# NUMPY RANDOM NUMBERS:

1. np.random.rand :

It generates an array with random numbers that are uniformly distributed between '0' and '1'.

2. np.random.randn :

It generates an array with random numbers that are normally distributed, mean = 0 and stdev (standard deviation) = 1.

3. np.random.randint :

It generates an array with random numbers (integers) that are uniformly distributed between '0' and given number.

4. np.random.uniform :

It generates an array with random numbers (float) between given numbers.

I/P - arr = np.random.rand(5)
Print(arr)

O/P - [0.6141691 0.64597138 0.79514445
0.3961621 0.18799783]

I/P - Print(np.random.rand(10,2))

O/P - [[0.355 0.840]
[0.961 0.038]
[0.463 0.595]
[0.894 0.578]
[0.518 0.938]
[0.458 0.027]
[0.568 0.290]
[0.392 0.732]
[0.312 0.829]
[0.253 0.337]]

I/P - print(np.random.randn(5))

O/P - [0.5022 -0.0277 1.4805 0.2721 0.4473]

I/p - print(np.random.randn(5,4))

O/p - [[0.07 0.02 0.20 1.15]
[1.17 0.27 1.41 -0.60]
[-1.94 1.30 0.75 0.06]
[1.08 -0.50 -1.25 0.43]
[-0.36 -0.36 0.75 -1.76]]

One Random Integer between '0' to '9':

I/p - print(np.random.randint(10))

O/p - 6.

I/p - Print(np.random.randint(10, size=(5,4))

O/p - [[6 0 3 3]
[8 5 8 0]
[9 2 5 2]
[9 2 4 0]
[1 6 2 8]]

I/p - Print(np.random.randint(10,40, size=(5,4)))

O/p - [[23 23 10 24]
[36 31 11 21]
[38 14 36 36]
[37 18 14 34]
[39 38 35 38]]

I/p - Print(np.random.uniform(10))

O/p - 8.2358

I/p - Print(np.random.uniform(10, size = (5,4)))

O/p - [[1.6160 2.4312 6.4811 8.4748]
[5.3871 6.0546 4.8357 9.7418]
[9.7660 5.6885 5.7278 9.8501]
[6.6645 3.3342 7.4625 7.3538]
[9.6552 3.1231 2.9380 1.8881]]

I/p - Print(np.random.uniform(10,40, size = (5,4)))

O/p - [[37.86 32.03 28.31 16.58]
[11.96 33.87 38.01 36.19]
[39.32 29.74 19.01 26.31]
[20.81 39.92 36.89 25.49]
[31.96 29.26 27.19 23.95]]

NUMPY INDEXING:

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

I/p - arr = 3 * np.arange(10) # because of broadcasting.

Print(arr)

Print(arr[2])

O/p - [0 3 6 9 12 15 18 21 24 27]

6

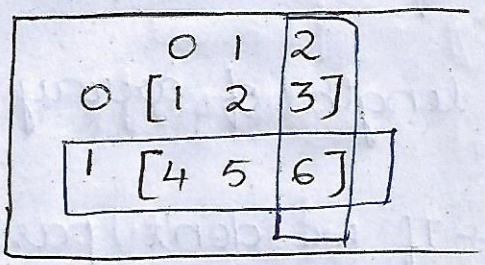
ACCESSING AN INDEX:

SYNTAX: arr[row index, column index]

I/p - arr = np.array([[1,2,3],[4,5,6]])

Print(arr[1,2])

O/p - 6




I/p - arr[1,2] = 99

Print(arr)

Replacing the index value of [1,2] i.e., 6 with 99

O/p - [[1 2 3]
[4 5 99]]

NOTE: NumPy arrays are mutable.

# NUMPY SLICING:

Slicing means taking elements from one given index to another given index.

↳ We pass slice instead of index as

[start : end]

↳ We define the step as [start : end : step]

↳ If we don't pass start it is considered as '0'.

↳ If we don't pass end it is considered as length of array in that dimension.

↳ If we don't pass step it is considered as '1'.

