Numpy

Introduction

- NumPy is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, and matrices.
- NumPy stands for Numerical Python.

Why Use NumPy?

In Python we have **lists** that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

Which Language is NumPy written in?

It is written partially in Python but most of the parts that require fast computation are written in C or C++.

Importing numpy

NumPy is usually imported under the np alias. **syntax**: import numpy as np

Creating array

arr = np.array([1, 2, 3, 4, 5]) # creating array using array() function.

print(type(arr),” ”,arr.dtype)   # <class 'numpy.ndarray'> int64

Array Shape & size

```
Returns tuple of ints ex: np.shape([[1, 3]])# (1, 2) size =2
```

Saving numpy arrays

```
>>> x = np.arange(10); np.save(outfile, x) #file has ext .npy
>>> np.load(outfile) # array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

numpy.zeros

Return a new array of given shape and type, filled with zeros.

```
np.zeros((2, 1))                    np.zeros((5,), dtype=int)
>> array([[ 0.],[ 0.]])            >> array([0, 0, 0, 0, 0])
```

numpy.ones

Return a new array of given shape and type, filled with ones. np.ones(5);array([1., *1., 1., 1., 1.])*

numpy.full

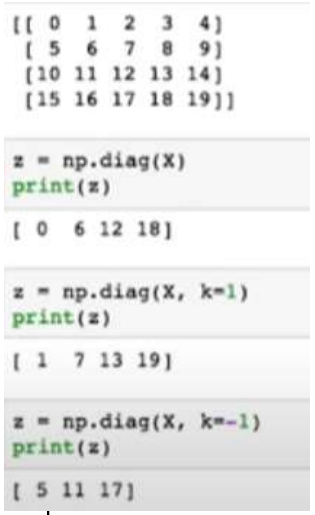
Return a new array of given shape and type, filled with fill_value.

np.full((2, 2), 10)	np.full((2, 2), [1, 2])
array([[10, 10], [10, 10]])	array([[1, 2], [1, 2]])

numpy.eye

Returns: ndarray of shape (N,M)An array where all elements are equal to zero, except for the k-th diagonal, whose values are equal to one.

np.eye(3, k=1)	np.eye(2, dtype=int)
array([[0., 1., 0.], [0., 0., 1.],[0., 0., 0.]]	array([[1, 0], [0, 1]])



```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
```

```
z = np.diag(X)
print(z)
```

```
[ 0  6 12 18]
```

```
z = np.diag(X, k=1)
print(z)
```

```
[ 1  7 13 19]
```

```
z = np.diag(X, k=-1)
print(z)
```

```
[ 5 11 17]
```

numpy.diag

Extract a diagonal or construct a diagonal array.

>> x #array([[0, 1, 2],[3, 4, 5],[6, 7, 8]])

>> np.diag(x) # array([0, 4, 8]) >> extracts the diagonal

>> np.diag(np.diag(x)) # array([[0, 0, 0], [0, 4, 0], [0, 0, 8]]) >> construct a diagonal array

numpy.linspace

Return evenly spaced numbers over a specified interval.

Returns num evenly spaced samples, calculated over the interval [start, stop].

np.linspace(2.0, 3.0, num=5, endpoint=False) # array([2. , 2.2, 2.4, 2.6, 2.8])

numpy.reshape

Gives a new shape to an array without changing its data.

The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

a = np.arange(6).reshape((3, 2)) # array([[0, 1],[2, 3], [4, 5]])

np.reshape(a, (3,-1))   # the unspecified value is inferred to be 2

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

numpy.random.rand

or numpy.random.random : Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1).

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

array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

numpy.random.randint

Return **random integers** from low (inclusive) to high (exclusive).

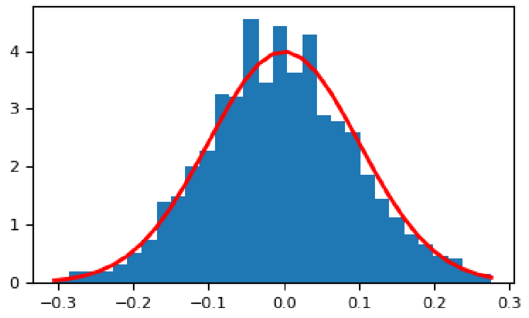
Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

```
np.random.randint(5, size=(2, 4))

array([[4, 0, 2, 1], [3, 2, 2, 0]])
```

numpy.random.normal

Draw random samples from a normal (Gaussian) distribution.

