Numpy

NumPy objects: NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.
NumPy main objects is the homogenous multidimension array. It is a table of elements all of the same type, indexed by a tuple of non-negative integers. NumPy arrays are stored at one continuous place in memory unlike lists, so processes numpy array can access and manipulate them very efficiently.

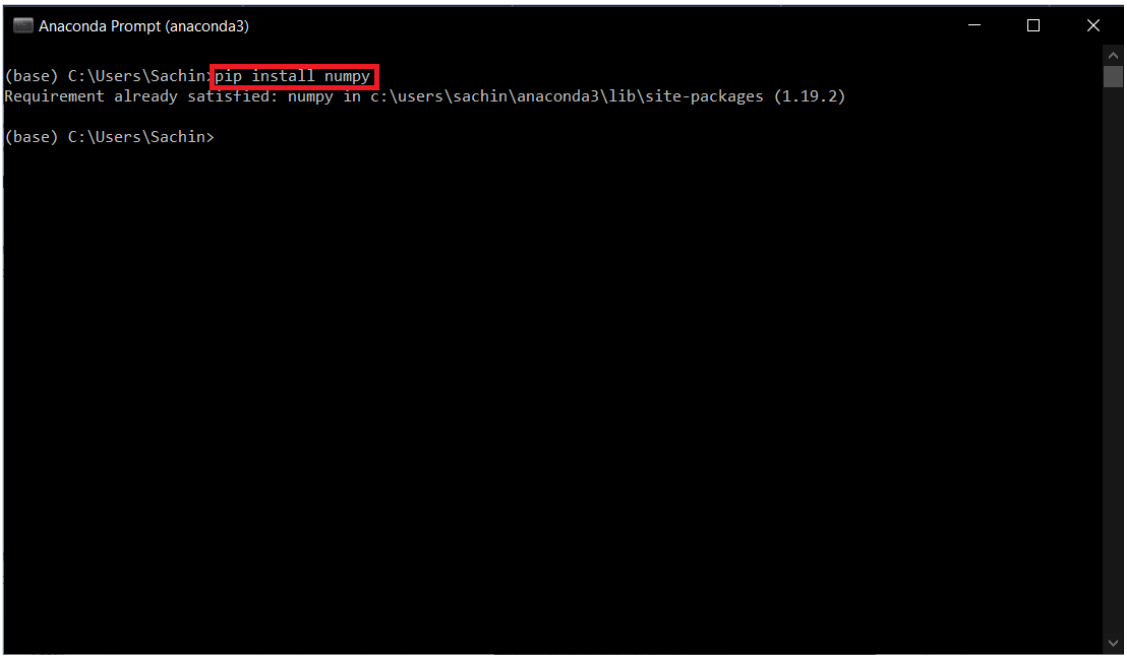
NOTE: In, Numpy dimensions are called axes.

> **There are several important differences between NumPy arrays and the standard Python sequences:**

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

Installing numPy

To install Python NumPy, go to your command prompt and type **"pip install numpy"**. Once the installation is completed, go to your IDE (For example: PyCharm) or Anaconda Jupyter Notebook and simply import it by typing: **"import numpy as np"**.



How to import NumPy

In order to start using NumPy and all of the functions available in NumPy, you'll need to import it. This can be easily done with this import statement:

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

NOTE: We shorten numpy to np in order to save time and also to keep code standardized so that anyone working with your code can easily understand and run it.

How to create numpy array

numpy.array(): array() function is used to create an array with data type and value list specified in its arguments.

Syntax:

In [ ]:
```
numpy.array(object)
```

Example: Create one Dimension array

In [2]:
```
import numpy as np
a = np.array([1,2,3])   #This is one axis.This axis has 3 elements.
print(a)
```

```
[1 2 3]
```

Example: Create two Dimension array

In [3]:
```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])   #This has two axes. The first axis hav
print(a)
```

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

In [4]:
```
import numpy as np
a = np.array([[1,2,3],[4,5,6],[7,8]])
```

```
print(a)
```

```
[list([1, 2, 3]) list([4, 5, 6]) list([7, 8])]
```

```
<ipython-input-4-0b4d934daac8>:2: VisibleDeprecationWarning: Creating an
ndarray from ragged nested sequences (which is a list-or-tuple of lists-o
r-tuples-or ndarrays with different lengths or shapes) is deprecated. If
you meant to do this, you must specify 'dtype=object' when creating the n
darray
  a = np.array([[1,2,3],[4,5,6],[7,8]])
```

NOTE: You must have the same no of elements in every row. You can't have different size of elements in row.

NOTE: NumPy array class is called **ndarray**. It is also known by the alias array.

Why we need numPy array when we have List:

Memory consumption between Numpy array and lists:

The very first reason to choose python NumPy array is that it occupies less memory as compared to list.

In [5]:
```
import sys
li_arr=[i for i in range(100)]  #create list array of size 100

print("List array: ",li_arr)
print("Type of array: ",type(li_arr))
print("How much size one element take: ",sys.getsizeof(li_arr[0]))    #c
print("How much size list array take: ",(sys.getsizeof(li_arr[0]))*len(l
```

```
List array:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1
7, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 3
5, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 5
3, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 7
1, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 8
9, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
Type of array:  <class 'list'>
How much size one element take:  24
How much size list array take:  2400
```

Code Explanation:As you can see from above code. We create a list array **li_arr** of 100 size. We calculate the size of each element. Each element take 24 bit in List array and the then the whole size of the List array take 2400 bit.

In [6]:
```
import numpy as np
np_arr=np.arange(100)    #create numPy array of size 100

print("numPy array: ",np_arr)
print("Type of array: ",type(np_arr))
print("How much size one element take: ",np_arr.itemsize)     #calculate
print("How much size list array take: ",np_arr.itemsize*np_arr.size)
```

```
numPy array:  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 1
9 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 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
Type of array:  <class 'numpy.ndarray'>
How much size one element take:  4
How much size list array take:  400
```

Code Explanation:As you can see from above code. We create a numpy array **np_arr** of 100 size. We calculate the size of each element. Each element take 4 bit in numpy array and the then the whole size of the numpy array take 400 bit.

Conclusion:

The above output shows that the memory allocated by list (denoted by **li_arr**) is 2400 whereas the memory allocated by the NumPy array **np_arr** is just 400. 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.

Time comparison between Numpy array and Python lists:

NumPy array is pretty fast in terms of execution numPy array takes less amount of time than List array to perform operation. Let's have a look.

In [7]:
```
import time

li_arr1=[i for i in range(100000)]
li_arr2=[i for i in range(100000)]
start_time=time.time()
li_add=[li_arr1[i]+li_arr2[i] for i in range(100000)]
final_time=time.time()

print("Time taken by Lists to perform addition: ",(final_time-start_time
```

```
Time taken by Lists to perform addition:  16.953706741333008
```

Code Explanation: As you can see from the above code. We create two list array **li_arr1** and **li_arr2** and perform addition element wise it takes time around 11.507 millisecond.

In [8]:
```
import time

np_arr1=np.arange(100000)
np_arr2=np.arange(100000)
start_time=time.time()
np_add=np_arr1+np_arr2
final_time=time.time()

print("Time taken by Lists to perform addition: ",(final_time-start_time
```

```
Time taken by Lists to perform addition:  0.4239082336425781
```

Code Explanation: As you can see from the above code. We create two numPy array **np_arr1** and **np_arr2** and perform addition element wise it takes memory around 0.4239 millisecond.

Conclusion:

In 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 11.965ms whereas the numpy array took almost 0.99ms. 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 np_arr1+np_arr2. That’s why working with numpy is much easier and convenient when compared to the lists.

Functionality:

numPy array provide large amount of functionality. If you have to perfrom any operation. You don't need to do element wise. You just have to write a+b and numpy will do addition element wise and give result to you.

In [9]:
```
import numpy as np

li_arr1=[i for i in range(10)]
li_arr2=[i for i in range(10)]
li_add=[li_arr1[i]+li_arr2[i] for i in range(10)]
print(li_add)
print()
np_arr1=np.arange(10)
np_arr2=np.arange(10)
np_add=np_arr1+np_arr2
print(np_add)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

[ 0  2  4  6  8 10 12 14 16 18]
```

Some important points about Numpy arrays:

- We can create a N-dimensional array in python using numpy.array().
- Array are by default Homogeneous, which means data inside an array must be of the same Datatype.
- Element wise operation is possible.
- Numpy array has the various function, methods, and variables, to ease our task of matrix computation.
- Elements of an array are stored contiguously in memory. For example, all rows of a two dimensioned array must have the same number of columns. Or a three dimensioned array must have the same number of rows and columns on each card.

