# ..main keh rha hun ke 3eesri dedo 5wee dedo or akhri dedo, to humne dekha hai ke selection contiguous form
 mai hota hai..
# ..yani 1 saath 1 honi chahiyen, ye jo idhar udhar sai uthakar dedo ye kese karenge, ye idhar udhar sai utha
n isko kehte..
# ..hain fancy indexing, to fancy indexing mai mujhe jo cheezen chahiyen hoti hain unki mai poori list banaka
r deta hun

data[[0,3,5]]
```

Out[4]:
```
array([[ -8,   8,   7,   2],
       [  3,  -1,  -2,  -6],
       [-15,  17, -13,   4]])
```

In [5]:
```
#boolean indexing now
# to huemin ab boolean indexing karni hai fancy indexing ki madad sai
# poori names array ko compare karega Bob sai, ab jawab main hamare pass boolean result ayega, jahan par Bob
 hoga wahn True..
# ..ayega jahan jahan Bob nhi hoga wahan False ayega

# ab yehi jo result hai mai isko fancy indexing mai index number dene ki bajaye main yehi result True False w
ala yahan pass ..
# ..kardun

names == 'Bob' # vectorize comparision,will return true where name is bob in names array.
```

Out[5]:
```
array([ True, False, False,  True, False, False, False])
```

this result can be passed in to data array to select True rows based on True and False (boolean)

In [6]:
```
# to yahan True wali rows ajayengi or False wali nhi ayengi
# isne mujhe 2 results diye hain joke True aye hain

data[[ True, False, False,  True, False, False, False]]   # this way
```

Out[6]:
```
array([[-8,  8,  7,  2],
       [ 3, -1, -2, -6]])
```

In [7]:
```
# a better way is:
# ab maine yahan par eik boolean expression pass kardi hai
data[names=='Bob'] # definitely inside data brackets boolean array will be generated by names=='Bob'
```

Out[7]:
```
array([[-8,  8,  7,  2],
       [ 3, -1, -2, -6]])
```

In [8]:
```
# wo le ao jahan Bob nhi hain yaani mujhe bar bar True False nhi likhna par rha expression likhne sai mera ka
am asaan hogya
# hum ye usko mask banakar derhe hain joke True False ki soorat mai convert horha hai

data[names!='Bob']
```

Out[8]:
```
array([[ 11, -17,   1, -10],
       [ 17,   0,  -4,   8],
       [-21,  -6,  11,  -2],
       [-15,  17, -13,   4],
       [  9, -11, -14, -16]])
```

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like & (and) and | (or)

In [10]:
```
# wo rows le ao jin ka naam Bob ho ya Will ho
# ab is mask ka ans True ya False mai hee ayega
mask = (names == 'Bob') | (names == 'Will')

# this is multiple names condition generating a sort of mask to provide to data array
##### The Python keywords (and) and (or) do not work with bolean arrays.##############
###### Use &(and) and |(or) instead.. #########3
```

In [11]:
```
data[mask]
```

Out[11]:
```
array([[ -8,   8,   7,   2],
       [ 17,   0,  -4,   8],
       [  3,  -1,  -2,  -6],
       [-21,  -6,  11,  -2]])
```

In [12]:
```
data[data < 0] = 0   # another way! It replaces all the values on True indexing with 0
```

In [14]:
```
# har 0 sai choti value 0 bangyi
data
```

Out[14]:
```
array([[ 0,  8,  7,  2],
       [11,  0,  1,  0],
       [17,  0,  0,  8],
       [ 3,  0,  0,  0],
       [ 0,  0, 11,  0],
       [ 0, 17,  0,  4],
       [ 9,  0,  0,  0]])
```

Fancy Indexing with columns

In [15]:
```
# column number 0 dedo or 3 dedo

data[:, [0,3]]
```

Out[15]:
```
array([[ 0,  2],
       [11,  0],
       [17,  8],
       [ 3,  0],
       [ 0,  0],
       [ 0,  4],
       [ 9,  0]])
```

Transposing Arrays and Swapping Axis

In [17]:
```
# tranpose karne ka matlab hota hai ke array ki direction ko change karna, yani jo row hai wo col banjayege o
r col row banjayega
print(data)