SYNTAX : arr[s:e:ss , s:e:ss]

s - start , e - end , ss - stepsize

```
I/P - arr = np.arange(10)
      Print(arr[: : 2])
      Print(arr[1 : 4])
      Print(arr[0 : -4])
```

o/p - [0 2 4 6 8]
[1 2 3]
[0 1 2 3 4 5]

I/p - arr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
print(arr[:2, 1:3])

o/p - [[2 3]
[6 7]]

I/p - print(arr[1:2, :])

o/p - [[5, 6, 7, 8]]

I/p - arr = np.array([[1,2],[3,4],[5,6]])
print(arr[2,1])
Print(arr[[0,1,2],[0,1,0]])

o/p - 6
[1 4 5]

| | 0 | 1 | |
|---|---|---|---|
| 0 | [1 | 2] | (0,0)=1 |
| 1 | [3 | 4], | (1,1)=4 |
| 2 | [5 | 6] | (2,0)=5 |

NUMPY MATHS : 

Element wise operations :

I/P - x = np.array([[1,2],[3,4]])

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

Print(np.add(x,y))

Print(np.subtract(x,y))

Print(np.multiply(x,y))

Print(np.divide(x,y))

O/P - [[6 8]
[10 12]]

[[-4 -4]
[-4 -4]]

[[5 12]
[21 32]]

[[0.2 0.3]
[0.4 0.5]]

## Matrix Multiplication :

SYNTAX : np.matmul (or) np.dot

I/p - print(np.matmul(x, y))
print(np.dot(x, y))

O/p - [[19 22]
[43 50]]

[[19 22]
[43 50]]

I/p - x = np.array([[1, 2], [3,4]])
print(np.sum(x))
print(np.sum(x, axis=0)) # Column wise
print(np.sum(x, axis=1)) # Row wise

O/p - 10
[4 6]
[3 7]

I/p - print(np.min(x))
print(np.max(x))

O/p - 1
4

| | 0 | 1 |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 3 | 4 |

axis = 0 → 1+3 = 4
2+4 = 6

axis = 1 → 1+2 = 3
3+4 = 7

# NUMPY MATHS :

## Element wise operations :

I/P - x = np.array([[1,2],[3,4]])

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

Print(np.add(x,y))

Print(np.subtract(x,y))

Print(np.multiply(x,y))

Print(np.divide(x,y))

O/P - [[6 8]
[10 12]]

[[-4 -4]
[-4 -4]]

[[5 12]
[21 32]]

[[0.2 0.3]
[0.4 0.5]]

## Matrix Multiplication :

SYNTAX : np.matmul (or) np.dot

I/P - print(np.matmul(x, y))
print(np.dot(x,y))

O/P - [[19 22]
[43 50]]

[[19 22]
[43 50]]

I/P - x = np.array([[1, 2], [3,4]])
Print(np.sum(x))
Print(np.sum(x, axis=0)) # Column wise
Print(np.sum(x, axis=1)) # Row wise

O/P - 10
[4 6]
[3 7]

I/P - print(np.min(x))
Print(np.max(x))

O/P - 1
4

| | 0 | 1 |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 3 | 4 |

axis = 0 → 1+3 = 4
2+4 = 6

axis = 1 → 1+2 = 3
3+4 = 7

I/p - Print(x)

Print(x.T)

O/p - [[1 2]
[3 4]]

[[1 3]
[2 4]]

I/p - np.exp(1)

O/p - 2.7182

$e^0 = 1$

$e^1 = 2.718 \simeq 2.72$

I/p - np.exp(2)

O/p - 7.3890

$e^2 = (2.72)^2 = 7.389$

NUMPY STATISTICS:

1. Mean → $\frac{n}{2}$
2. Median → odd: $\frac{n+1}{2}$ , even: $\frac{n}{2} + \frac{n}{2} + 1$
3. Mode → this is not used
4. Standard deviation → std = $\sqrt{\text{variance}}$
5. Correlation coefficient

I/p - x = np.array([160, 180, 146, 162, 184, 180])

Print(np.mean(x))

Print(np.median(x))

Print(np.var(x))

Print(np.std(x))

O/p - 168.666

171.0

187.555

13.695

I/p - heights = np.array([160, 180, 146, 162, 184, 180])

weights = np.array([50, 78, 45, 51, 80, 60])

np.corrcoef(heights, weights)

O/p - array([[1. , 0.885]

[0.885, 1. ]])

NUMPY RESHAPE:

Reshape means changing the shape of an array.

↳ The shape of an array is the number of elements in each dimension.

↳ By reshaping, we can add or remove dimensions or change number of elements in each dimension.