Why numPy Arrays are faster than Python Lists because of the following reasons:

- An array is a collection of homogeneous data-types that are stored in contiguous memory locations. On the other hand, a list in Python is a collection of heterogeneous data types stored in non-contiguous memory locations.
- The NumPy package breaks down a task into multiple fragments and then processes all the fragments parallelly.
- The NumPy package integrates C, C++, and Fortran codes in Python. These programming languages have very little execution time compared to Python.

Data types in numPy:

numPy has some extra data types, and refer to data types with one character.

In [ ]:
```
i      integers
b      boolean
u      unsigned integer
f      float
c      complex float
m      time delta
M      datetime
o      object
s      string
u      unicode string
v      fixed chunk of memory for other types
```

Numpy array creation routines

numpy.empty(): It creates an uninitialized array of specified shape and dtype. It uses the following constructor

Syntax:

In [ ]:
```
numpy.empty(shape, dtype=float)
```

Example:

In [10]:
```
import numpy as np
x = np.empty(3)
print(x)
```

```
[4.24399158e-314 8.48798317e-314 1.27319747e-313]
```

In [11]:
```
import numpy as np
x = np.empty([3,3])
print(x)
```

```
[[0.00000000e+000 0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000 6.06712613e-321]
 [2.11392372e-307 1.69117157e-306 6.23059726e-307]]
```

In [12]:
```
import numpy as np
x = np.empty([3,3], dtype=int)
print(x)
```

```
[[3014709 3014704      49]
 [5177420 4259907 4259916]
 [5242960 4259908 4259924]]
```

NOTE: The elements in an array show random values as they are not initialized.

numpy.zeros(): Returns a new array of specified size, filled with zeros

Syntax:

In [ ]:
```
numpy.zeros(shape, dtype=float)
```

Example:

In [14]:
```
import numpy as np
x = np.zeros(5)
print(x)
```

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

In [15]:
```
import numpy as np
x = np.zeros([3,5])
```

```
print(x)
```

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

In [16]:
```
import numpy as np
x = np.zeros([3,5], dtype=int)
print(x)
```

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

numpy.ones(): Returns a new array of specified size and type, filled with ones.

Syntax:

In [ ]:
```
numpy.ones(shape, dtype=float)
```

Example:

In [17]:
```
import numpy as np
x = np.ones(5)
print(x)
```

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

In [18]:
```
import numpy as np
x = np.ones([3,3])
print(x)
```

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

In [19]:
```
import numpy as np
x = np.ones([3,3],dtype=int)
print(x)
```

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

numpy.arange(): This function returns an ndarray object containing evenly spaced values within a given range.

Syntax:

In [ ]:
```
numpy.arange(start,stop,step,dtype=None)
```

Example:

In [21]:
```
import numpy as np
x = np.arange(5) #default value of start is 0 and step is 1
print(x)
```

```
[0 1 2 3 4]
```

In [22]:
```
import numpy as np
x = np.arange(1,11) #default value of step is 1
print(x)
```

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

In [23]:
```
import numpy as np
```

```
x = np.arange(1,10,2)
print(x)
```

```
[1 3 5 7 9]
```

numpy.linspace(): This function is similar to arange() function. In this function, instead of step size, the number of evenly spaced values between the interval is specified.

Syntax:

In [ ]:
```
numpy.linspace(start,stop,num=50,dtype=None)
```

Example:

In [24]:
```
import numpy as np
x = np.linspace(1,100)
print(x)
```

```
[  1.           3.02040816   5.04081633   7.06122449   9.08163265
  11.10204082  13.12244898  15.14285714  17.16326531  19.18367347
  21.20408163  23.2244898   25.24489796  27.26530612  29.28571429
  31.30612245  33.32653061  35.34693878  37.36734694  39.3877551
  41.40816327  43.42857143  45.44897959  47.46938776  49.48979592
  51.51020408  53.53061224  55.55102041  57.57142857  59.59183673
  61.6122449   63.63265306  65.65306122  67.67346939  69.69387755
  71.71428571  73.73469388  75.75510204  77.7755102   79.79591837
  81.81632653  83.83673469  85.85714286  87.87755102  89.89795918
  91.91836735  93.93877551  95.95918367  97.97959184 100.        ]
```

In [25]:
```
import numpy as np
x = np.linspace(10,20,5)
print(x)
```

```
[10.  12.5 15.  17.5 20. ]
```

numpy.logspace(): This function returns an ndarray object that contains the numbers that are evenly spaced on a log scale. Start and stop endpoints of the scale are indices of the base, usually 10.

Syntax:

In [ ]:
```
numpy.logspace(start,stop,num=50,dtype=None)
```

Example:

In [26]:
```
import numpy as np
# default base is 10
a = np.logspace(1, 50)
print(a)
```

```
[1.e+01 1.e+02 1.e+03 1.e+04 1.e+05 1.e+06 1.e+07 1.e+08 1.e+09 1.e+10
 1.e+11 1.e+12 1.e+13 1.e+14 1.e+15 1.e+16 1.e+17 1.e+18 1.e+19 1.e+20
 1.e+21 1.e+22 1.e+23 1.e+24 1.e+25 1.e+26 1.e+27 1.e+28 1.e+29 1.e+30
 1.e+31 1.e+32 1.e+33 1.e+34 1.e+35 1.e+36 1.e+37 1.e+38 1.e+39 1.e+40
 1.e+41 1.e+42 1.e+43 1.e+44 1.e+45 1.e+46 1.e+47 1.e+48 1.e+49 1.e+50]
```

In [27]:
```
import numpy as np
# default base is 10
a = np.logspace(1.0, 50.0, num = 10)
print(a)
```

```
[1.00000000e+01 2.78255940e+06 7.74263683e+11 2.15443469e+17
 5.99484250e+22 1.66810054e+28 4.64158883e+33 1.29154967e+39
 3.59381366e+44 1.00000000e+50]
```

rand(): Create an array of the given shape and populate it with random samples from a

uniform distribution over [0, 1].

Syntax:

In [ ]:
```
numpy.random.rand(d0, d1,..., dn)
```

Example:

In [28]:
```
import numpy as np
np.random.rand(2)
```

Out[28]:
```
array([0.2140632 , 0.61256127])
```

In [29]:
```
import numpy as np
np.random.rand(3,3)
```

Out[29]:
```
array([[0.61137563, 0.91107174, 0.89263204],
       [0.69246493, 0.14974486, 0.39764434],
       [0.82721706, 0.8448935 , 0.68690354]])
```

randn(): Return a sample (or samples) from the "standard normal" distribution. Unlike rand which is uniform:

Syntax:

In [ ]:
```
numpy.random.rand(d0, d1,..., dn)
```

Example:

In [30]:
```
np.random.randn(2)
```

Out[30]:
```
array([-0.07108736,  0.79518728])
```

In [31]:
```
import numpy as np
np.random.randn(3,3)
```

Out[31]:
```
array([[-0.68062144,  0.66716992,  1.44250905],
       [ 0.90220174, -2.55390536,  0.27463694],
       [-1.3574581 , -0.39897059, -0.16041008]])
```

The most important attributes of an ndarray objects are:

1. **ndarray.ndim:** The numbers of axes(dimensions) of the array.
2. **ndarray.shape:** The dimension of the array.This is a tuple of integers indicating the size of array in each dimensions. For a matrix with n rows and m columns, shape will be (n*m)
3. **ndarray.size:** The total numbers of elements of the array. This is equal to the products of the elements of the shape
4. **ndarray.dtype:** An objects describing the type of elements of the array one can create or specify dtype using standard Python types
5. **ndarray.itemsize:** This array attribute returns the length of each element of array in bytes.
6. **ndarray.data:** The buffer containing the actual elements of the array. Normally, we wont need to use this attribute because we will access the elements in the arrayusing index facilities.