data.T   # T >>> property
```

```
[[ 0  8  7  2]
 [11  0  1  0]
 [17  0  0  8]
 [ 3  0  0  0]
 [ 0  0 11  0]
 [ 0 17  0  4]
 [ 9  0  0  0]]
```

Out[17]:
```
array([[ 0, 11, 17,  3,  0,  0,  9],
       [ 8,  0,  0,  0,  0, 17,  0],
       [ 7,  1,  0,  0, 11,  0,  0],
       [ 2,  0,  8,  0,  0,  4,  0]])
```

In [19]:
```
np.transpose(data)    # now it is function for transposing an array
```

Out[19]:
```
array([[ 0, 11, 17,  3,  0,  0,  9],
       [ 8,  0,  0,  0,  0, 17,  0],
       [ 7,  1,  0,  0, 11,  0,  0],
       [ 2,  0,  8,  0,  0,  4,  0]])
```

Universal functions: Fast Element-wise Array Functions

In [20]:
```
arr = np.arange(10)
arr
```

Out[20]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [22]:
```
# square root nikalega
np.sqrt(arr)
```

Out[22]:
```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])
```

In [23]:
```
# exponent nikaldo
np.exp(arr)
```

Out[23]:
```
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])
```

In [26]:
```
x = np.random.randn(8)*10
y = np.random.randn(8)*10

print(x)
print(y)
```

```
[ -5.49663316  -9.56041831   4.95093046  -0.9657229  -18.51868003
  -4.06422409  17.39609821 -12.58495204]
[ -2.8038292   -8.93213397   4.0305147  -12.30349809 -13.46911593
  10.17398458   3.38524474 -14.52932958]
```

In [27]:
```
# in do array mai jo max value ho usko nayi array ka part banado
np.maximum(x,y)   # compares both arrays element wise and max is included in result
```

Out[27]:
```
array([ -2.8038292 ,  -8.93213397,   4.95093046,  -0.9657229 ,
       -13.46911593,  10.17398458,  17.39609821, -12.58495204])
```

In [28]:
```
arr = np.random.randn(7) * 5
print(arr)
```

```
[-1.75715505 -1.66704554  3.56851443  2.79731953 -5.86275329  9.28218929
 -1.39589745]
```

In [32]:
```
# modf() ka function apke tamaam numbers ko tor deta hai apke decimal part alag kardeta hai or whole part lag kardeta hai
# .. or return karta hai 2 tuple ki sorat mai remainder or whole part

remainder, whole_part = np.modf(arr)  # modf separate whole n decimal parts and return them
```

In [33]:
```
remainder
```

Out[33]:
```
array([-0.75715505, -0.66704554,  0.56851443,  0.79731953, -0.86275329,
        0.28218929, -0.39589745])
```

```
In [34]: whole_part
```

```
Out[34]: array([-1., -1.,  3.,  2., -5.,  9., -1.])
```

List of Unary funcs (Ufuncs) take single array input

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as a separate array
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise (equivalent to `~arr`).

List of binary funcs:

`take two arrays` input

Binary universal functions

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum; fmax ignores NaN
minimum, fmin	Element-wise minimum; fmin ignores NaN
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation (equivalent to infix operators & |, ^)

Linear Algebra

> Like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Unlike some languages like MATLAB, multiplying two two-dimensional arrays with * sign is an element-wise product instead of a matrix dot product. Thus, there is a function dot, both an array method and a function in the numpy namespace, for matrix multiplication. > Linear ALgebra mai matrix opoeratins perform hote hain

In [39]:
```
# ye hamare pass x or y 2 matrix hain

x = np.array([[1.,2.,3.], [4.,5.,6.]])
y = np.array([[6.,23.], [-1,7], [8,9]])

print(x)
print("============")
print(y)
```

```
[[1. 2. 3.]
 [4. 5. 6.]]
============
[[ 6. 23.]
 [-1.  7.]
 [ 8.  9.]]
```

Note:

* to multiply two matrices (element wise multiplication or mirror multiplication) both matrix must have same shape
* matrix multiplication 2 tarah ki hoti hai:
* 1: Mirror Multiplication: element wise hoti hai

In [41]:
```
# to shape same nhi hai to ye error derha hai

x * y
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-41-1b211a33497a> in <module>
      1 # to shape same nhi hai to ye error derha hai
      2
----> 3 x * y

