NUMPY LIBRARY

- **Numpy** is the fundamental package for scientific computing in python

Basics you go through this to become Advance in Numpy :-

1. In order to start learning Numpy, you should be familiar with ARRAY and ARRAY RELATED operations like Indexing.
2. It will be beneficial to you if you go through basics of Matrix before Learning NUMPY.

   Matrix Addition, Matrix Multiplication, Array indexing, loading data in NUMPY are some of the basics steps to learn NUMPY

3. Once you are done with these basics you can go for ARRAY COMPARISIONS , LOGICAL OPERATIONS Shape and statistical functions like MEAN, MEDIAN and STANDARD DEVIATION.

4. After completion this stuff , you are free to learn advance concepts like Flattening, Reshaping, and Shuffling.

1. What is NUMPY?

NUMPY is a package in python used for scientific computations using which you can perform a variety of operations like:

1. Operations on Arrays.
2. Random number generation.
3. Shape manipulation.
4. Linear algebra operations

- You should have a decent knowledge of PYTHON before getting started with Numpy.
- Focus on LISTS and TUPLES. These are the two most important DATA TYPES that will be using in NUMPY ARRAYS
- Make sure you understand LIST COMPREHENSION, SLICING and INDEXING.
- Trust me it will help you a lot while working with 3 or higher Dimensional Arrays.

BEGGINERS GUIDE TO NUMPY ARRAYS

1. WHAT IS NUMPY?

- NUMPY is a open source fundamental library for DATA SCIENCE with PYTHON.
- Its stands for Numerical python
- NUMPY was developed by Travis Oliphant in 2005.
- It's a fast and powerful library for working with Multidimensional Arrays and Matrices.
- It provides a large number of functions to work with those Arrays.

2. WORKING WITH NUMPY ARRAY?

- Arrays are the arrangement of data in Tabular form (in the form of rows and columns)

SYNTAX:-

```
import numpy as np
```

- The most common function of the NUMPY package is ARRAY.
- ARRAY takes a number of parameters as an INPUT.

EX:-

```
np.array(object, dtype = None, copy = True, order = None, subok = False,
```

OBJECT : it is the sequence you want to pass into an Array .

Dtype: the data type of the Resultant Array.

Copy: - By default it is true. It returns an array copy of the given object.

Order :- C (row-major) or f(column-major)

Subok:- it is used to make a subclass of the base array.

Ndmin:- specifies the minimum dimensions of the final array.

- If we want to use ARRAYS ,you can use it like

```
>>>import numpy as np
>>>np.array([1,2,3])
array([1,2,3])
```

OUTPUT:-

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

NUMPY BASICS:-

Operator	Description
np.array([1,2,3])	1d array
np.array([(1,2,3),(4,5,6)])	2d array
np.arange(start,stop,step)	range array

PLACE HOLDERS:-

Operator	Description
np.linspace(0,2,9)	Add evenly spaced values btw interval to array of length
np.zeros((1,2))	Create and array filled with zeros
np.ones((1,2))	Creates an array filled with ones
np.random.random((5,5))	Creates random array
np.empty((2,2))	Creates an empty array

ARRAY:-

Syntax	Description
array.shape	Dimensions (Rows,Columns)
len(array)	Length of Array
array.ndim	Number of Array Dimensions
array.dtype	Data Type
array.astype(type)	Converts to Data Type
type(array)	Type of Array

COPYING/SORTING:-

Operators	Description
np.copy(array)	Creates copy of array
other = array.copy()	Creates deep copy of array
array.sort()	Sorts an array
array.sort(axis=0)	Sorts axis of array

ARRAY MANIPULATION:-

Adding or Removing Elements

Operator	Description
np.append(a,b)	Append items to array
np.insert(array, 1, 2, axis)	Insert items into array at axis 0 or 1
np.resize((2,4))	Resize array to shape(2,4)
np.delete(array,1,axis)	Deletes items from array

Combining Arrays:-

Operator	Description
np.concatenate((a,b),axis=0)	Concatenates 2 arrays, adds to end
np.vstack((a,b))	Stack array row-wise
np.hstack((a,b))	Stack array column wise

Splitting Arrays:-

Operator	Description
numpy.split()	Split an array into multiple sub-arrays.
np.array_split(array, 3)	Split an array in sub-arrays of (nearly) identical size
numpy.hsplit(array, 3)	Split the array horizontally at 3rd index

More other operators:-

Operator	Description
other = ndarray.flatten()	Flattens a 2d array to 1d
array = np.transpose(other) array.T	Transpose array
inverse = np.linalg.inv(matrix)	Inverse of a given matrix