Example:

In [32]:
```
import numpy as np

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

print("Array a is: :",a)
print("Array b is: ",b)

print("\nNo. of dimension of a: ",a.ndim)
print("No. of dimension of b: ",b.ndim)

print("\nShape of array a: ",a.shape)
print("Shape of array b: ",b.shape)

print("\nSize of a: ",a.size)
print("Size of b: ",b.size)

print("\nData type of a: ",a.dtype)
print("Data type of b: ",b.dtype)

print("\nItemsize of a: ",a.itemsize)
print("Itemsize of a: ",b.itemsize)

print("\nData of a is: ",a.data)
print("Data of b is: ",b.data)
```

```
Array a is: : [1 2 3]
Array b is:  [[1 2 3]
 [4 5 6]]

No. of dimension of a:  1
No. of dimension of b:  2

Shape of array a:  (3,)
Shape of array b:  (2, 3)

Size of a:  3
Size of b:  6

Data type of a:  int32
Data type of b:  int32

Itemsize of a:  4
Itemsize of a:  4

Data of a is:  <memory at 0x000001CD592BFDC0>
Data of b is:  <memory at 0x000001CD57B7BBA0>
```

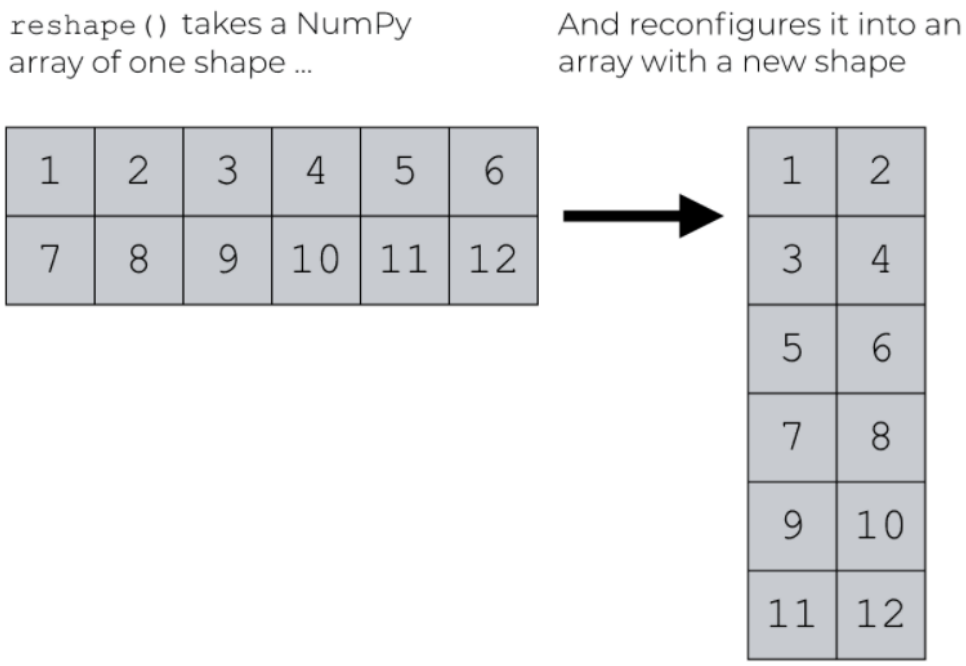
Numpy array reshaping:

- Reshaping means changing the shape of the array.
- The shape of the array is the number of elements in each dimensions.
- By reshaping we can add or remove dimensions or change number of elements in each dimension.

Syntax:

In [ ]:
```
new_array=old_array.reshape((m,n))
```

m,n specifies the tuple of value specifies the new shape.



Example:

In [33]:
```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5, 6],[7, 8, 9, 10, 11, 12]])
newarr = arr.reshape(6, 2)          #reshaping the arr into 6×2 matrix
print(newarr)
```

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

In [34]:
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
newarr = arr.reshape(4, 4)          #reshaping the arr into 4×4 matrix
print(newarr)
```

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

In [35]:
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)          #reshaping the arr into 2×3×2 matr
print(newarr)
```

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

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

In [36]:
```
import numpy as np
```

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

```
---------------------------------------------------------------------------
--
ValueError                                Traceback (most recent call las
t)
<ipython-input-36-ef36cc95f6ef> in <module>
      2
      3 arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
----> 4 newarr = arr.reshape(3, 3)
      5 print(newarr)

ValueError: cannot reshape array of size 8 into shape (3,3)
```

NOTE: You can see from the above code. As long as elements required for reshaping are equal in both shapes.

We can reshape an 8 elements of 1D array into 4 elements 2 rows in 2D array but we can not reshape it into a 3 elements 3 rows in 2D array as that would require 3*3=9 elements

Unknown Dimension

You are allowed to have one "unknown" dimension.

Meaning that you don not have to specify an exact number for one of the one dimension in the reshape method.

Pass -1 as the value, and numPy will calculate this Number for you.

Example:

In [37]:
```
import numpy as np

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

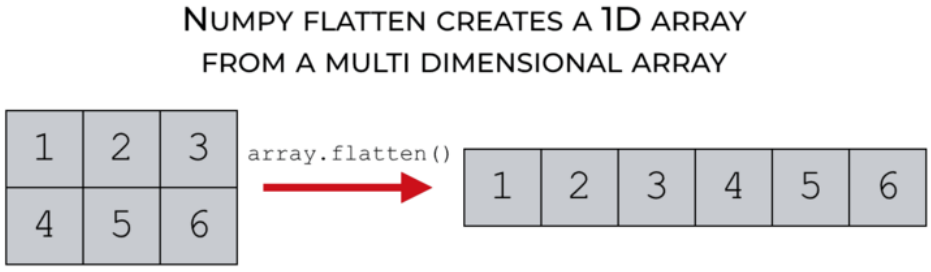
```
[[1 2 3 4]
 [5 6 7 8]]
```

In [38]:
```
import numpy as np

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

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

Flattening a NumPy array



flatten(): The flatten() function is used to get a copy of an given array collapsed into one dimension.

Example:

In [39]:
```
import numpy as np
y = np.array([[1,1], [1,1]])
y.flatten()
```

Out[39]:
```
array([1, 1, 1, 1])
```

ravel(): 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.

Example:

In [40]:
```
import numpy as np
x = np.array([[1,1], [1,1]])
print(np.ravel(x))
```

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

NOTE: The important diferrence between ravel() and flatten() is that flatten() returns the copy of the new array while the ravel() return the reference of original array. This means if we try to make any change in flatten() array it will not affect the original array but in ravel() it will affect the original array.

In [41]:
```
import numpy as np

arr=np.array([[1,2,3],[4,5,6],[7,8,9]])
ravel_arr=arr.ravel()
flatten_arr=arr.flatten()

print("Array is: \n",arr)
print("Ravel array: ",ravel_arr)
print("Flatten array: ",flatten_arr)
print("**********************************")
flatten_arr[0]=22
print(arr)
print("**********************************")
ravel_arr[0]=22
print(arr)
```

```
Array is:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Ravel array:  [1 2 3 4 5 6 7 8 9]
Flatten array:  [1 2 3 4 5 6 7 8 9]
**********************************
[[1 2 3]
 [4 5 6]
 [7 8 9]]
**********************************
[[22  2  3]
 [ 4  5  6]
 [ 7  8  9]]
```

Code Explanation: As you can see in the above code when i tried to update first element in flatten array it didn't affect the original array but in the case of ravel it affect the original array and update the value of first element to 22.

Difference between flatten() and ravel()

arr.ravel():

1. Return only reference/view of original array.
2. ravel is faster than flatten() as it does not occupy any memory.

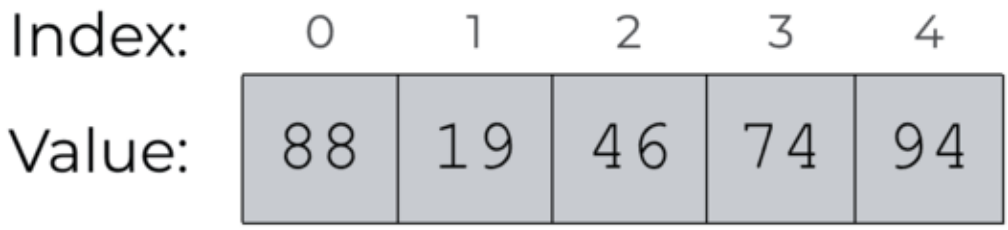
arr.flatten():

1. Return copy of original array.
2. flatten is comparitively slower than ravel() as it occupies memory.

Array indexing:

You can access an array elements by referring to its index number.
The indexes in numpy array starts with 0, meaning that the first has index 0 and the 2nd has index 1 etc.