```
mu, sigma = 0, 0.1 # mean and standard deviation

>>> s = np.random.normal(mu, sigma, 1000) #1000 x 1000
```



numpy.delete

Return a new array with sub-arrays along an axis deleted. For a one dimensional array, this returns those entries not returned by arr[obj].

numpy.delete(arr, obj, axis=None) axis=0 for rows and axis= 1 for columns

obj :slice, int or array of ints Indicate indices of sub-arrays to remove along the specified axis.

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

np.delete(arr, 1, 0) # delete first raw as axis =0 for rows

array([[ 1,  2,  3,  4],[ 9, 10, 11, 12]])
```

numpy.append

Append values to the end of an array.

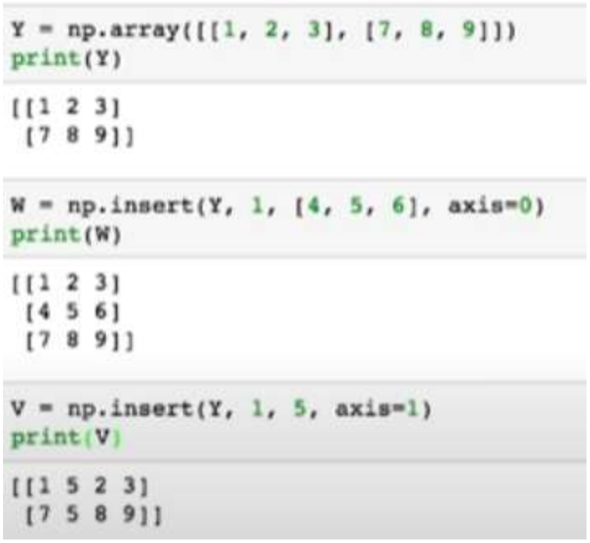
numpy.append(arr, values, axis=None)

np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0) appended as   rows as axis=0

array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) but if axis =1 we append columns here

numpy.insert

Insert values along the given axis before the given indices.



```
Y = np.array([[1, 2, 3], [7, 8, 9]])
print(Y)

[[1 2 3]
 [7 8 9]]

W = np.insert(Y, 1, [4, 5, 6], axis=0)
print(W)

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

V = np.insert(Y, 1, 5, axis=1)
print(V)

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

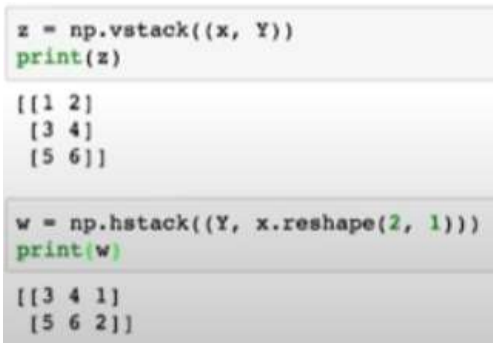
[5, 6]]

Numpy.hstack

Numpy.vstack

Stack arrays in sequence horizontally hstack

(column wise) Or vertically ***vstack***.



```
z = np.vstack((x, Y))
print(z)

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

w = np.hstack((Y, x.reshape(2, 1)))
print(w)

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

NumPy Array Indexing

You can access an array element by referring to its index

number.index -1 refer to last item

Ex: arr = np.array([[1,2,3,4,5], [6,7,8,9,10]]); print(arr[0,1]) # for the first row, second column

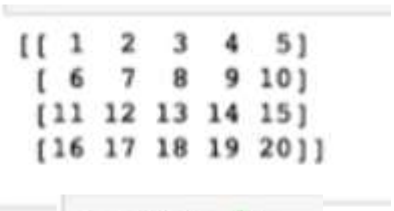
Use negative indexing to access an array from the end.ex arr[1, -1] for 1st raw and last column

