Numpy: Numpy is a python package which stands for 'Numerical python'. It is the core library for scientific computing. which contains a powerful n-dimensional array object.

* This is the most important library for Datascience.
* Datascience has a concept called as array. But python does not have a data type called Array, so using numpy library we can create an array.
* Numpy is mainly used to create an n-dimensional array object.
* Numpy has lots of tools (methods or functions) by using which we can manipulate this n-dimensional array object.

How do we install Numpy:

To install python Numpy we do

```
pip install numpy
```

Once the install is done we need to import the numpy module. Usually we use alias 'np'.

```
import numpy as np
```

Array: It is an object that is a collection of similar type of data (same data type)

Array - (Homogenous data)
(same Data type (SDT))

Lists, tuples, set, dictionary - (Heterogenous data)
(Different Data type (DDT))

* Array is an object of array class

Numpy Array: It is a powerful n-dimensional array object which is in the form of rows and columns.

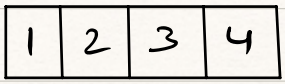
n-dimensional - Here n stands for rank

* In python max n is 32 so 0-32 we can use. so max we can create 32-dimensional array.

n=0 (scalar value) i.e (no dimension)

ex:- 1, 2, 3, ... scalar values

n=1 (vector) i.e (single dimension)

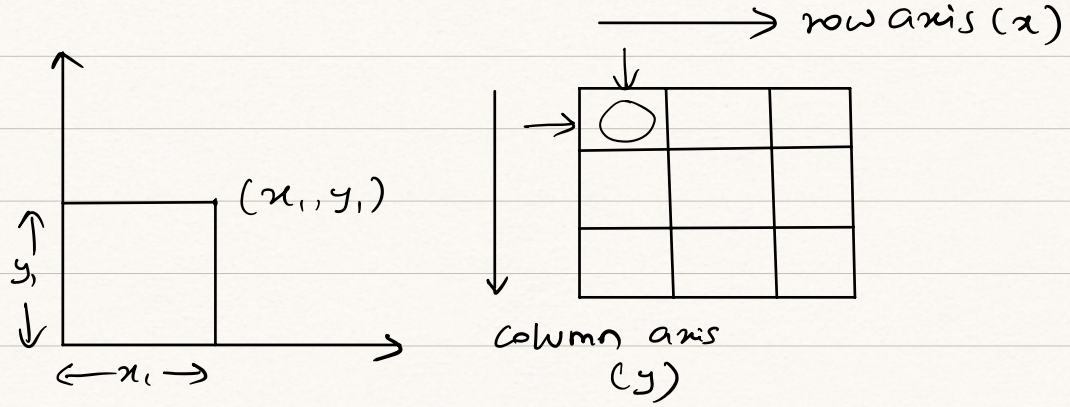


It can be column vector or row vector

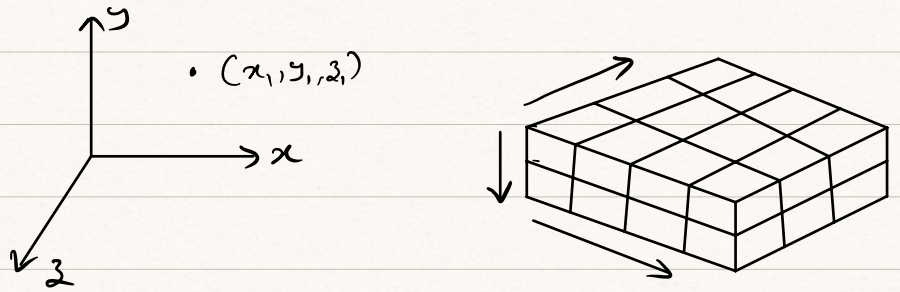
* By default it is a column vector. we can use transpose to convert it to row vector.

Transpose – converts rows to columns
columns to rows

<u>n=2</u> (matrices) i.e (two dimensional)



* Human being can visualize upto 3-dimensions.
Above 3-dimensions we take help of linear Algebra.



n=3 or n>3 (Tensors) ie (3D to n-dimensional array)

0 Dimension - Scalar

1 Dimension - Vector

2 Dimension - Matrices

3+ Dimension - Tensors

* Numpy can create upto 32 Dimension array and has lots of tools to manipulate them.