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

In [44]:
```
# now i create matrix with same shape

z = np.arange(6).reshape(2,3)
print(z)
print(x)
```

```
[[0 1 2]
 [3 4 5]]
[[1. 2. 3.]
 [4. 5. 6.]]
```

In [45]:
```
# now i can multiply
# to ye mirror multiplication horhi har element apne saamne wale element sai multiply horha hai to ye matrix multilplication..
# .. ka tareeqa nhi hota (0 x 1.), (1 x .2) and so on
x*z
```

Out[45]:
```
array([[ 0.,  2.,  6.],
       [12., 20., 30.]])
```

In linear algebra rows of first matrix is multiplied by the columns of second matrix and this is only possible when columns of first matrix are equal to the rows of second matrix

(A)			
4	5	3	1
3	8	5	2
5	6	3	4

(B)			
0	8	6	5
1	6	7	8
1	0	3	9
4	9	0	2

Shape or order of Matrix A = (r,c)>>>(3,4)

Shape or order of Matrix B = (r,c)>>>(4,4)

Rule: Col of A == Cols of B

In [46]:
```
# Order of x
x.shape
```

Out[46]:
```
(2, 3)
```

In [47]:
```
# Order of y
y.shape
```

Out[47]:
```
(3, 2)
```

In [48]:
```
# Col of x == rows of y
# hence matrix multiplication can be done
```

In [49]:
```
np.dot(x,y)
```

Out[49]:
```
array([[ 28.,  64.],
       [ 67., 181.]])
```

In [51]:
```
x@y
```

Out[51]:
```
array([[ 28.,  64.],
       [ 67., 181.]])
```

In [52]:
```
np.dot(y,x)  # y.y also fulfilling the rule
```

Out[52]:
```
array([[ 98., 127., 156.],
       [ 27.,  33.,  39.],
       [ 44.,  61.,  78.]])
```

Note:

- It is not necessary if a.b is possible so b.a will be
- np.dot(a,b)
- a.dot(b)
- a@b
- All are same

numpy.linalg

has a standard set of matrix decompositions and things like inverse and determinant. These are implemented under the hood via the same industry standard linear algebra libraries used in other languages like MATLAB and R, such as BLAS, LAPACK, or possibly (depending on your NumPy build) the proprietary Intel MKL (Math Kernel Library):Numpy linear algebra ki 1 library bani wi hai, usse hum kuch functions hum import kar rhe hian

In [53]:
```
from numpy.linalg import inv, qr
```

In [57]:
```
X = np.array(np.random.randn(5,5)*10, dtype="int32")
X
```

Out[57]:
```
array([[  0,   5,  -2,  -3,   2],
       [ -5,   1,   0,   5,   4],
       [ -5,   5,   1, -11,   9],
       [  7,   1,  -8,  -9, -12],
       [ 11,   6,   2, -11,  10]])
```

In [59]:
```
# inverse nikalunga array ka
inv(X)
```

Out[59]:
```
array([[-0.03252818,  0.03220612, -0.05861514,  0.01771337,  0.06763285],
       [ 0.44669887, -0.29871176, -0.06634461, -0.13929147, -0.07729469],
       [ 0.30056361, -0.51912238, -0.02769726, -0.22926731, -0.102657  ],
       [ 0.05958132,  0.04991948, -0.07085346, -0.02254428,  0.00483092],
       [-0.22681159,  0.30253623,  0.03188406,  0.08514493,  0.09782609]])
```

The expression X.T.dot(X) computes the dot product of X with its transpose X.T

In [60]:
```
# X ka transpose lo, uska dot product nikaaldo apne hee X ke sath
mat = X.T.dot(X)
mat
```

Out[60]:
```
array([[ 220,   43,  -39, -154,  -39],
       [  43,   88,   -1, -140,  107],
       [ -39,   -1,   73,   45,  121],
       [-154, -140,   45,  357,  -87],
       [ -39,  107,  121,  -87,  345]])
```