Dimensions in Arrays

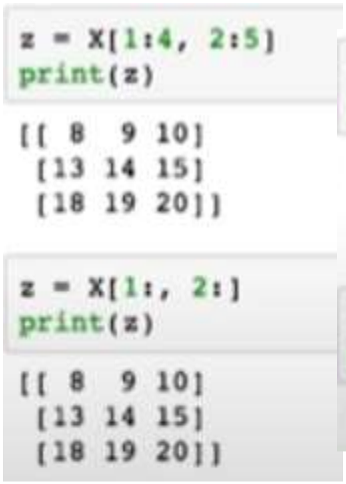
- Scalars: 0-d array has single value ex: arr = np.array(42)
- 1-D Arrays: has 0-D arrays as its elements ex: arr = np.array([1, 2, 3, 4, 5])
- 2-D Arrays: has 1-D arrays as its elements ex: arr = np.array([[1, 2, 3], [4, 5, 6]])

We can check Number of Dimensions using ndim

NumPy Array Slicing



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

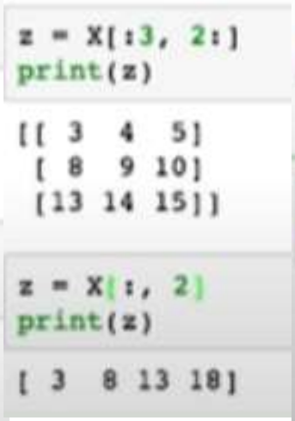


```
z = X[1:4, 2:5]
print(z)
```

```
[[ 8  9 10]
 [13 14 15]
 [18 19 20]]
```

```
z = X[1:, 2:]
print(z)
```

```
[[ 8  9 10]
 [13 14 15]
 [18 19 20]]
```

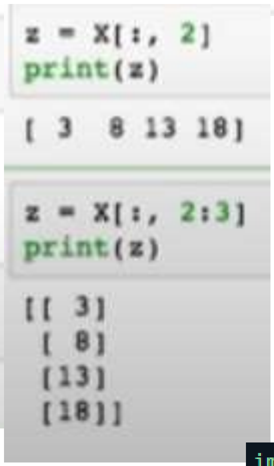


```
z = X[:3, 2:]
print(z)
```

```
[[ 3  4  5]
 [ 8  9 10]
 [13 14 15]]
```

```
z = X[:, 2]
print(z)
```

```
[ 3  8 13 18]
```



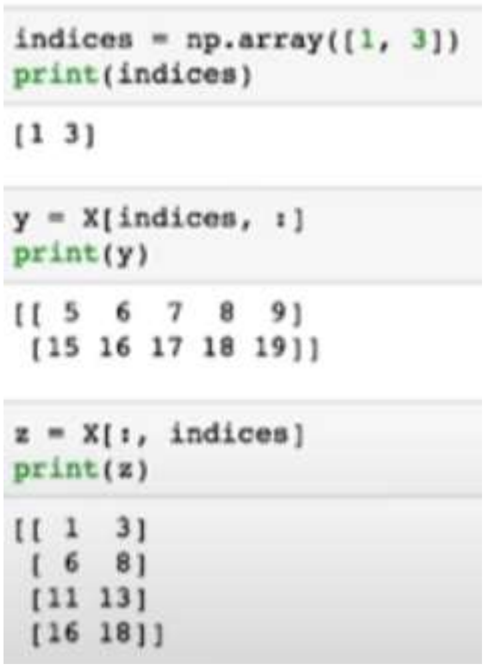
```
z = X[:, 2]
print(z)
```

```
[ 3  8 13 18]
```

```
z = X[:, 2:3]
print(z)
```

```
[[ 3]
 [ 8]
 [13]
 [18]]
```

Slicing with indices



```
indices = np.array([1, 3])
print(indices)
```

```
[1 3]
```