One Dimension array:

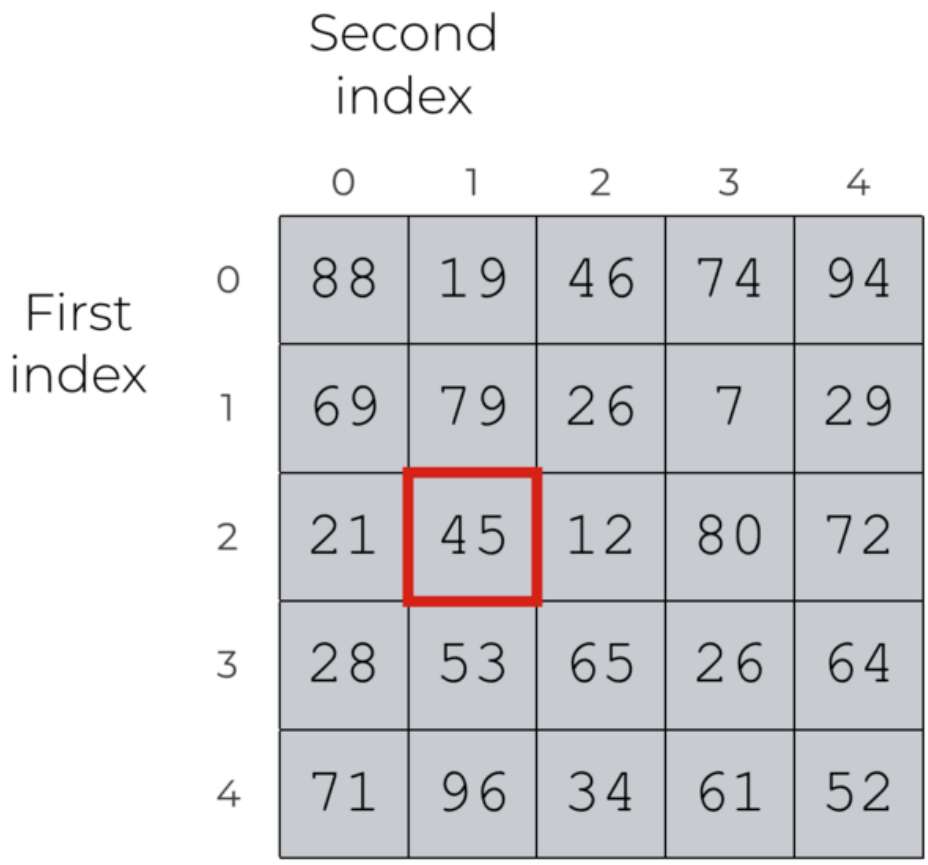


Example:

In [42]:
```
import numpy as np
arr=np.array([88,19,46,74,94])
print(arr[1])
```

```
19
```

Two Dimension array:



Example:

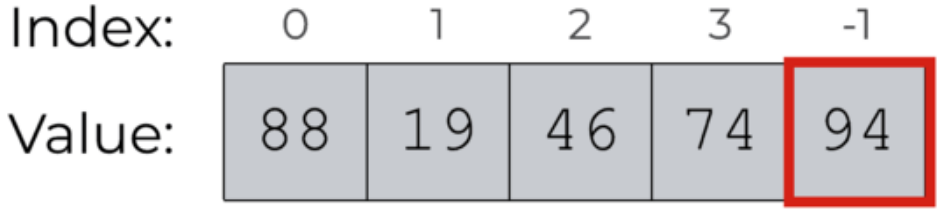
In [43]:
```
import numpy as np
arr=np.array([[81,19,46,74,94],[69,79,26,7,29],[21,45,12,80,72],[28,53,6
print(arr[0])         #it will print first row
print(arr[1])         #it will print second row

print(arr[0][1])      #it will print second element of first row
print(arr[2][1])      #it will print second element of third row.
```

```
[81 19 46 74 94]
[69 79 26  7 29]
19
45
```

Negative indexing:

You can also specify negative indexes while accessing the array. Consider the example given below



One Dimension array:

Example:

In [44]:
```
import numpy as np

arr=np.array([88,19,46,74,94])
print(arr[-1])
print(arr[-2])
print(arr[-3])
```

```
94
74
46
```

Two Dimension array:

Example:

In [45]:
```
import numpy as np
arr=np.array([[1,2,3,4,5],[6,7,8,9,10]])
print(arr[1,-1])
```

```
10
```

In [46]:
```
import numpy as np
arr=np.array([[1,2,3,4,5],[6,7,8,9,10]])
print(arr[0,-3])
```

```
3
```

In [47]:
```
import numpy as np
arr=np.array([[1,2,3,4,5],[6,7,8,9,10]])
print(arr[-1,3])
```

```
9
```

In [48]:
```
import numpy as np
arr=np.array([[1,2,3,4,5],[6,7,8,9,10]])
print(arr[-1,-1])
```

```
10
```

Slicing:

numPy array slicing refers to accessing portion or a subset of numPy array while the original array remain unaffected. You can use indexes of array elements to create array slice as per the following syntax:

slice=[StartIndex : StopIndex : Steps]

- The StartIndex represents the index from where the array slicing is supposed to begin. Its default value is 0, i.r., the array begins from index 0 if no StartIndex is specified.
- The StopIndex represents the last index upto which the array slicing will go on. Its default value is (length(array)-1) or the index of the last element in the array element in the array.
- Steps represent the number of steps. It is an optional parameter. Steps, if defined, specifies the number of elements to jump over while counting from StartIndex to StopIndex. By default it is 1.

- The array slices created, include elements failing between the indexes StartIndex and StopIndex, including StartIndex and not including StopIndex

Slicing in One Dimension array:

Example:

In [49]:
```
import numpy as np
arr=np.array([0,1,2,3,4,5,6,7,8,9,10])
print(arr[:5])                        #by default its startindex start fro
```

```
[0 1 2 3 4]
```

In [50]:
```
import numpy as np
arr=np.array([0,1,2,3,4,5,6,7,8,9,10])
print(arr[5:])                        #by default its stopindex go till la
```

```
[ 5  6  7  8  9 10]
```

In [51]:
```
import numpy as np
arr=np.array([0,1,2,3,4,5,6,7,8,9,10])
print(arr[4:10:2])
```

```
[4 6 8]
```

In [52]:
```
import numpy as np
arr=np.array([1,2,3,4,5,6,7,8,9,10])
print(arr[::])                       #by default it startindex is 0 stopin
```

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

Slicing in Two Dimension array:

Example:

In [53]:
```
import numpy as np

arr=np.array([[1,2,3,4,5,6],[7,8,9,10,11,12],[13,14,15,16,17,18]])
print(arr[0,1:5])                #it select first row and element from 1
```

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

In [54]:
```
import numpy as np

arr=np.array([[1,2,3,4,5,6],[7,8,9,10,11,12],[13,14,15,16,17,18]])
print(arr[1,1:5])                #it select second row and element from
```

```
[ 8  9 10 11]
```

In [55]:
```
import numpy as np

arr=np.array([[1,2,3,4,5,6],[7,8,9,10,11,12],[13,14,15,16,17,18]])
print(arr[0:3,1])                #it select row from first to second and
```

```
[ 2  8 14]
```

In [56]:
```
import numpy as np

arr=np.array([[1,2,3,4,5,6],[7,8,9,10,11,12],[13,14,15,16,17,18]])
print(arr[1:4,2])                #it select row from first to third and
```

```
[ 9 15]
```

In [57]:
```
import numpy as np
```

```
arr=np.array([[1,2,3,4,5,6],[7,8,9,10,11,12],[13,14,15,16,17,18]])
print(arr[1:4,2:4])              #it select row from first to third and
```

```
[[ 9 10]
 [15 16]]
```

Numpy Array copy vs view

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

Example:

In [58]:
```
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]
```

view(): 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.

Example:

In [59]:
```
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]
```

Check if array owns it's data:

As you know copy owns the data and view does not own the data, but how can we check this?
Every numPy has the attribute "base" that return none if array owns the data.
Otherwise the "base" attribute refers to the original object.

Example:

In [60]:
```
import numpy as np
arr=np.array([1,2,3,4])
x=arr.copy()
y=arr.view()
print(x.base)
print(y.base)
```

```
None
[1 2 3 4]
```

Iterating arrays using nditer():

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

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

Example:

In [61]:
```
import numpy as np
arr = np.array([1,2,3,4,5])
for x in np.nditer(arr):
  print(x)
```

```
1
2
3
4
5
```

In [62]:
```
import numpy as np
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
for x in np.nditer(arr):
  print(x)
```

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

Enumerated iteration using ndenumerate:

Enumeration means something sequence number of something one by one. sometime we require corresponding index of the elements while iterating, the ndenumerate() method can be used for those usecases

Example:

In [63]:
```
import numpy as np
arr=np.array([1,2,3])
for idx,x in np.ndenumerate(arr):
    print("In index {} element present is {}".format(idx,x))
```

```
In index (0,) element present is 1
In index (1,) element present is 2
In index (2,) element present is 3
```

In [64]:
```
import numpy as np
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
for idx,x in np.ndenumerate(arr):
    print("In index {} element present is {}".format(idx,x))
```

```
In index (0, 0, 0) element present is 1
In index (0, 0, 1) element present is 2
In index (0, 1, 0) element present is 3
In index (0, 1, 1) element present is 4
In index (1, 0, 0) element present is 5
In index (1, 0, 1) element present is 6
```

```
In index (1, 1, 0) element present is 7
In index (1, 1, 1) element present is 8
```