Commonly used numpy.linalg functions

Function	Description
diag	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
dot	Matrix multiplication
trace	Compute the sum of the diagonal elements
det	Compute the matrix determinant
eig	Compute the eigenvalues and eigenvectors of a square matrix
inv	Compute the inverse of a square matrix
pinv	Compute the Moore-Penrose pseudo-inverse of a matrix
qr	Compute the QR decomposition
svd	Compute the singular value decomposition (SVD)
solve	Solve the linear system Ax = b for x, where A is a square matrix
lstsq	Compute the least-squares solution to Ax = b

In [62]:
```
# Question: How can we get element form 3d array?
# Now i want to access element 38

arr3d = np.arange(64).reshape(4,4,4)
arr3d
```

Out[62]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]],

       [[16, 17, 18, 19],
        [20, 21, 22, 23],
        [24, 25, 26, 27],
        [28, 29, 30, 31]],

       [[32, 33, 34, 35],
        [36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]],

       [[48, 49, 50, 51],
        [52, 53, 54, 55],
        [56, 57, 58, 59],
        [60, 61, 62, 63]]])
```

In [64]:
```
# arr3d[depth,row,col]

arr3d[2,1,2]
```

Out[64]:
```
38
```

----------Syllabus Completed----------

Additional And Informational:

Importing Image Data into NumPy Arrays

In machine learning, Python uses image data in the from of a NumPy array, i.e, [Height, Width, Channel] format. To enhance the performance of the predictive model, we must know how to load and manipulate images. In Python, we can perfrom one task in different ways. We have options from NumPy to Pytorch and CUDA, depending on the complexity of the problem.

By the end of this example, you will have hands-on experience with:

```
* Manipulation of an image using the Pillow and NumPy libraries and saving it to your local system.
* Loading and displaying an image using Matplotlib, OpenCV and Keras API
* Converting the loaded images to the NumPy array and back

* Reading images as arrays in Keras API and OpenCV
```

Pillow Library

Pillow is preferred image manipulation tool. Python version 2 used Python Image Library(PIL), and Python version 3 uses Pillow Python Library, an upgrade of PIL.

You should first create a virtual environment in Anaconda for different projects. Make sure you have supporting packages like NumPy, SciPy, and Matplotlib to install in the virtual environment you create.

Once you set up the packages, you can easily install Pillow using pip.

#

```
pip install Pillow
```

#

```
pip install Pillow --upgrade
```

Image ko NumPy array mai convert karne keliye main Python ki libraray Pillow use karunga, ye kaam mai Pillow sai bhi karsakta hun, matplotlib sai bhi karsakta hun, Keras joke TensorFlow ki API hai us sai bhi karsataka hun, OpenCV use karsata hun

In [2]:
```
import PIL
print("Pillow Version:", PIL.__version__)
```

```
Pillow Version: 8.0.1
```

In [3]:
```
# Load and show an image with Pillow
# Image ki class import karli

from PIL import Image
```

In [4]:
```
# Open the image from working directory
image = Image.open('shahid.jpg')

image
```

Out[4]:

In [5]:
```
# summarize some details about the image, row and col hain pixels ki shakal main to 640 rows and 640 cols hai
n
# ye 2 dimensional image hai
print(image.format)
print(image.size)
print(image.mode)
```

In [6]:
```
# Pillow ka eik object hai
type(image)
```

```
JPEG
(640, 640)
RGB
```

Out[6]:
```
PIL.JpegImagePlugin.JpegImageFile
```

Converting image into NumPy array

In [7]:
```
# asarray() ka function data type change karne keliye use karte hain

imgarray = np.asarray(image)
```

In [8]:
```
# ab meri image array bangyi
type(imgarray)
```

Out[8]:
```
numpy.ndarray
```

In [9]:
```
# 3d hai qk color image hai
# width, height, channels(RGB)
imgarray.shape
```

Out[9]:
```
(640, 640, 3)
```

In [10]:
```
# vector banadiya
# 3eeno factors multiply hogye, or 1228800 pixels is image ke mojood hain
imgarray.ravel().shape
```

Out[10]:
```
(1228800,)
```

In [11]:
```
print(640 * 640 * 3)
```