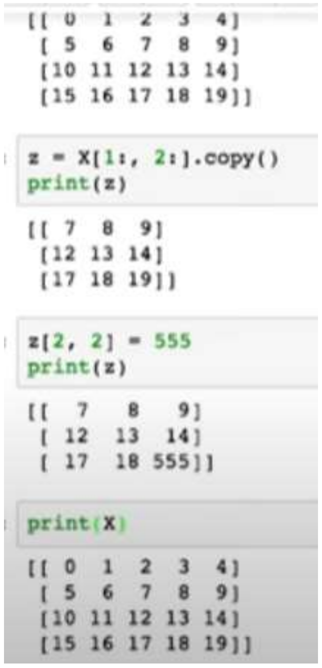
```
y = X[indices, :]
print(y)
```

```
[[ 5  6  7  8  9]
 [15 16 17 18 19]]
```

```
z = X[:, indices]
print(z)
```

```
[[ 1  3]
 [ 6  8]
 [11 13]
 [16 18]]
```

numpy.copy



```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
```

```
z = X[1:, 2:].copy()
print(z)
```

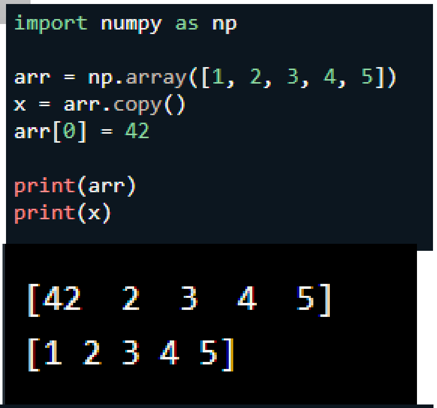
```
[[ 7  8  9]
 [12 13 14]
 [17 18 19]]
```

```
z[2, 2] = 555
print(z)
```

```
[[  7   8   9]
 [ 12  13  14]
 [ 17  18 555]]
```

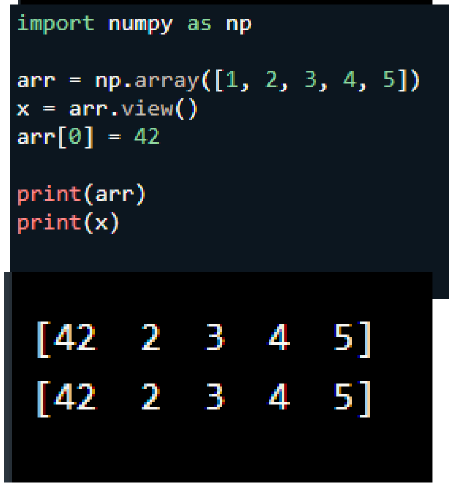
```
print(X)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
```



```
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]
```



```
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]
```

NumPy Array Copy vs View

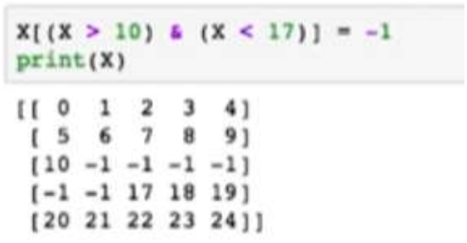
The copy is a new array, and the view is just a view of the original array.

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

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.

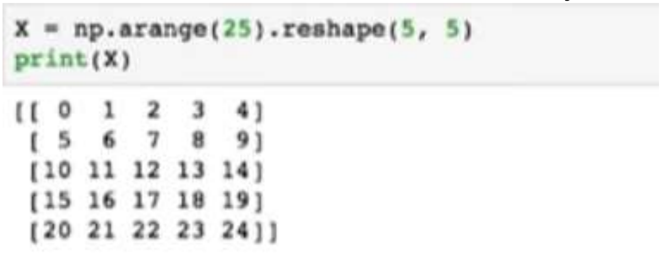
Boolean indexing

- Numpy allows you to use an array of boolean values as an index of another array.
- Each element of the boolean array indicates whether or not to select the elements from the array.