Numpy Joining Array:

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

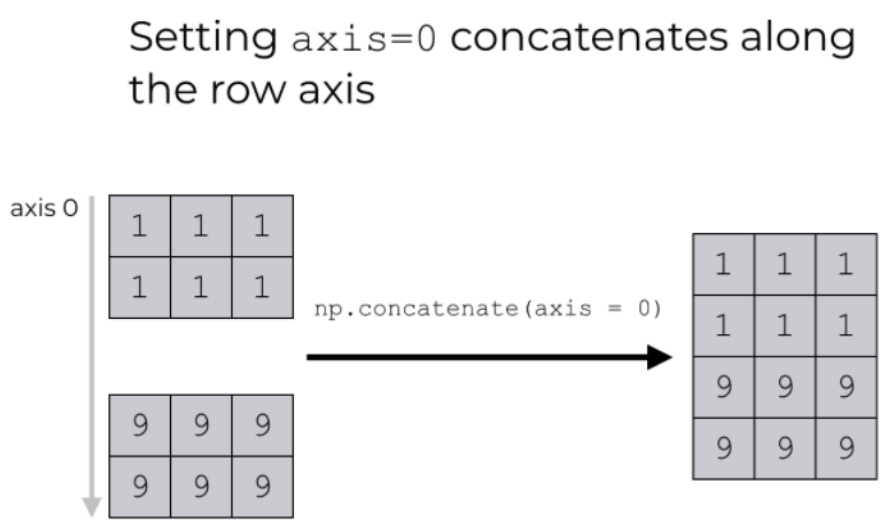
We pass a sequence of arrays that we want to join to the concatenate() function,along with the axis. If the axis is not explicitly passed, it taken as 0

Example:

In [65]:
```
import numpy as np
arr1=np.array([1,2,3])
arr2=np.array([4,5,6])
arr=np.concatenate((arr1,arr2),axis=0)
print(arr)
```

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

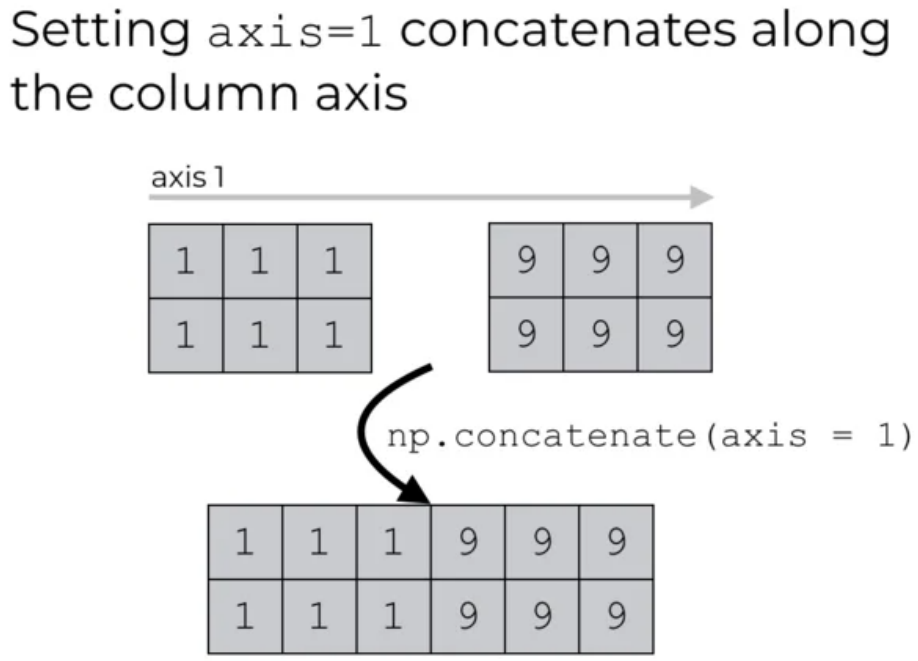
Example:Join two 2D arrays along row(axis=0)



In [66]:
```
import numpy as np
arr1=np.array([[1,1,1],[1,1,1]])
arr2=np.array([[9,9,9,],[9,9,9]])
arr=np.concatenate((arr1,arr2),axis=0)
print(arr)
```

```
[[1 1 1]
 [1 1 1]
 [9 9 9]
 [9 9 9]]
```

Example:Join two 2D arrays along column(axis=1)



In [67]:
```
import numpy as np
arr1=np.array([[1,1,1],[1,1,1]])
arr2=np.array([[9,9,9,],[9,9,9]])
arr=np.concatenate((arr1,arr2),axis=1)
print(arr)
```

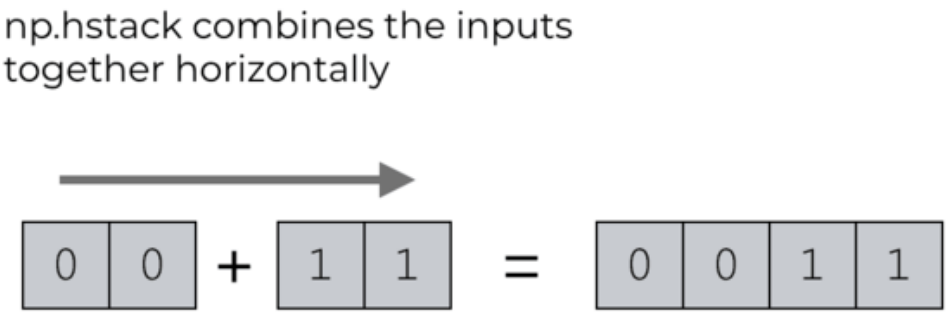
```
[[1 1 1 9 9 9]
 [1 1 1 9 9 9]]
```

Joining array using stack function:

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

One Dimension array

Example: stacking along horizontal wise:

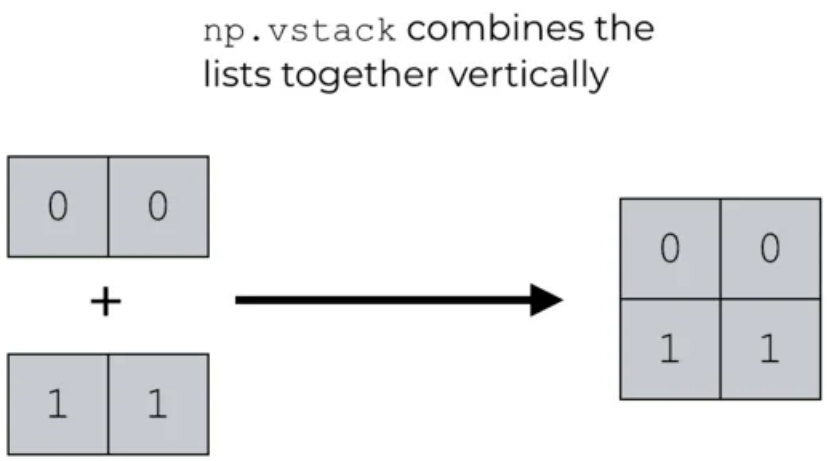


In [68]:
```
import numpy as np
arr1=np.array([0,0])
arr2=np.array([1,1])
```

```
arr=np.hstack((arr1,arr2))
print(arr)
```

```
[0 0 1 1]
```

Example: stacking along vertical wise:

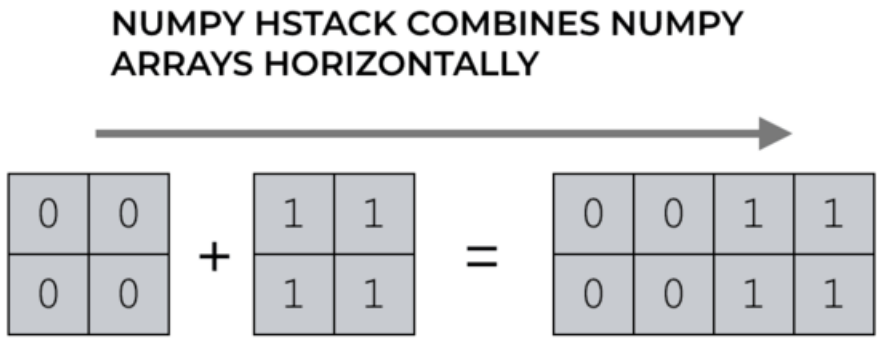


In [69]:
```
import numpy as np
arr1=np.array([0,0])
arr2=np.array([1,1])
arr=np.vstack((arr1,arr2))
print(arr)
```

```
[[0 0]
 [1 1]]
```