Mathematics:-

Operations

Operator	Description
np.add(x,y) x + y	Addition
np.substract(x,y) x - y	Subtraction
np.divide(x,y) x / y	Division
np.multiply(x,y) x @ y	Multiplication
np.sqrt(x)	Square Root
np.sin(x)	Element-wise sine
np.cos(x)	Element-wise cosine
np.log(x)	Element-wise natural log
np.dot(x,y)	Dot product
np.roots([1,0,-4])	Roots of a given polynomial coefficients

Comparison:-

Operator	Description
==	Equal
!=	Not equal
<	Smaller than
>	Greater than
<=	Smaller than or equal
>=	Greater than or equal
np.array_equal(x,y)	Array-wise comparison

Basic statistics:-

Operator	Description
np.mean(array)	Mean
np.median(array)	Median
array.corrcoef()	Correlation Coefficient
np.std(array)	Standard Deviation

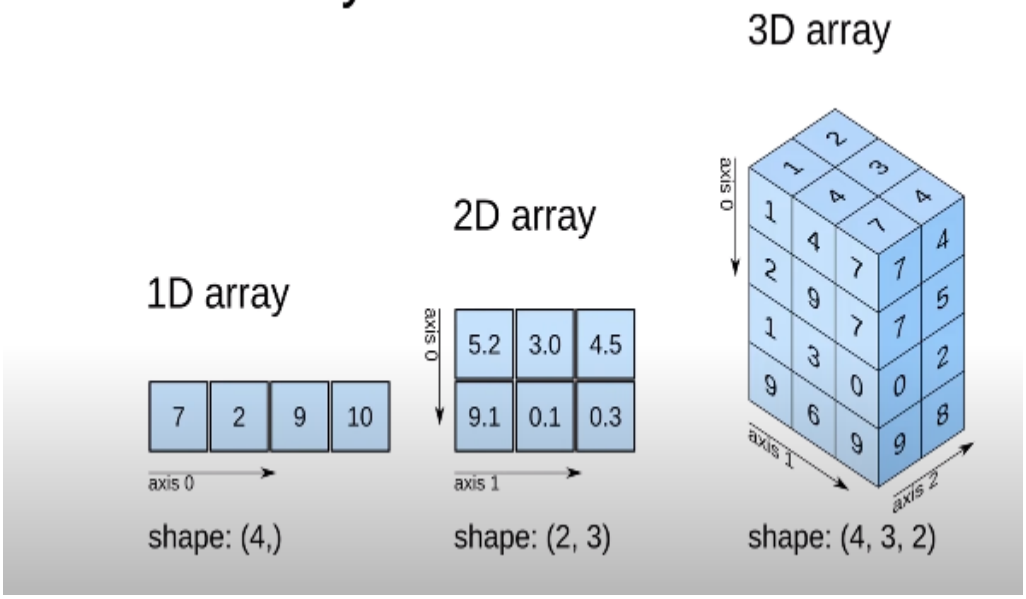
More operators:

Operator	Description
array.sum()	Array-wise sum
array.min()	Array-wise minimum value
array.max(axis=0)	Maximum value of specified axis
array.cumsum(axis=0)	Cumulative sum of specified axis

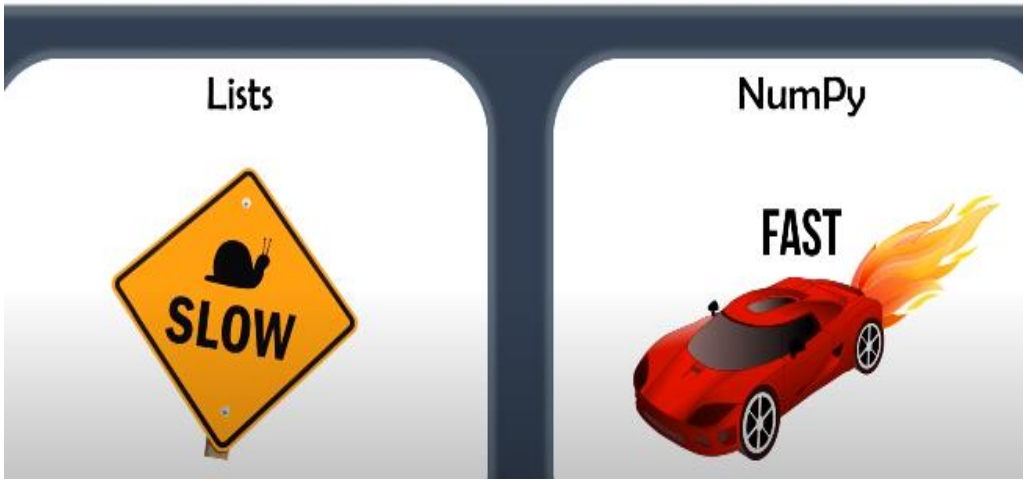
SLICING and SUBSETTING:-

Operator	Description
array[i]	1d array at index i
array[i,j]	2d array at index[i][j]
array[i<4]	Boolean Indexing, see Tricks
array[0:3]	Select items of index 0, 1 and 2
array[0:2,1]	Select items of rows 0 and 1 at column 1
array[:1]	Select items of row 0 (equals array[0:1, :])
array[1:2, :]	Select items of row 1
[comment]: <> (	array[1,...]
array[: :-1]	Reverses array

What is NumPy?