I/p - arr = np.arange(0, 50, 5)

Print(arr)

O/p - [0 5 10 15 20 25 30 35 40 45]

I/p - arr2d = arr.reshape(2, 5)

Print(arr2d)

O/p - [[0 5 10 15 20]
[25 30 35 40 45]]

I/p - arr2d = arr.reshape(5, 2)

Print(arr2d)

O/p - [[0 5]
[10 15]
[20 25]
[30 35]
[40 45]]

I/P - arr2d = arr.reshape(3,3)

O/P - error - can't reshape array of size 10 into shape (3,3)

I/P - arr = np.arange(5,50)
print(arr.shape)

O/P - (45, )

I/P - print(arr.reshape(5,9))

O/P - [[5 6 7 8 9 10 11 12 13]
[14 15 16 17 18 19 20 21 22]
[23 24 25 26 27 28 29 30 31]
[32 33 34 35 36 37 38 39 40]
[41 42 43 44 45 46 47 48 49]]

I/P - Print(arr.reshape(3, 3, 5))

O/P - [[[5 6 7 8 9]
[10 11 12 13 14]
[15 16 17 18 19]]

```
[[[20  21  22  23  24]
  [25  26  27  28  29]
  [30  31  32  33  34]]

 [[35  36  37  38  39]
  [40  41  42  43  44]
  [45  46  47  48  49]]]
```

FLATTEN:

this function return a copy of the array collapsed in one dimension.

↳ If you modify the value of this array, we can notice that the value of original array is not affected.

↳ Flatten() is comparatively slower than ravel() as it occupies memory.

↳ Flatten is a method of an ndarray object.

I/p - print(arr.flatten())

O/p - [5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46
47 48 49]

LINSPACE:

The linspace() function returns evenly spaced numbers over a specified interval [start, stop]. The endpoint of the interval can optionally be excluded.

SYNTAX: np.linspace(start, end, # of elements)

I/p - print(np.linspace(1,5,9))

O/p - [1. 1.5 2. 2.5 3. 3.5 4. 4.5 5.]

I/p - print(np.linspace(10,13,6))

O/p - [10. 10.6 11.2 11.8 12.4 13.]

SORTING:



This function returns a sorted copy of an array.

I/p - arr = np.random.randint(50, 100, size=(5,10))

Print(arr)

O/p - [[89 98 58 70 88 69 84 88 81 52]
[57 76 83 53 60 92 82 81 74 70]
[75 89 84 86 81 62 84 54 69 51]
[87 98 76 72 87 73 52 84 61 57]
[77 94 90 53 60 58 88 77 96 56]]

I/p - np.sort(arr, axis = 0) # column wise sorting

O/p - array([[57, 76, 58, 53, 60, 58, 52, 54, 61, 51],
[75, 89, 76, 53, 60, 62, 82, 77, 69, 52],
[77, 94, 83, 70, 81, 69, 84, 81, 74, 56],
[87, 98, 84, 72, 87, 73, 84, 84, 81, 57],
[89, 98, 90, 86, 88, 92, 84, 88, 96, 70]])

i/p - np.sort(arr, axis = 1) # Row wise sorting

o/p - array([[52, 56, 69, 70, 81, 84, 88, 88, 89, 98],
[53, 57, 60, 70, 74, 76, 81, 82, 83, 92],
[51, 54, 62, 69, 75, 81, 84, 84, 86, 89],
[52, 57, 61, 72, 73, 76, 84, 87, 87, 98],
[53, 56, 58, 60, 77, 77, 88, 90, 94, 96]])

STACKING:

This function is used to join a sequence of same dimension arrays along a new axis.

The axis parameter specifies the index of the new axis in the dimensions of the result.

For Example:

If axis = 0 it will be the first dimension and if axis = -1 it will be the last dimension.

SYNTAX: numpy.stack(arrays, axis)

I/P - arr1 = np.arange(5,15).reshape(2,5)

Print(arr1)

150

O/P - [[5 6 7 8 9]
[10 11 12 13 14]]

I/P - arr2 = np.arange(25,35).reshape(2,5)

Print(arr2)

[[25 26 27 28 29]
[30 31 32 33 34]]

I/P - np.vstack([arr1,arr2]) # Vertical stacking

O/P - array([[5, 6, 7, 8, 9],
[10, 11, 12, 13, 14],
[25, 26, 27, 28, 29]
[30, 31, 32, 33, 34]])

X, Y
vstack = X
Y

I/P - np.hstack([arr1, arr2]) # Horizantal stacking.