```
1228800
```

Load and display an image with matplotlib

Ab mai yehi kaam matplotlib ki library sai karunga

In [12]:
```
# Load and display an image with matplotlib
# maine image ki class uthali or plot karne ka pyplot uthaliya
from matplotlib import image
from matplotlib import pyplot
```

In [13]:
```
# load image as pixel array
# ab qk ye matplotlib ki library hai to usne automatically numpy array mai convert kardi image
# 3d image hai yaani 3 channels hain iske
img = image.imread('shahid.jpg')
img
```

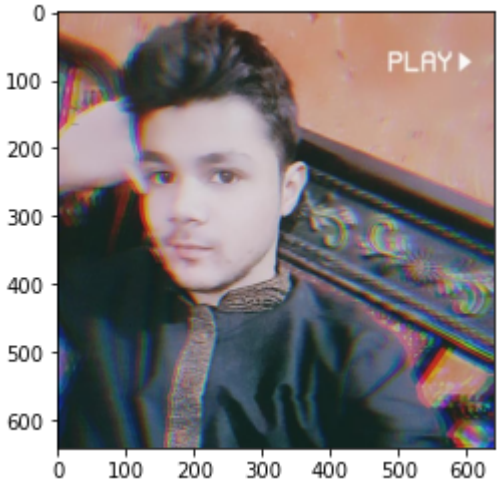
Out[13]:
```
array([[[188, 136, 112],
        [188, 136, 112],
        [188, 136, 112],
        ...,
        [197, 148, 131],
        [197, 148, 131],
        [197, 148, 131]],

       [[188, 136, 112],
        [188, 136, 112],
        [188, 136, 112],
        ...,
        [197, 148, 131],
        [197, 148, 131],
        [197, 148, 131]],

       [[189, 137, 113],
        [189, 137, 113],
        [189, 138, 111],
        ...,
        [197, 148, 131],
        [197, 148, 131],
        [197, 148, 131]],

       ...,

       [[ 21,  38,  56],
        [ 20,  37,  53],
        [ 20,  36,  51],
        ...,
        [ 34,  36, 173],
        [ 44,  53, 184],
        [ 59,  72, 200]],

       [[ 21,  38,  54],
        [ 20,  37,  53],
        [ 20,  36,  51],
        ...,
        [ 34,  39, 169],
        [ 43,  55, 177],
        [ 59,  76, 194]],

       [[ 21,  38,  54],
```

```
        [ 20,  38,  52],
        [ 20,  36,  51],
        ...,
        [ 33,  38, 164],
        [ 40,  55, 174],
        [ 55,  75, 188]]], dtype=uint8)
```

In [14]:
```
# summarize shape of the pixel array
print(img.dtype)
print(img.shape)
```

```
uint8
(640, 640, 3)
```

In [15]:
```
# display the array of pixels as an image
# array of pixels ko image main kese convert karenge, to pehle imread ka function tha jo read karta hai image ko array mai
# ab mai yahan imshow() ka function ko use kar rha hun jo dobarah array ko img mai convert kardega
pyplot.imshow(img)
pyplot.show()
```



Manipulating and saving the image

Now that we have converted our image into a NumPy array, we might come across a case where we need to do some manipulations on an image before using it into the desired model. In this section you will be able to build a grayscale converter. You can also resize the array of the pixel image and trim it.

After performing the manipulations, it is important to save the image before performing further steps. The format argument saves the file in different formats such as PNG, GIF, or JPEG.

For example, the code below loads the photograph in JPEG format and saves it in PNG format

to kia mai 2 images ko apas mai jor sakta hun ? concatenate karlenege

In [19]:
```
import numpy as np
from PIL import Image

myImg = Image.open('shahid.jpg')
myImg = myImg.convert('L')        # L isko grayscale mai convert kardega
myImg = np.array(myImg)
# myImg = np.array(Image.open('abc.jpg').convert())  # uppar 3 lines waal kaam 1 line mai bhi hosakta hai

print(type(myImg))

gr_img = Image.fromarray(myImg).save('gr_shahid.png')

print(gr_img)
```

```
<class 'numpy.ndarray'>
None
```

In [ ]:
```

```