How are Lists different from Numpy?



Why is NumPy Faster? - Fixed Type

3	1	2	4
5	7	1	2
4	1	0	1

Why is NumPy Faster? - Fixed Type

Int32

00000000 00000000 00000000 00000101

Size

Reference Count

Object Type

Object Value

Why is NumPy Faster? - Fixed Type

Int32

00000000 00000000 00000000 00000101

00000000 00000000 00000000 00011100

00000001 00111101 11111110 10111100 00011010 11011101 10100100 11011000

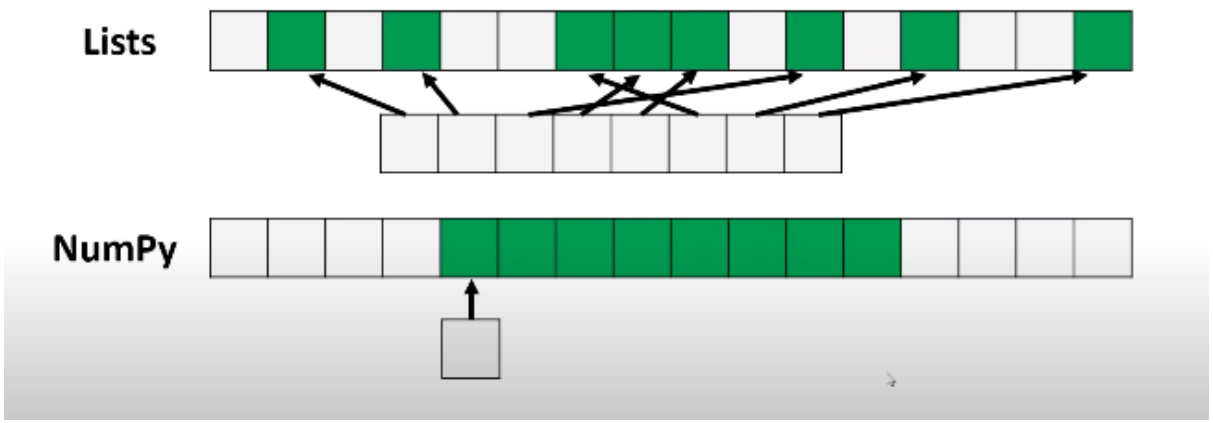
11001010 10111110 01100001 01000100 11111100 00000000 11001100 01011111

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000101

Why is NumPy Faster? - Fixed Type

- Faster to read less bytes of memory
- No type checking when iterating through objects

Why is NumPy Faster? - Contiguous Memory



Benefits:

- SIMD Vector Processing
- Effective Cache Utilization

How are Lists different from Numpy?

Lists	NumPy
Insertion, deletion, appending, concatenation, etc.	Insertion, deletion, appending, concatenation, etc. Lots More ☺

How are Lists different from Numpy?

Lists	NumPy
a = [1,3,5] *b = [1,2,3]*	*a = np.array([1,3,5])* *b = np.array([1,2,3])*
*a*b = ERROR*	*a*b = np.array([1,6,15])*

Applications of NumPy?

Mathematics (MATLAB Replacement)

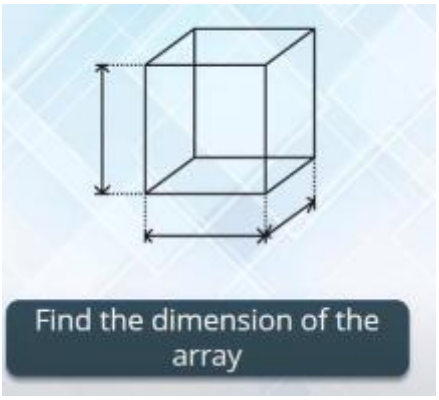
Plotting (Matplotlib)

Backend (Pandas, Connect 4, Digital Photography)

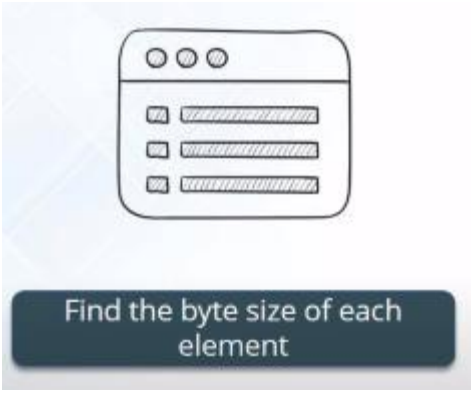
Machine Learning

WHAT IS MULTI DIMENSIONAL ARRAY??