Properties of Array:

1) Array contains homogenous data.
2) Arrays are mutable.
3) Inside an array each element will have an index value.
4) Arrays are ordered
5) Indexing and slicing is possible.
6) Duplicates are allowed.

Creating a n-dimensional array:

```
Pip install numpy
import numpy as np
```

1) Default function for creating an array is array.

```
np.array (data or object, dtype, ndmin)
          ↓ mandatory      ↘ ↙ optional
```

ex:- `np.array (5)` → 5 (array of 0 Dimension)

`np.array ([5,4,3])` → array([5,4,3]) (array of 1-Dimension)

For creating 2-Dimensional array we use list inside a list.

`np.array ([[1,2,3],[4,5,6]])` — matrices

(rows)

* Here the inner lists are rows.

1	2	3
4	5	6

For creating n-dimensional array we use n lists inside each other.

[[[[[[.....[ ].....]]]]]]

* Or we can use ndim to mention the rank based on this it will create dimensions without the lists inside the lists notation.

So ndim = 3 creates 3-dimensional array.

ex:-

x = np.array([6,7,8,9], ndim = 3)

↳ array([[[6,7,8,9]]])

dtype (Datatype): In the creation process of array we can mention the dtype which is the datatype of the elements inside the array.

* Since array is same data type all the elements are converted to data type mentioned and array is created.
* default data type is float.
* we can use python data types int, float, complex, bool etc.
* Numpy has large set of numeric data types that can be used to create arrays. These datatypes help in allocating memory based on

the programmers need. so we can save memory for huge data.

* If we want to use methods like insert, extend, pop so on we can import another module called 'array'.
* In numpy datatypes of array need not be defined unless a specific data type is required.

Data type	Description
bool_	boolean (True or False) stored as byte
int8	Byte (-128 to 127)
int16	integer (-32768 to 32767)
int32	integer (-2.15E-9 to 2.15E+9)
int64	integer (-9.22E-18 to 9.22E+18)
unit8	unsigned integer (0 to 255)
unit16	unsigned integer (0 to 65535)
unit32	unsigned integer (0 to 4.29E+9)
unit64	unsigned integer (0 to 1.84E+19)
float16	Half precision signed float
float32	Single precision signed float
float64	double precision signed float
Complex64	Complex number: two 32-bit floats (real and imaginary components)
Complex128	Complex number: two 64-bit floats (real and imaginary components)

Basic operations done on arrays:

1) Creating array using array function:

ex:-

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

→

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

2) Printing array dimensions:

ndim function returns the number of dimensions of an array.

ex:-

```
x = np.array([[1,2,3],[4,5,6],[7,8,9]])
x.ndim
```

→ 2

3) Printing shape of an array:

Shape helps us to find the shape or size of an array or matrix.

* Shape is denoted in tuple format.
* The number of elements in the shape tuple denotes the number of dimensions.

ex:-

when shape is ( )

↳ 0 dimensions

when shape is (4,)

↳ 1 dimension with 4 elements

when shape is (3,4)

↳ 2 dimensions with 3 and 4 elements in each dimension respectively.

when shape is (3,4,4)

↳ 3 dimensions with 3,4,4 elements in each dimensions.

* we can get total elements by multiplying all the elements in the shape tuple.

ex:- (3,4) then total elements is 3x4=12

ex:- x = np.array([6,7,8,9])

x.shape

↳ (4,)

So array is 1-D

6	7	8	9

and total elements are 4.

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

x.shape

↳ (3,3)

So the array is 3x3 array

1	2	3
4	5	6
7	8	9

and total elements = 3x3 = 9

* Array can only have same data type so when the data of different types is given then dynamic implicit data conversion is done based on preference.

highest preference - string
next preference - complex
next preference - float
next preference - int

ex:-

```
np.array([6,7,8,9,1.3])
  ↳ array([6.0, 7.0, 9.0, 1.3])

np.array([6,7,8,9,1.3,'a'])
  ↳ array(['6','7','8','9','1.3','a'])

np.array([6,7,8,9,1+2j])
  ↳ array([6+0j, 7+0j, 8+0j, 9+0j, 1+2j])
```

* To find the data type we can use dtype.

i.e

```
x=np.array([6,7,8,1.3,1+2j,"a"])
```

x.dtype

↳ dtype('<U64') – this is complex 64

* Single dimensional array is simillar to list with same data type.