Two Dimension array

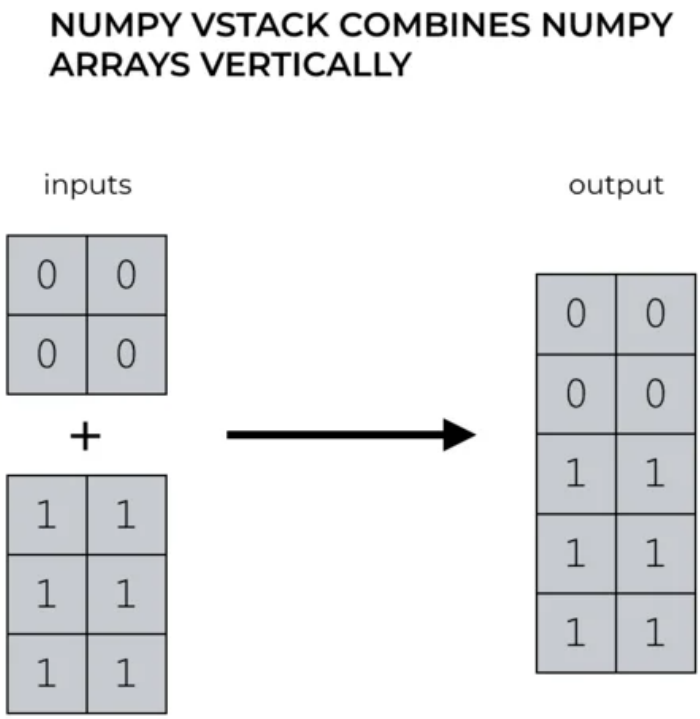
Example: stacking along horizontal wise:



In [70]:
```
import numpy as np
arr1=np.array([[0,0],[0,0]])
arr2=np.array([[1,1],[1,1]])
arr=np.hstack((arr1,arr2))
print(arr)
```

```
[[0 0 1 1]
 [0 0 1 1]]
```

Example: stacking along vertical wise:



In [71]:
```
import numpy as np
arr1=np.array([[0,0],[0,0]])
arr2=np.array([[1,1],[1,1]])
arr=np.vstack((arr1,arr2))
print(arr)
```

```
[[0 0]
 [0 0]
 [1 1]
 [1 1]]
```

Splitting array

we use array_split() for splitting the one array into multiple.

One Dimension array

Example:

In [72]:
```
import numpy as np
arr=np.array([1,2,3,4,5,6])
newarr=np.array_split(arr,2)
print(newarr)
```

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

In [73]:
```
import numpy as np
arr=np.array([1,2,3,4,5,6])
newarr=np.hsplit(arr,3)
print(newarr)
```

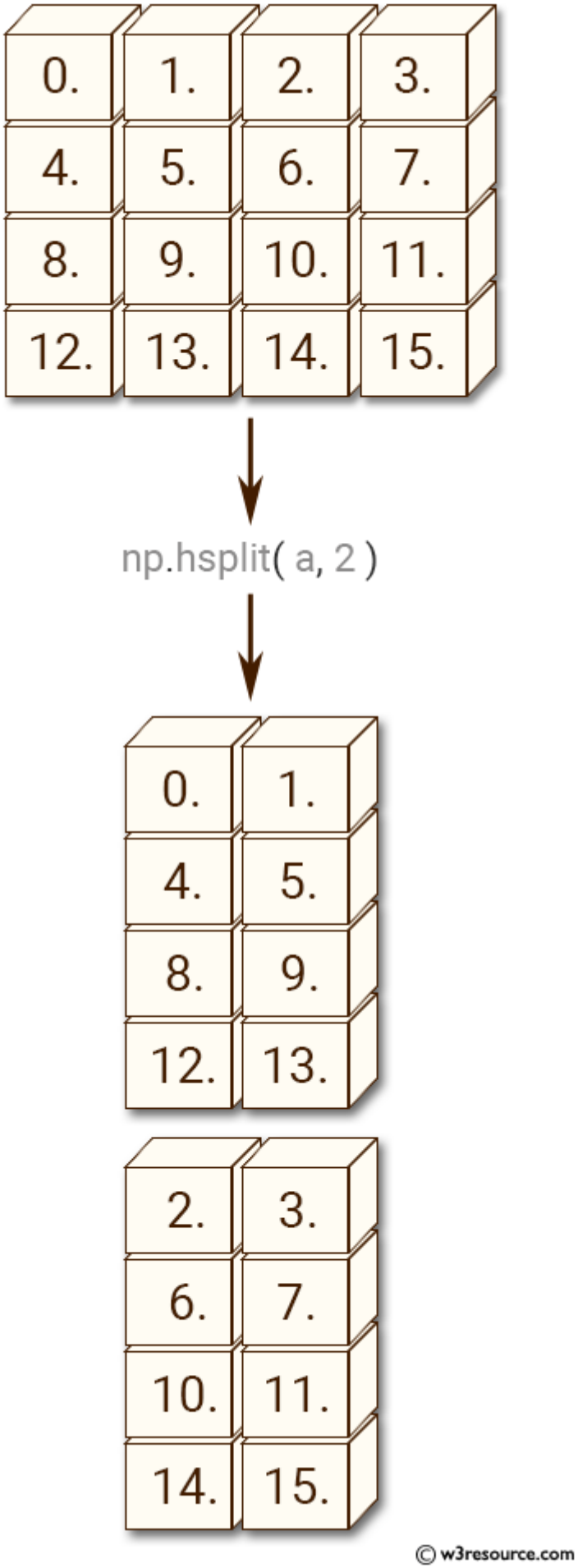
```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

In [74]:
```
import numpy as np
arr=np.array([1,2,3,4,5,6])
newarr=np.hsplit(arr,6)
print(newarr)
```

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

Two Dimension array

Example: numpy.hsplit() function split an array into multiple sub-arrays horizontally (column-wise). hsplit is equivalent to split with axis=1, the array is always split along the second axis regardless of the array dimension.

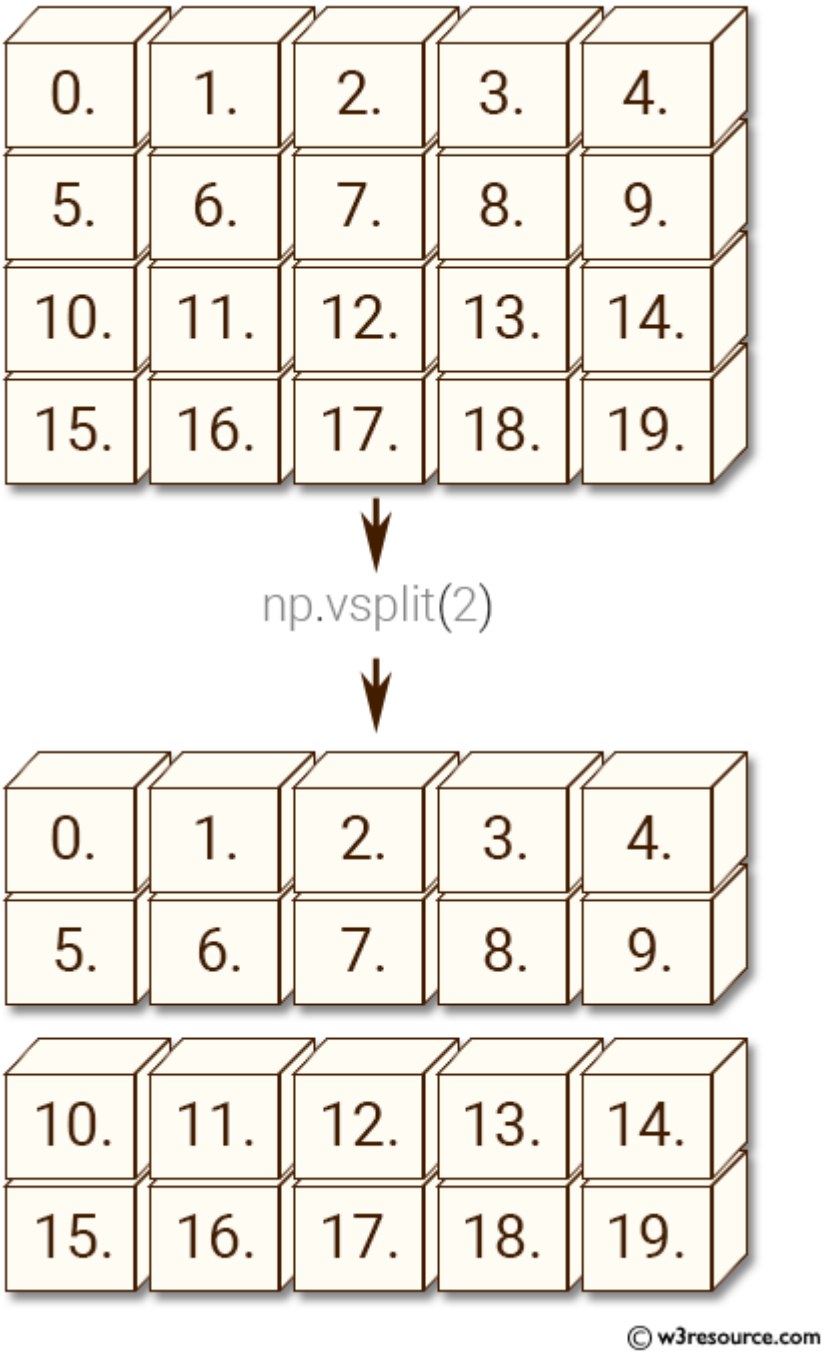


In [75]:
```
import numpy as np
```