Numpy Operations

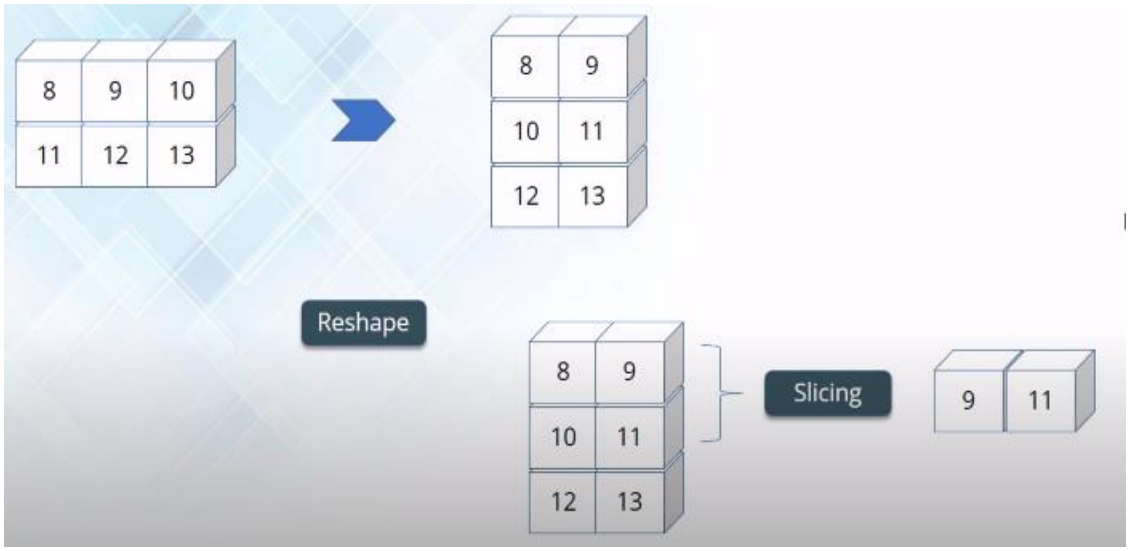


Find the dimension of the array

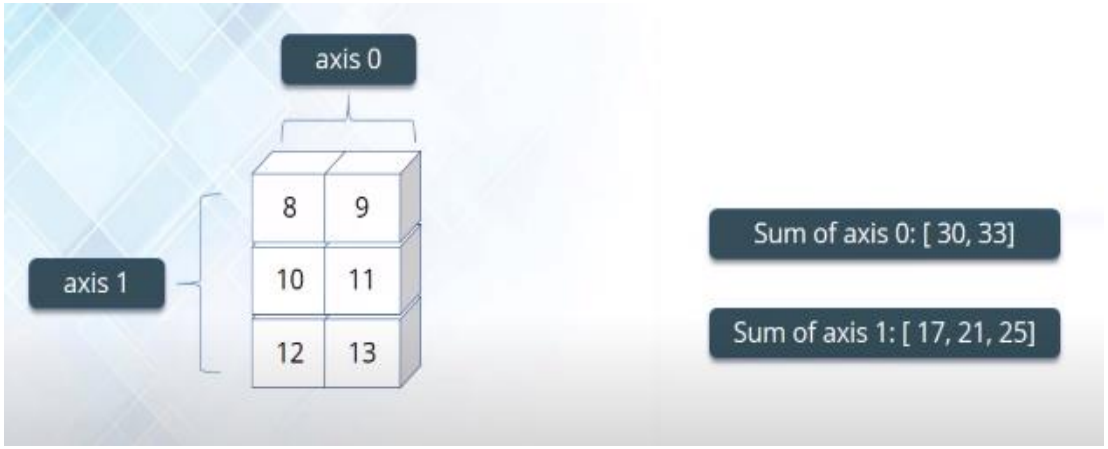


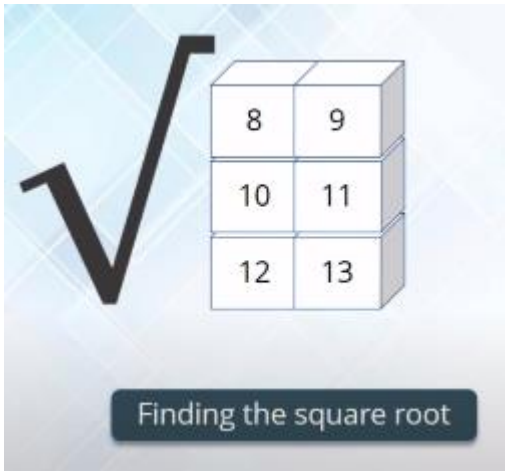
Find the byte size of each element

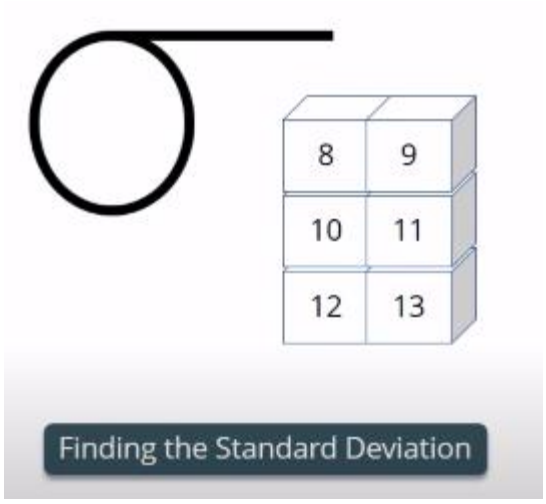
Find the data type of the elements

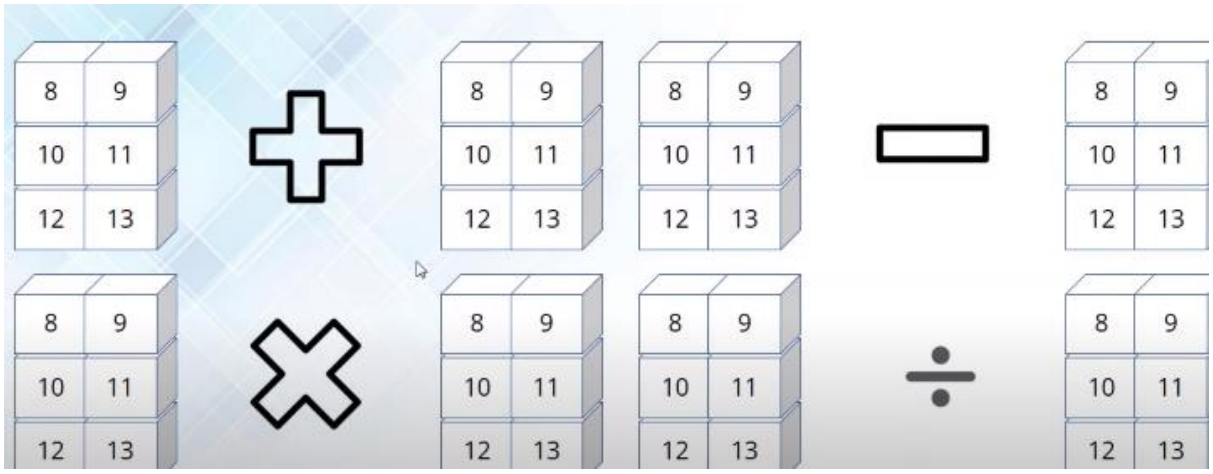

8 9 10
11 12 13
8 9
10 11
12 13
Reshape
8 9
10 11
12 13
Slicing
9 11


1 2 3 4
1
Min
4
Max
10
Sum

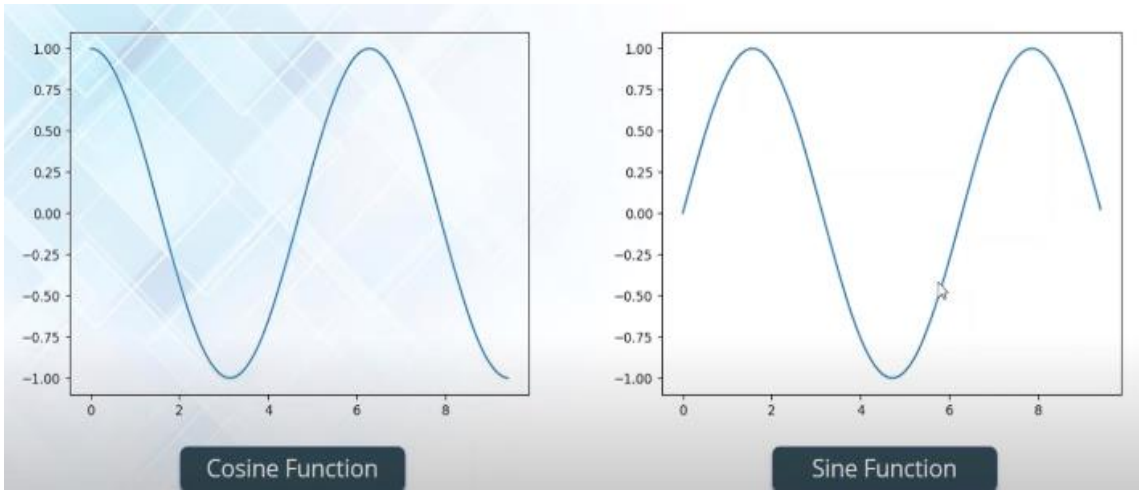

axis 0
axis 1
8 9
10 11
12 13
Sum of axis 0: [30, 33]
Sum of axis 1: [17, 21, 25]


8 9
10 11
12 13
Finding the square root


8 9
10 11
12 13
Finding the Standard Deviation


8 9
10 11
12 13
8 9
10 11
12 13
8 9
10 11
12 13
8 9
10 11
12 13
8 9
10 11
12 13
8 9
10 11
12 13
8 9
10 11
12 13
8 9
10 11
12 13

Numpy Special Functions



CREATING NUMPY ARRAYS:

- They are many functions that are specifically used to create ARRAYS

1. np.array():-

   This is the standard function to create array in numpy.
   You pass a list or tuple as an object and the array is ready.