So we can do indexing and slicing.

ex:-

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

x[0]

↳ 6

x[0:3]

↳ array([6,7,8])

if we make x[0]=100

x

↳ array([100,7,8,5,6,7])

More additional functions for creating arrays:

1) Creating an array with all zeros:

function which will create n-dimensional array object where all the elements are zeros.

```
zeros ( shape, dtype = None)
```

shape will be in tuple format and by giving shape the function will know the dimension of the array.

* default dtype is Float. For other dtypes we need to mention.

ex:-

```
np.zeros ((3,3), dtype = int)
↳ array ([[0,0,0]
          [0,0,0]
          [0,0,0]])
```

→ 2-dimensional array with 3 elements in row and 3 elements in column.

2) Creating an array with all one's:

function which will create n-dimensional array object where all the elements are one's.

```
ones ( shape , dtype = None)
```

ex:-

```
np.ones ((3,3), dtype = int)
```

↳ array([[1, 1, 1]
[1, 1, 1]
[1, 1, 1]])

2 dimensional with 3 rows and 3 columns

np.ones((2,1), dtype=int)

↳ array([[1]
[1]])

2 dimensional with 2 rows and 1 column.

3) **Full** - function which will create n-dimensional array where objects are what we mention.

full(shape, num we want, dtype=None)

ex:-

np.full((3,3), 11, dtype=int)

↳ array([[11, 11, 11]
[11, 11, 11]
[11, 11, 11]])

2 dimensional with 3 rows and 3 columns of elements (11)

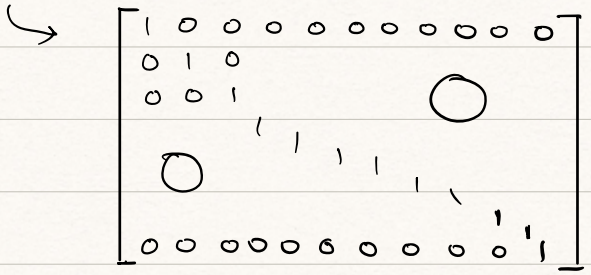
4) **eye:** function which will create n-dimensional array which is a identity matrix. (square matrix with elements 1 in the diagonal).

Since this is a square matrix shape is not needed. we just need to mention no.of rows.

eye (no.of rows, dtype = None)

ex:-

np.eye (10, dtype = int)

↳ 

5) diag (diagonal) - Function which will create n-dimensional array which has the elements passed in the diagonal. By knowing diagonal elements we already know no.of rows and columns and there by the dimension of the array. so input is just the dimensional elements.

diag (elements to be inserted, dtype = None)

ex:- np.diag ([2, 3, 4, 5])

↳ array([[2, 0, 0, 0]
[0, 3, 0, 0]
[0, 0, 4, 0]
[0, 0, 0, 5]])

i.e $\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix}$

if x = np.diag([2,3,4,5])

np.diag(x)

↳ array([2,3,4,5])

Till now we have created n-dimensional arrays.

* There are functions to create 1-dimensional array.

1) arange (array range) function that will create single dimensional array. It behaves similarly as range function.

np.arange(sv, Ev, step)

- sv → starting value
- Ev → Ending value (not included)
- step → step size

default step size is '1'.

ex:-

np.arange(1, 10, 1)

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

↳ array([1, 2, 3, 4, 5, 6, 7, 8, 9])

np.arange(1, 10, 2)

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

↳ array([1, 3, 5, 7, 9])

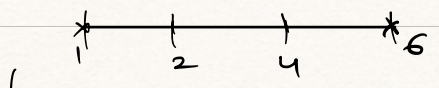
2) Linspace (linear space) – Function that will create single dimension array. It will take starting values (SV) & Ending values (EV) and between these two values how many times we want to divide.

This will include SV & EV in the elements.

np.linspace (SV, EV, how many times to divide)

ex:-

np.linspace (1, 6, 4)



↳ array ([1, 2, 4, 6])

* these numbers will be approximately equal spaced not exact.
* we need to have dtype as default float and not change otherwise will loose data.
* default no of times to divide is 50 if nothing is mentioned.

ex:-

np.linspace (1, 10, 6, dtype=int)

↳ this is wrong as dtype has to be float otherwise we will loose data.