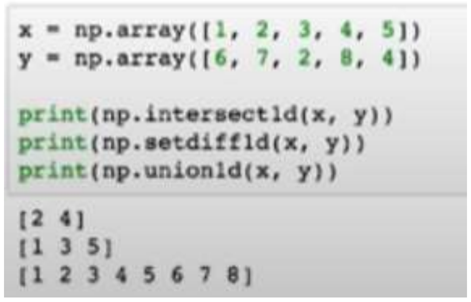
```
X[(X > 10) & (X < 17)] = -1
print(X)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 -1 -1 -1 -1]
 [-1 -1 17 18 19]
 [20 21 22 23 24]]
```



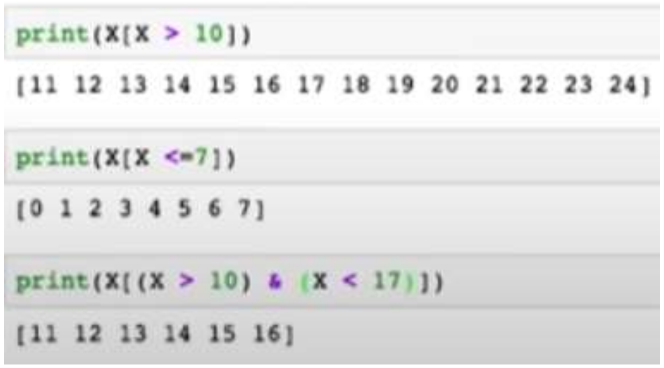
```
X = np.arange(25).reshape(5, 5)
print(X)
```

```
[[ 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]]
```



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

print(np.intersect1d(x, y))
print(np.setdiff1d(x, y))
print(np.union1d(x, y))
```

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



```
print(X[X > 10])
```

```
[11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

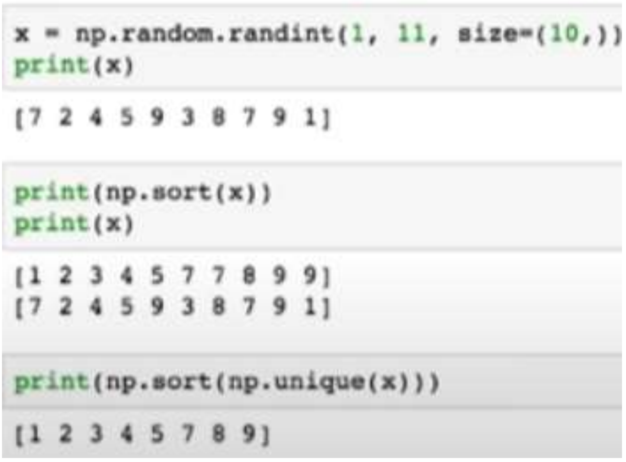
```
print(X[X <=7])
```

```
[0 1 2 3 4 5 6 7]
```

```
print(X[(X > 10) & (X < 17)])
```

```
[11 12 13 14 15 16]
```

Sorting array



```
x = np.random.randint(1, 11, size=(10,))
print(x)
```

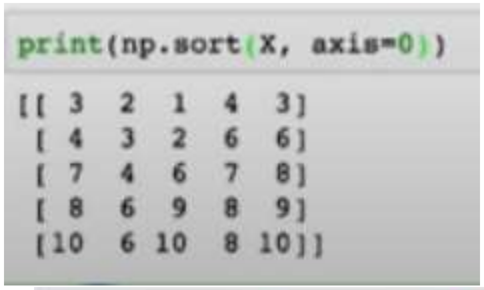
```
[7 2 4 5 9 3 8 7 9 1]
```

```
print(np.sort(x))
print(x)
```

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

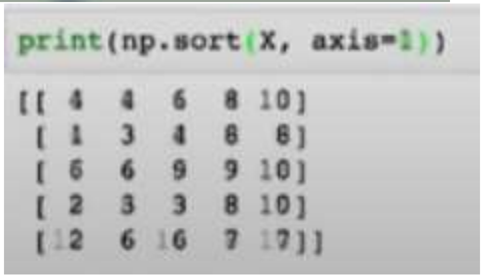
```
print(np.sort(np.unique(x)))
```

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



```
print(np.sort(X, axis=0))
```