2. np.arange():-

   It is similar to the range() function of python.

   It runs through particular values one by one and appends to make an array.

Ex :
```
np.arange(start,end,stride)
```

start: the starting number is optional to enter as it is 0 by default.
end: this is the ending number to which an array will run. Remember that array will run till **end-1** element.
stride: it is the number of steps you want to skip

Ex1:-
```
>>>np.arange(1, 10, 3)
```

Output:
```
array([1, 4, 7])
```

Explanation: - In the above example, the starting position is 1, ending is 10 and the Stride is 3. Therefore, it will run till 9 and prints every third element.

3. np.zeros():-

   This helps to create a quick array of zeros of specified order

Ex:
```
>>>np.zeros((3,3))
```

Output:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Explanation: - This is used if you want to create the array to be used later for number Storing purposes.

4. np.ones():-

It is same as np.zeros(), it just replaces zeros one ones

Ex:-

```
>>>np.ones((3,3))
```

Output:-

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

5. np.empty():-

It creates an array of garbage content. Its values are random.

Ex:-

```
>>>np.empty((2,3))
```

Output:-

```
array([[0.65670626, 0.52097334, 0.99831087],
       [0.07280136, 0.4416958 , 0.06185705]])
```

Explanation: - This is only used because it is faster than np.zeros and np.ones. This is due to the reason that all the values are random and not specified.

6. np.linspace():-

The linspace() function returns an array of evenly spaced numbers.

Ex:-

```
>>>np.linspace(3,9,3)
```

Output:

```
array([3., 6., 9.])
```

Explanation:- the resultant array contains 3,6 and 9. This is because we made the starting point as 3 and end point as 9 and we want 3 evenly spaced numbers between them. This is simple math, what are 3 evenly spaced numbers between 3 and 9(including both)? They are 3,6 and 9.

NUMPY ARRAY ATTRIBUTES:-

Numpy arrays have various attributes that can make Working with them easier. They help in organizing data in Fast and convenient ways.

1. np.array().shape and np.array().reshape():-

These attributes helps to determine the order of the Array and allow changes in them

Ex:

```
>>>import numpy as np
>>>cg = np.array([[1,2,3],[1,2,3]])
>>>cg.shape
```

Output:-

```
(2,3)
```

Explanation: - np.array.shape returns the tuple of the order of the array.

➢ You can change its order as

Ex1 :

```
>>>cg.shape = (3,2)
>>>cg
```

Output:-

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

Explanation:-

Notice that the order of the array changed from **2 rows and 3 columns** to **3 rows and 2 columns.**

➔ You can also do the same thing using the reshape function

Ex:-

```
>>>cg = np.array([[1,2,3],[1,2,3]])
>>>cg.reshape(3,2)
>>>cg
```

Output :-

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

Note: We do not use parenthesis with .shape but we do with .reshape(). This is because **.shape** is an attribute of the array while **.reshape()** is a function of array.

Below is another example of reshape() function:-

Ex:-

```
>>> np.arange(30).reshape(5,6)
```

Output :-

```
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]])
```

➔ You can also create a 3 dimensional array or greater?

Ex:-

```
>>>np.arange(30).reshape(5,2,3)
```

Output :-

```
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]]])
```

Tip:

Make sure that the multiplication of the order of an array is equal to the number of elements. Else it will not work. For example, in our previous example, we have 30 elements and the order we defined was **(5,2,3)** i.e, **5*2*3 = 30**.

3. ndim:-

This shows the dimension of the data in the array.

Ex:-

```
>>>a = np.array([[1,2,3],[1,2,3]])
>>>a.ndim
```

Output:-

```
2
```

4. itemsize:-

This attribute tells the size of the data type of the data stored in the array.

Ex:-

```
>>>a = np.array([[1,2,3],[1,2,3]])
>>>a.itemsize
```

Output:- `4`

INDEXING AND SLICING ARRAYS:-

- Array Slicing is no different than list or string slicing in particular. So, if you are familiar with slicing

1. 1D Arrays:-

Working with 1D Array is very simple.
you just have to select the index number and you are done.