Accessing from an array:

```
q = np.diag([10, 11, 12])
  ↳ array([[10, 0, 0]
           [0, 11, 0]
           [0, 0, 12]])

np.diag(q)
  ↳ array([10, 11, 12])
```

* using diag function we can create a diagonal array or find the array passed to create the diagonal array.

ex:-

```
w = np.eye(10, dtype=int)
np.diag(w)
  ↳ array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

Random number array creation :

Inside numpy library there is a random module and random function inside that module.

If we want to create n-dimensional array with random numbers.

We only need to give shape to the random function.

First we need to import random module

```
import random
np.random.random((2,3))
```

↳ array([[0.2..., 0.1..., 0.6...]
[0.4..., 0.5..., 0.7...]])

Main differences between Arrays & Lists :

Lists	Arrays
1) The operations are applied on the complete list ex:- [1,2,3,4] + 2 ↳ x error [1,2,3] * 2 [1,2,3,1,2,3]	1) Operations are applied on individual elements of the array (i.e element wise) ex:- a=np.array([1,2,3,4]) a+2 ↳ array([2,4,5,6]) a*2 ↳ array([2,4,6,8])
2) Consumes large memory as dynamically allocated	2) more compact in memory size as we can mention the numpy data type to be used.
3) Computation is slow.	3) Fast computation as much as 100 times more than lists.
4) Single dimensional	4) n-dimensional
5) Same or different data types	5) Same data type elements.
6) Lists can be appended and size can be extended	6) Once created array size cannot be increased.
7) No need to explicitly import a module for declaration.	7) Need to explicitly declare a module for declaration.
8) Preferred for shorter sequence of data items	8) Preferred for longer sequence of data items.

ex:- To show arrays are faster than lists.

```
import time
```

Lists

```
st=time.time()
l=[]
for y in range(1,10000000):
    l.append(y*2)
et=time.time()
print(et-st)
```

↳ 2.50027...

array

```
st=time.time()
x.nparange(1,10000000)*2
et=time.time()
Print(et-st)
```

↳ 0.08001....

0.08001... is way faster than 2.50027.... seconds.

Accessing elements inside n-dimensional array:

n=0 → scalar so no accessing

n=1 → Vector, single dimension so similar to lists.

ex:-

```
x = np.arange(1, 9)
x
↳ array([1,2,3,4,5,6,7,8])
x[0]
↳ 1
x[:3]
↳ array([1,2,3])
```

n=2 → matrices, we have to access elements in 2-dimensions.

Here we will have row index & column index

	0	1	2
0	a_1	a_2	a_3
1	a_4	a_5	a_6
2	a_7	a_8	a_9

→ column index

↓ row index

i.e (row index, column index)

ex:-

```
z = np.random.random((3,3))
```

```
z[0,2]
```

↳ a_7

```
z[0,0]
```

↳ a_1

reshape ( ) function : (important function)

Reshaping arrays

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

Can we reshape in to any shape ?

Yes, as long as the elements required for reshaping are equal in both shapes.

* Size of an array cannot be changed once created.
* Shape of an array can be changed but only in the multiples of the array elements.

i.e we can reshape an array containing 8 elements in to a (2x4) i.e (2,4) array or (4,2) array.
But not (3,3) as 3x3=9 and we only have 8 elements.

```
np.arange(1,10)
```

→ 1D with 9 elements

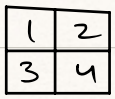
```
array([1,2,3,4,5,6,7,8,9])
```

based on the elements we can define the shape

```
shape (2,3)
```

→ 2x3=6 i.e 6 elements

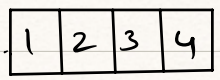
row x column = elements

if 

2x2 so shape (2,2) and 4 elements total

so shaped can be changed in multiples only

(4, 1) or (1, 4) Possible



ex:-

```
x = np.arange(1, 10)
    ↳ array([1, 2, 3, 4, 5, 6, 7, 8, 9])

x.shape
    ↳ (9, )

x.reshape(3, 3)
    ↳ array([[1, 2, 3]
             [4, 5, 6]
             [7, 8, 9]])

x.reshape(3,10)
    ↳ x error as 3x10=30, we only
      have 9 elements

x.reshape(9, 1)
    ↳ array([[1]
             ⋮
             [9]])
```

```
x.reshape(1,9)
↳ array([[1, 2, 3, ..... 9]])
```

reshape will only give the temporary result and will not update the original array.

If we want to use the reshaped array we need to store it to a new variable.