O/P - array([[5, 6, 7, 8, 9, 25, 26, 27, 28, 29]
[10, 11, 12, 13, 14, 30, 31, 32, 33, 34]])

X, Y
hstack = X Y

CONCATENATE:

This function concatenate a sequence of arrays along an existing axis.

i/p - np.concatenate([arr1, arr2], axis = 0)

o/p - array([[5, 6, 7, 8, 9],
[10, 11, 12, 13, 14],
[25, 26, 27, 28, 29],
[30, 31, 32, 33, 34]])

i/p - np.concatenate([arr1, arr2], axis = 1)

o/p - array([[5, 6, 7, 8, 9, 25, 26, 27, 28, 29],
[10, 11, 12, 13, 14, 30, 31, 32, 33, 34]])

APPEND:

This appends values along the mentioned axis at the end of the array.

i/p - np.append(arr1, arr2, axis = 1)

o/p - array([[5, 6, 7, 8, 9, 25, 26, 27, 28, 29]
[10, 11, 12, 13, 14, 30, 31, 32, 33, 34]])

I/P - np.append(arr1, arr2, axis = 0)

O/P - array([[5, 6, 7, 8, 9],
[10, 11, 12, 13, 14],
[25, 26, 27, 28, 29],
[30, 31, 32, 33, 34]])

WHERE:

This function returns the indices of elements in an input array where the given condition is satisfied.

I/P - arr = np.arange(50,100).reshape(5,10)
print(arr)

O/P - [[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]]

I/p - np.where(arr > 64, 0, arr)

O/p - array([[50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
[60, 61, 62, 63, 64, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

I/p - np.where(arr > 64, arr/10, arr)

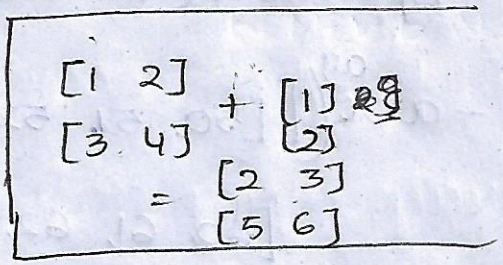
O/p - array([[50., 51., 52., 53., 54., 55., 56., 57., 58., 59.],
[60., 61., 62., 63., 64., 6.5, 6.6, 6.7, 6.8, 6.9],
[7., 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9],
[8., 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9],
[9., 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9]])

I/p - arr1 = np.array([[1,2],[3,4]])
arr2 = np.array([[1],[2]])
Print(arr1)
Print(arr2)

O/p - [[1 2]
[3 4]]
[[1]
[2]]

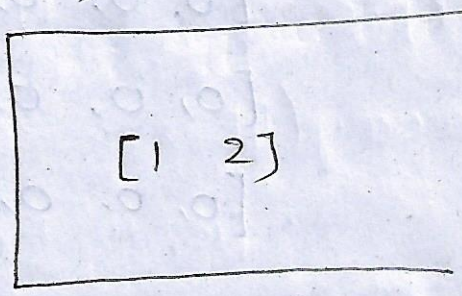
I/p - Print(arr1 + arr2)

O/p - [[2 3]
[5 6]]



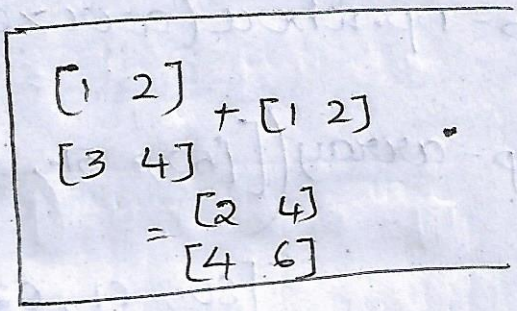

I/p - arr3 = np.array([1, 2])
Print(arr3)

O/p - [1 2]



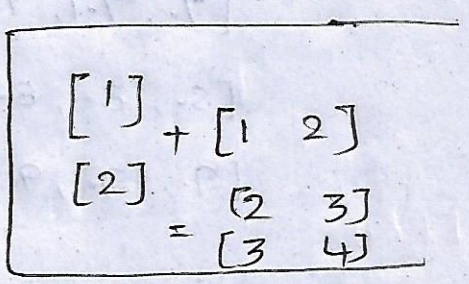

I/p - Print(arr1 + arr3)

O/p - [[2 4]
[4 6]]




I/p - Print(arr2 + arr3)

O/p - [[2 3]
[3 4]]




I/p - arr4 = np.array([[1,2,3],[4,5,6]])
Print(arr4)

O/p - [[1 2 3]
[4 5 6]]

I/p - print (arr2 + arr4)

O/p - [[2 3 4]
[6 7 8]]

$$\begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} = \begin{bmatrix}2 & 3 & 4\\6 & 7 & 8\end{bmatrix}$$

I/p - print (arr4 + arr3)

O/p - Error -- operands could not be broadcast together with shapes (2,3) (2,)