Ex:-

```
cg = np.arange(1,6)
print(cg)
```

Output:- This creates an array of 5 elements from 0 to 4

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

- Now to select an element from it just type cg[element-index]

Ex:-1 if you want to select element 3 from it then all you have to do is

```
cg[2]
```

The output will be 3. This is because index starts from 0.

Ex:- 2 Suppose you want to select every element starting from 2 in this array

```
print(cg[1:])
```

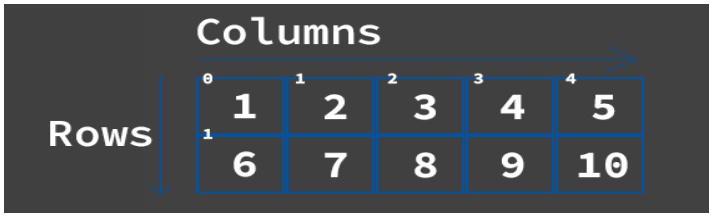
Output:- `[2 3 4 5]`

2. 2D ARRAYS :-

- We have created a two dimension array.

Ex:-
```
a = np.array([[1,2,3,4,5],[6,7,8,9,10]])
```

➔ Look at the image below for better understanding of how it works



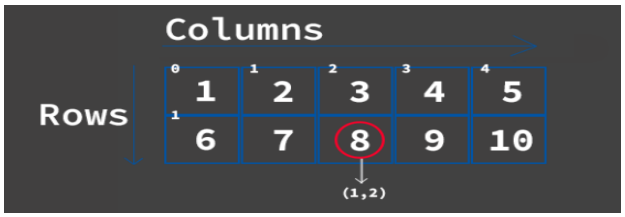
➔ Indexing in **2D array is slightly different than 1D Array**. It is written like this

```
cg = np.array([[1,2,3,4,5],[6,7,8,9,10]])
cg[row,column]
```

➔ So you first have to select the row of the element and then you select its column.

➔ Suppose if you want to select **8** from the array then you first select its row which is **1** **since** it is in the second. Row and column which is **2** since it is third column.

```
cg = np.arange(1,11).reshape(2,5)
print(cg[1,2])
```



➔ Now if you want to print all the elements of the first row then you can do this by

Ex:-
```
a = np.arange(1,11).reshape(2,5)
print(a[0,:]
```

Output:-

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

EXPLANATION:-

This is because you selected 1st row which is at
Index **0** and **empty: empty** which represents that
You select from **0** to **end** with stride **1.** These are default values.

These are the basics of slicing and indexing. There's a lot to slicing but its all practice. You will only understand slicing when you experiment with it.

3. JOINING ARRAYS :-

- You can also join two or more arrays into a single new array.
  - Let us consider two arrays

    Ex:-

```
a1 = np.array([1,2,3])
a2 = np.array([4,5,6])
```

  - In NumPy there's a function named **concatenate()** which allows us to join the arrays both horizontally and vertically.
  - Though it must satisfy the condition, It takes **3 parameters**

```
np.concatenate((sequence),axis,out)
```

Explanation :-

Sequence: The list or tuple of arrays that you want to concatenate.

Axis(Optional): How do you want to join? Along rows or columns?
By default, the Axis is 0 which is rows. To join along with columns,
You can change it to Axis=1.

Out(Optional): If provided, the destination to place the result. The shape must
Be correct **matching** that of what concatenate would have
Returned if no out argument were specified.

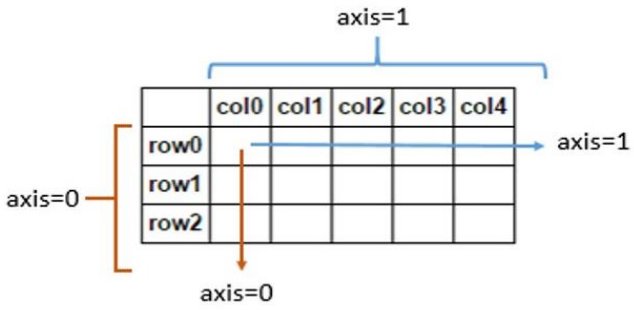
Ex:-

```
print(np.concatenate((a1,a2),axis=0))
```

Output:-

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

Below is an image which can help you visualize axis



➔ Since it is a 1D Array, you can only join it in one way. If we create a 2D Array, then you will be able to join it along with both rows and columns provided it has the same number of rows or columns.