```
y = x.reshape(3,3)
y
↳ array([[1,2,3]
         [4,5,6]
         [7,8,9]])
x
↳ array([1,2,3,4,5,6,7,8,9])
```

If we want to permanently change the shape of x we can use shape function to do it.

```
x.shape = (3,3)
x
↳ array([[1,2,3]
         [4,5,6]
         [7,8,9]])
```

How to perform Arithmetic operations on array

(+, −, *, /, %, //)

* when performing operations on arrays they will be performed element wise.
* The result of these operations is temporary and if we want to use it we need to save it to a variable and use it.

operations with integers

using operator	using function
x+2	np.add(x, 2)
x-2	np.subtract(x,2)
x*2	np.multiply(x,2)
x/2	np.divide(x, 2)
x//2	np.floor_divide(x,2)
x%2	np.mod(x, 2)

Similar operations can be done with other arrays

ex:- a = ([[[1, 2]
[3, 4]
[5, 6]
[7, 8]]])

b = ([[[1, 2]
[3, 4]
[5, 6]
[7, 8]]])

a+b = ([[2,4] or np.add(a,b)
[6,8]
[10,12]
[14,16]])

* For doing operations the arrays need to be in same shape. If different shape it is advanced topic. (covered later)

Comparision operators on arrays

(>, <, ==, !=, <=, >=)

* Any operations on array is performed elementwise.
* When we use comparision operator on an array we will get a boolean array.

ex:- a = ([[1,2]
[3,4]
[5,6]
[7,8]])

a>4
↳ array([[False, False]
[False, True]
[True, True]
[True, True]])

* Arrays shape has to match when we are comparing 2 arrays.

```
x = ([[[5, 6, 7]          y = ([[[1, 2, 3]
      [8, 9, 10]                [10, 11, 12]
      [11, 12, 13]]])           [13, 14, 15]]])
```

```
x > y
↳ array([[[True, True, True]
          [True, True, True]
          [False, False, False]]])
```

We have mainly 3-types of accessing in 2-dimensional

1st type - Indexing based slicing

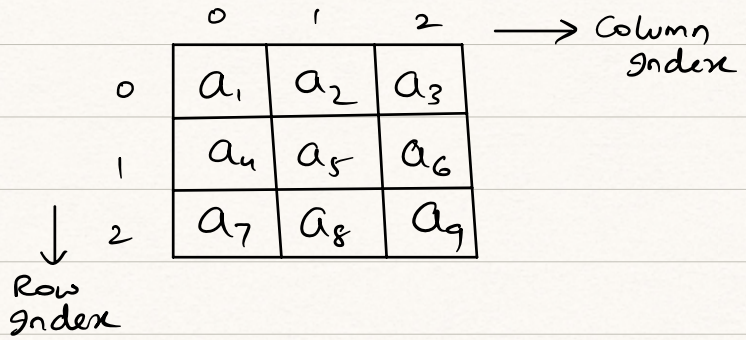
2nd type - Integer based indexing

3rd type - boolean based indexing

Indexing based slicing:

In 2D when we are slicing based on Indexing we need to provide both the row index and column index.

```
var [ SRI : ERI , SCI : ECI ]
      ↙     ↓     ↓     ↘
   Starting  Ending  Starting  Ending
   Row       Row     Column    Column
   Index     Index   Index     Index
             (not included)    (not included)
```



i.e (row index, column index)

So if $Y = \begin{matrix} & \begin{matrix}0 & 1 & 2\end{matrix} \\ \begin{matrix}0\\1\\2\end{matrix} & \begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}\end{matrix}$

```
y[0:1, 0:]
    → array([1, 2, 3])

y[1:, 0:]
    → array([[4, 5, 6]
             [7, 8, 9]])
```

Properties of Indexing based slicing technique

1) With the help of Indexing based slicing we cannot take out arbitrary values. (rows & columns). They have to be sequential.

2) When we get subarray from the original array using Indexing based slicing technique they both will have same dimension.

   i.e `y.ndim` → 2 and `y[1:, 0:].ndim` → 2 is same.

3) When we get subarray from the original array using Indexing based slicing technique. The sub array will be a view to the original array.

   - View is when we change the subarray the changes will reflect in original array.