```
arr=np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11],[12,13,14,15]])
newarr=np.hsplit(arr,2)
print(newarr)
```

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

Example: numpy.hsplit() function split an array into multiple sub-arrays horizontally (column-wise). hsplit is equivalent to split with axis=1, the array is always split along the second axis regardless of the array dimension.



In [76]:
```
import numpy as np
arr=np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11],[12,13,14,15]])
```

```
newarr=np.vsplit(arr,2)
print(newarr)
```

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

NOTE: If the array has less elements than required, it will adjust from the end accordingly.

Searching arrays:

You can search an array for a certain value, and return the indexes that get a match.
\ To search an array, use the where() method.

Example:

In [77]:
```
import numpy as np
arr=np.array([1,2,3,4,5,6,7])
x=np.where(arr==4)
print(x)
```

```
(array([3], dtype=int64),)
```

Sorted arrays:

sorting means putting elements in a ordered sequences.
Ordered sequences is any sequences that has an order corresponding to elements like numeric or alphabetical, ascending or descending.

Example:

In [78]:
```
import numpy as np
arr=np.array([3,2,0,1])
print(np.sort(arr))
```

```
[0 1 2 3]
```

NOTE: This method returns a copy of the array leaving the original array unchanged.

Numpy filter array:

Getting some elements out of an existing array and creating a new array out of them is called filtering.
In numpy, you filter an array using a boolean corresponding to indexes in the array.
If the value at an index is True that elements is contained in the filtered array, if the value at that index is False that elements is executed from the filtered array.

Example:

In [79]:
```
import numpy as np
arr=np.array([41,42,43,44])
x=[True,False,True,False]
newarr=arr[x]
print(newarr)
```

```
[41 43]
```

Example:

```

```

In [80]:
```
import numpy as np
arr=np.array([41,42,43,44])
filter_arr=[]

for elements in arr:
    if elements>42:
        filter_arr.append(True)
    else:
        filter_arr.append(False)

newarr=arr[filter_arr]
print(newarr)
```

```
[43 44]
```

Matrix:

Matrix is the collection of numbers arranged into rectangular form of row and columns.

we can implement matrix:

1. Numpy arrays
2. Matrix() function

Example: Implementing matrix using numpy array

In [81]:
```
import numpy as np
a=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(type(a))
print(a)
```

```
<class 'numpy.ndarray'>
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Example: Implementing matrix using matrix function

In [82]:
```
import numpy as np
a=np.matrix([[1,2,3],[4,5,6]])
print(type(a))
print(a)
```

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

Operation on matrix

In [83]:
```
import numpy as np

A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
B=np.matrix([[10,20,30],[40,50,60],[70,80,90]])
print("Matrix A is: \n",A)
print("Matrix B is: \n",B)
```

```
Matrix A is:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Matrix B is:
 [[10 20 30]
```

```
 [40 50 60]
 [70 80 90]]
```

Addition: Two matrices can be added together if and only if they have the same dimension. Their sum is obtained by summing each element of one matrix to the corresponding element of the other matrix.

In [84]:
```
import numpy as np

A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
B=np.matrix([[10,20,30],[40,50,60],[70,80,90]])
print("Addition of matrix A and B: \n",A+B)
```

```
Addition of matrix A and B:
 [[11 22 33]
 [44 55 66]
 [77 88 99]]
```

subtraction: Two matrices can be subtracted together if and only if they have the same dimension. Their subtraction is obtained by subtracting each element of one matrix to the corresponding element of the other matrix.

In [85]:
```
import numpy as np

A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
B=np.matrix([[10,20,30],[40,50,60],[70,80,90]])
print("Subtraction of matrix A and B: \n",A-B)
```

```
Subtraction of matrix A and B:
 [[ -9 -18 -27]
 [-36 -45 -54]
 [-63 -72 -81]]
```

multiplication: In Matrix multiplication is the binary operation on two matrices, resulting in the formation of one matrix. For multiplication, the number of columns of the first matrix should be equal to the second matrix's number of rows. In the resulting matrix, its number of rows will be equal to that of the first matrix, and the number of columns will be equal the to columns of the second matrix.

In [86]:
```
import numpy as np

A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
B=np.matrix([[10,20,30],[40,50,60],[70,80,90]])
print("Multiplication of matrix A and B using dot method: \n",np.dot(A,B
print("Multiplication of matrix A and B using dot method: \n",np.matmul(A
```

```
Multiplication of matrix A and B using dot method:
 [[ 300  360  420]
 [ 660  810  960]
 [1020 1260 1500]]
Multiplication of matrix A and B using dot method:
 [[ 300  360  420]
 [ 660  810  960]
 [1020 1260 1500]]
```

transpose(): Transpose of a matrix is obtained by changing rows to columns and columns to rows. In other words, transpose of A[][] is obtained by changing A[i][j] to A[j][i].

In [87]:
```
import numpy as np

A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
B=np.matrix([[10,20,30],[40,50,60],[70,80,90]])
```

```
print("Transpose of A: \n",A.transpose())
print("Transpose of A: \n",B.transpose())
```

```
Transpose of A:
 [[1 4 7]
 [2 5 8]
 [3 6 9]]
Transpose of A:
 [[10 40 70]
 [20 50 80]
 [30 60 90]]
```

Inverse of matrix:

In [88]:
```
import numpy as np
A=np.matrix([[1,2,3],[0,1,4],[5,6,0]])
print("Matrix A is: \n",A)
B=np.linalg.inv(A)
print("Inverse of matrix A is: \n",B)
```

```
Matrix A is:
 [[1 2 3]
 [0 1 4]
 [5 6 0]]
Inverse of matrix A is:
 [[-24.  18.   5.]
 [ 20. -15.  -4.]
 [ -5.   4.   1.]]
```

Power of matrix:

In [89]:
```
import numpy as np
A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
print("Matrix A is: ",A)
B=np.linalg.matrix_power(A,2)
print("Power of matrix A is: ",B)
```

```
Matrix A is:  [[1 2 3]
 [4 5 6]
 [7 8 9]]
Power of matrix A is:  [[ 30  36  42]
 [ 66  81  96]
 [102 126 150]]
```

Determinant of a matrix:

In [90]:
```
import numpy as np
A=np.matrix([[2,3,1],[6,5,2],[1,4,7]])
print("Matrix A is: \n",A)
B=np.linalg.det(A)
print("Inverse of matrix A is: ",B)
```

```
Matrix A is:
 [[2 3 1]
 [6 5 2]
 [1 4 7]]
Inverse of matrix A is:  -46.99999999999999
```

min(): function is used to find the element-wise minimum of array elements.

In [91]:
```
import numpy as np
A=np.matrix([[2,3,1],[6,5,2],[1,4,7]])
print("Matrix A is: \n",A)
print("The minimum in array a is: ",np.min(A))
```

```
Matrix A is:
 [[2 3 1]
 [6 5 2]
```

```
 [1 4 7]]
The minimum in array a is:  1
```

argmin(): Returns the indices of the minimum values along an axis.

In [92]:
```
import numpy as np
A=np.matrix([[2,3,1],[6,5,2],[1,4,7]])
print("Matrix A is: \n",A)
print("The index of minimum in array a is: ",a.argmin())
```

```
Matrix A is:
 [[2 3 1]
 [6 5 2]
 [1 4 7]]
The index of minimum in array a is:  0
```

max(): function is used to find the element-wise maximum of array elements.

In [93]:
```
import numpy as np
A=np.matrix([[2,3,1],[6,5,2],[1,4,7]])
print("Matrix A is: \n",A)
print("The maximum in array a is: ",np.max(A))
```

```
Matrix A is:
 [[2 3 1]
 [6 5 2]
 [1 4 7]]
The maximum in array a is:  7
```

argmax(): Returns the indices of the maximum values along an axis.