> JOINING along axis=0

- You cannot join an array which has 1 row with an array which has 2 row
-

Ex:-

```
a1 = np.arange(1,11).reshape(2,5)
a2 = np.arange(11,21).reshape(2,5)
np.concatenate((a1,a2))
```

Output:-

```
[[ 1,  2,  3,  4,  5],
 [ 6,  7,  8,  9, 10],
 [11, 12, 13, 14, 15],
 [16, 17, 18, 19, 20]]
```

> JOINING along axis=1

- Further, if you want to join them along with columns then you can change the axis to 1. Since both arrays have the same number of rows, then you can join them along columns

Ex:-

```
print(np.concatenate((a1,a2),axis=1))
```

Output:-

```
[[ 1  2  3  4  5 11 12 13 14 15]
 [ 6  7  8  9 10 16 17 18 19 20]]
```

Explanation:- In the example, we took only 2 arrays and concatenated them but you can join as Many arrays as you want. If you are starting out as a beginner, it will be more than Enough.

4. SPLITTING ARRAYS

- You can also split an array into two or more arrays and store them differently.
- There are several functions for splitting arrays too.
- The most common of them is the **split ()** function.

```
split(array, indices_or_sections, axis=0)
```

Array: The array you want to split.
Indices or Sections: Index numbers from which you want to split the array or the Number of sections you want your array to split. I would Recommend using sections unless it is necessary to use indices.
Axis: By which axis you'd want to split (by default it is 0).

➔ In the case of a 1D array, you don't really have a choice as to how you want to split it. You can only choose from which element you want to split.
Let us consider a 1D array

➔ Now we have an array of 4 elements. If we want to split it into two then

Ex:-
```
a = np.arange(4)
```

Output:-
```
[array([0, 1]), array([2, 3])]
```

Explanation: - But this function isn't really much helpful when you are not sure About how much elements you have in array. Because it splits the elements evenly into new arrays.

- **NOTE:**- For example, if you try to split this array into 3 parts then it will throw an error.

- To **prevent** that, we have another function named **array_split()**. Just replace **split()** with **array_split()** and it will work fine.

- EX:
```
b = np.array_split(a,3)
print(b)
```

Output:-
```
[array([0, 1]), array([2]), array([3])]
```

- You can also store each array into a new variable

Ex:-

```
c = b[0]
d = b[1]
e = b[2]
print(c)
print(d)
print(e)
```

Output:-

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

Explanation:- **Split()** is only used because it is a little faster in comparison to **array_split()**

>> But it doesn't make much of a difference in time.

>> I would recommend using **array_split()** as it reduces the chances of errors.

1. SPLITTING along axis=0

- Let us consider a two dimensional array. We will be splitting this array along axis 0 i.e., splitting along rows.

Ex:-

```
a1 = np.arange(1,13).reshape(2,6)
b = np.array_split(a1,2)
```

Output:-

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

2. SPLITTING along axis=1

- If you want to split them along with columns then you can change the axis to 1.

Ex:-

```
b = np.array_split(a1,2,axis=1)
print(b)
```

Output:-

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

COMPARISION: - Arrays, Lists, Tuples

- **Vectorized Operations:** One of the main differences between Arrays, Lists, and Tuples is vectorized operations.

- Only Arrays allow vectorized operations i.e. when you apply a function it gets applied to each element of an array and not to array itself.

- Ex:-

```
>>import numpy as np
>>cg = [1,2,3,4]
>>cg_array = np.array(cg)
>>cg_array += 2
>>cg_array
>>array([3, 4, 5, 6])
```

- If you try the same with list or tuple it will throw an error.

- Ex:-

```
>>>cgt = (1,2,3,4)
>>>cgt += 1
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    cgt += 1
TypeError: can only concatenate tuple (not "int") to tuple
```

- **Data Type Declaration:** Arrays need to be declared while list, tuples, and dictionaries, etc. do not need to be declared i.e., if you want to use arrays then you have to declare them using the **array()** class while you do not have to do it with lists or tuples.

- **Mutability:** Mutability means the ability to be changed. Data inside arrays can be changed while the data in a tuple cannot be changed or modified.

- **Heterogeneous Data**: While arrays, lists, and tuples all are used to store data, **arrays cannot store heterogeneous data**.

- **Ex:-**

```
>>> from numpy import array
>>> a = [1,2.0,True,"string"]
>>> b = array(a)
>>> a
[1, 2.0, True, 'string']
>>> b
array(['1', '2.0', 'True', 'string'], dtype='<U32')
```

Explanation:- If you observe carefully, the list I passed in array has each element of different data type but when printed as an array, it converted all the data as **"string"**.

- This is not the case with lists or tuples.

	Array	List	Tuple
Vectorized Operations	Yes	No	No
Mutability	Yes	Yes	No
Pre-Defined Data Type	No	Yes	Yes
Heterogeneous Data	No	Yes	Yes

WHEN AND WHEN not to use ARRAYS?

When to use Arrays: -

- You have to store a large amount of data.
- The data you want to store is of the same type.
- You may perform operations on each element.

When not to use Arrays: -

- The Data Type is different
- The data is very small
- You do not have to perform operations on each element.

- **THE above all MOST IMPORTANT of these FUNCTIONS Which are Necessity for using NUMPY**