```
i.e  z = y[1:, 0:]
     z[0, 0]
        → 4

     z[0, 0] = 100
```

z

↳ array ([[100, 5, 6]
[7, 8, 9]])

y

↳ array ([[1, 2, 3]
[100, 5, 6]
[7, 8, 9]])

** data is changed in the original array also.

Integer based Indexing technique:

Here the slicing method like y[1:, 0:] will not be used. Instead the format will be.

var [[e1RI, e2RI, ...], [e1CI, e2CI, ...]]

- e1RI: element 1 Row index
- e2RI: element 2 Row index
- e1CI: element 1 Column index
- e2CI: element 2 Column index

ex:-

```
y = np.arange(1,10).reshape(3,3)
```

```
y
↳ array ([[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]])
```

(rows 0, 1, 2)

```
y[[2,2],[1,2]]
  ↳ array ([8,9])
```

i.e y=	0	1	2	3	4
0	a_1	a_2	a_3	a_4	a_5
1	a_6	a_7	a_8	a_9	a_{10}
2	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
3	a_{16}	a_{17}	a_{18}	a_{19}	a_{20}
4	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}

→ column index

↓ row index

```
y [[2,3],[3,2]]
  ↳ array ([a13, a17])
```

y [[2, 3, 4], [1, 2, 3]]

↳ array ([a_{12}, a_{18}, a_{24}])

Properties of Integer based Indexing technique

1) With the help of Integer based Indexing we can directly take out arbitrary values. (rows & columns). (They don't have to be sequential). As we are mentioning row index and column index of each element specifically.

2) When we get subarray from the original array using Integer based Indexing technique. Subarray will have lower rank (lower dimension) compared to original array.

3) When we get subarray from the original array using Integer based Indexing technique. The sub array will be a copy to the original array.

- Copy is when we change the subarray the changes will not be reflected in the original array.

ex:-

z = y [[2, 3, 4], [1, 2, 3]]

↳ array ([a_{12}, a_{18}, a_{24}])

Single dimension so similar to lists

z.ndim

↳ 1

y.ndim

↳ 2

z[0]

↳ a_{12}

z[0] = a_{15}

z

↳ array([a_{15}, a_{18}, a_{24}])

y

↳ array([[a_1, a_2, a_3, a_4, a_5]
[a_6, a_7, a_8, a_9, a_{10}]
[a_{11}, a_{12}, a_{13}, a_{14}, a_{15}]
[a_{16}, a_{17}, a_{18}, a_{19}, a_{20}]
[a_{21}, a_{22}, a_{23}, a_{24}, a_{25}]])

* element in y is unchanged.

Boolean Indexing technique (also called masking)

* when we are using comparision operators on arrays we will get a boolean array.
* If we want to see values for whenever the boolean is returning as True in boolean array we use boolean Indexing.
* The condition we use to get boolean array is called boolean masking and it filters out False values and only gives values when True.

ex:-

```
x = ([[[1, 2, 3]
       [4, 5, 6]
       [7, 8, 9]]])
```

boolean mask x % 2 == 0 (even numbers)

↳ array ([[[False, True, False]
[True, False, True]
[False, True, False]]])

boolean Indexing

↙ It uses boolean mask

x[x % 2 == 0]

↳ array ([2, 4, 6, 8])

* If we have to apply multiple conditions regular 'and' and 'or' will not work.

* Instead we use

& - and

| - or

↳ these are not bitwise operators

(they are only and and or for arrays in numpy)

so (cond1) | (cond2)

↙ paranthesis is must.

ex:-

Get elements from x that are divisible by 5 or divisible by 7.

i.e (x%5==0) | (x%7==0)

boolean Indexing:

x[(x%5==0) | (x%7==0)]

↳ array([5, 7])

Properties of boolean Indexing:

1) we will use boolean Indexing when ever we want to filter out using conditions.

2) when we get subarray from the original array using boolean based Indexing technique. Subarray will have lower rank (lower dimension) compared to original array.

3) when we get subarray from the original array using boolean based indexing technique. The sub array will be a copy to the original array.

- Copy is when we change the subarray the changes will not be reflected in the original array.