In [94]:
```
import numpy as np
A=np.matrix([[2,3,1],[6,5,2],[1,4,7]])
print("Matrix A is: \n",A)
print("The index of maximum in array a is: ",a.argmax())
```

```
Matrix A is:
 [[2 3 1]
 [6 5 2]
 [1 4 7]]
The index of maximum in array a is:  5
```

Array broadcasting:

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.

In [95]:
```
from numpy import array
a = array([1.0, 2.0, 3.0])
b = array([2.0, 2.0, 2.0])
a * b
```

Out[95]:
```
array([2., 4., 6.])
```

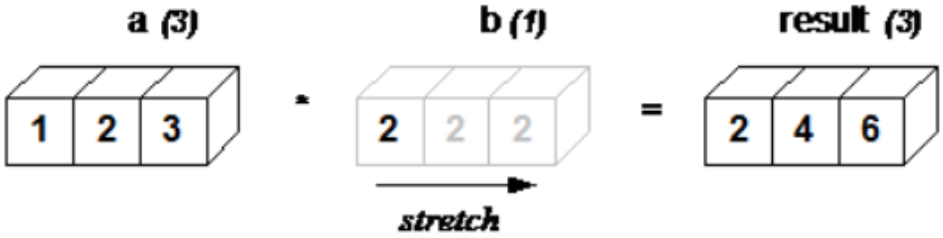
This code is working fine because both the array is of same size. But in case if one of the

array is of different size then what happens. Let's see

In [96]:
```
from numpy import array
a = array([1.0,2.0,3.0])
b = 2.0
a * b
```

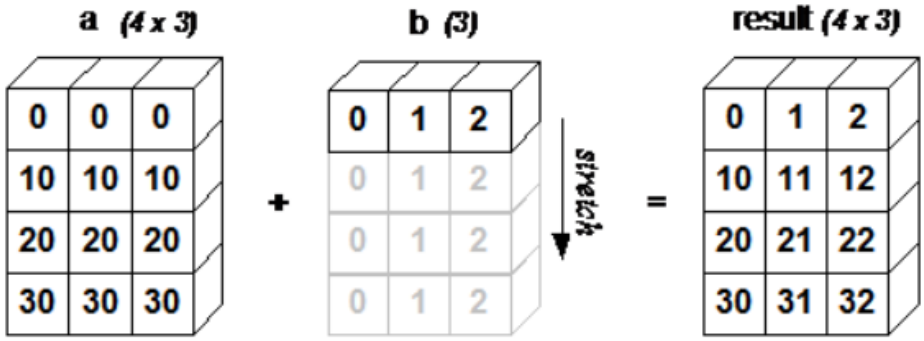
Out[96]:
```
array([2., 4., 6.])
```

The result is equivalent to the previous code where b was an array. We can think of the scalar b being stretched during the arithmetic operation into an array with the same shape as a. The new elements in b, are simply copies of the original scalar. The stretching analogy is only conceptual. numpy is smart enough to use the original scalar value without actually making copies so that broadcasting operations are as memory and computationally efficient as possible. Because above code moves less memory, (b is a scalar, not an array) around during the multiplication.



In [97]:
```
from numpy import array
a = array([[ 0.0,  0.0,  0.0],
           [10.0, 10.0, 10.0],
           [20.0, 20.0, 20.0],
           [30.0, 30.0, 30.0]])
b = array([1.0, 2.0, 3.0])
a + b
```

Out[97]:
```
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])
```



Limitation of broadcasting

Numpy broadcasting has a strict set of rules to make the operation on arrays consistent and fail-safe. These are two general rules of broadcasting in numpy:

- When we perform an operation on NumPy arrays, NumPy compares the shape of the array element-wise from right to left. Two dimensions are compatible only when they are equal or one of them is 1. If two dimensions are equal, the array is left intact. If the dimension is one, the array is broadcasted along that dimension. If none of the two conditions is satisfied, NumPy throws a ValueError, indicating the array cannot be broadcasted. The arrays are broadcasted if and only if all dimensions are compatible.
- The compared arrays need not have the same number of dimensions. The array having a smaller number of dimensions can be easily scaled along the missing dimension.

In [98]:
```
a = np.arange(12).reshape(4, 3)
print("Shape of a is:", a.shape)

b = np.arange(4).reshape(4, 1)
print("Shape of b is:", b.shape)

print("Sum: \n", a + b)
```

```
Shape of a is: (4, 3)
Shape of b is: (4, 1)
Sum:
 [[ 0  1  2]
 [ 4  5  6]
 [ 8  9 10]
 [12 13 14]]
```

Code Explanation: Sum of arrays having compatible dimensions: The arrays have dimension (4, 3) and (4, 1) which are compatible. The array b is stretched along the 2nd dimension to match the dimension of a.

In [99]:
```
a = np.arange(16).reshape(4, 4)
print("Shape of a is:", a.shape)

b = np.arange(4).reshape(4, 2)
print("Shape of b is:", b.shape)

print("Sum: \n", a + b)
```

```
Shape of a is: (4, 4)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last
t)
<ipython-input-99-5b64ccef287d> in <module>
      2 print("Shape of a is:", a.shape)
      3
----> 4 b = np.arange(4).reshape(4, 2)
      5 print("Shape of b is:", b.shape)
      6

ValueError: cannot reshape array of size 4 into shape (4,2)
```

Code Explanation: The dimensions are (4, 4) and (4, 2). The broadcasting fails because the broadcasted dimension has to be initially 1.

Explore via coding:

Given a 2D list, create a numpy 2D array using it.
Note: Given 2D list is [[1,2,3],[4,5,6],[7,8,9]].

Print the numpy array

In [100…
```
import numpy as np
l=[[1,2,3],[4,5,6],[7,8,9]]

arr=np.array(l)
print(arr)
```

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

Create an integer array of size 10. Where all the values should be 0 but the fifth value should be 1.
Print the elements of array.

In [101…
```
import numpy as np

arr=np.zeros(10)
arr[4]=1
print(arr)
```

```
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
```

Create an array with values ranging consecutively from 9 to 49(both inclusive).
Print the numpy array.

In [102…
```
import numpy as np

arr=np.arange(9,50,2)
print(arr)
```

```
[ 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49]
```

You are given a rope of 5m length. Cut the rope into 9 parts such that each part is of equal length.
Note array elements are the points where cut is to be made and round upto 2 decimal place.
Print the array element.

In [103…
```
import numpy as np

arr=np.round(np.linspace(1,5,9),decimals=2)
print(arr)
```

```
[1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
```

Find indices of non zero elements from the array [1,2,0,0,4,0]
Print the index of non zero elements.

In [104…
```
import numpy as np

l=[1,2,0,0,4,0]

arr=np.array(l)
print(arr)

print((np.where(arr!=0)))
```

```
[1 2 0 0 4 0]
(array([0, 1, 4], dtype=int64),)
```

Given an integer array of size 10. Print the index of elements which are multiple of

3.

Note: Generate the following array: array[1,3,5,7,9,11,13,15,17,19]

In [105…
```
import numpy as np

arr=np.array([1,3,5,7,9,11,13,15,17,19])

print(np.where(arr%3==0))
```

```
(array([1, 4, 7], dtype=int64),)
```

Given an integer array of size 10. Replace the odd number in numpy array with -1.

Note: Generate the following array: array[1,2,3,4,5,6,7,8,9,10]

In [106…
```
import numpy as np
filter_arr=[]
arr=np.array([1,2,3,4,5,6,7,8,9,10])

for element in arr:
    if element%2==0:
        filter_arr.append(element)
    else:
        filter_arr.append(-1)

print(filter_arr)
```

```
[-1, 2, -1, 4, -1, 6, -1, 8, -1, 10]
```

Given a 1D array, negate all elements which are between 3 and 8(both inclusive)?

NOTE: Generate the following array [1,2,3,4,5,6,7,8,9,10]

In [107…
```
import numpy as np

arr=np.arange(1,11)
filter_arr=[]

for element in arr:
    if(element>2 and element<9):
        filter_arr.append(element*-1)
    else:
        filter_arr.append(element)

print(filter_arr)
```

```
[1, 2, -3, -4, -5, -6, -7, -8, 9, 10]
```

Given age and height of 10 students in two different numpy array with name age and height in cms.Print the age of those students whose height is above 155cm. Print the numpy array.

In [108…
```
import numpy as np

age=[18,15,16,71,54,65,12,23,47,86]
height=[140,148,160,189,145,169,145,200,210,190]

arr_age=np.array(age)
arr_height=np.array(height)

filter_arr=[]

for height in arr_height:
    if(height>155):
        filter_arr.append(True)
```

```
    else:
        filter_arr.append(False)

print(arr_age[filter_arr])
```

```
[16 71 65 23 47 86]
```