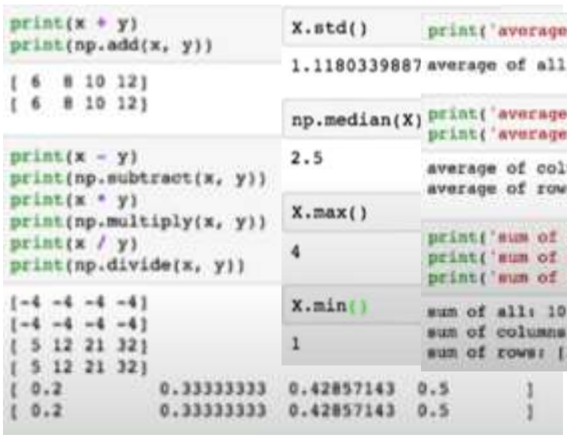
```
[[ 3  2  1  4  3]
 [ 4  3  2  6  6]
 [ 7  4  6  7  8]
 [ 8  6  9  8  9]
 [10  6 10  8 10]]
```



```
print(np.sort(X, axis=1))
```

```
[[ 4  4  6  8 10]
 [ 1  3  4  8  8]
 [ 6  6  9  9 10]
 [ 2  3  3  8 10]
 [12  6 16  7 17]]
```

Numpy arthimatic operations

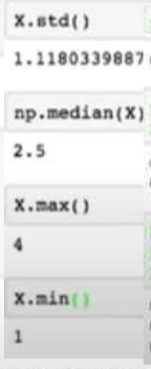


```
print(x + y)
print(np.add(x, y))
```

```
[ 6  8 10 12]
[ 6  8 10 12]
```

```
print(x - y)
print(np.subtract(x, y))
print(x * y)
print(np.multiply(x, y))
print(x / y)
print(np.divide(x, y))
```

```
[-4 -4 -4 -4]
[-4 -4 -4 -4]
[ 5 12 21 32]
[ 5 12 21 32]
[ 0.2         0.33333333  0.42857143  0.5       ]
[ 0.2         0.33333333  0.42857143  0.5       ]
```



```
X.std()
```

```
1.1180339887
```

```
np.median(X)
```

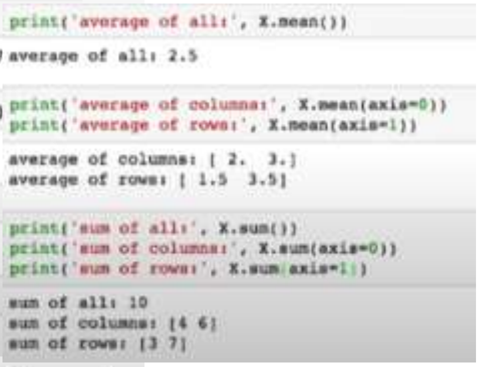
```
2.5
```

```
X.max()
```

```
4
```

```
X.min()
```

```
1
```



```
print('average of all:', X.mean())
```

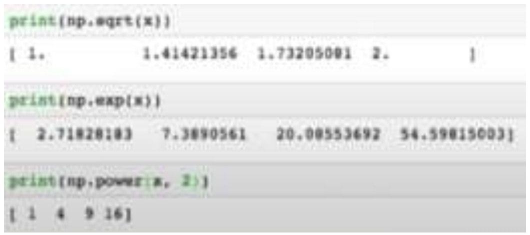
```
average of all: 2.5
```

```
print('average of columns:', X.mean(axis=0))
print('average of rows:', X.mean(axis=1))
```

```
average of columns: [ 2.  3.]
average of rows: [ 1.5  3.5]
```

```
print('sum of all:', X.sum())
print('sum of columns:', X.sum(axis=0))
print('sum of rows:', X.sum(axis=1))
```

```
sum of all: 10
sum of columns: [4 6]
sum of rows: [3 7]
```



```
print(np.sqrt(x))
```

```
[ 1.          1.41421356  1.73205081  2.        ]
```

```
print(np.exp(x))
```

```
[  2.71828183   7.3890561   20.08553692  54.59815003]
```

```
print(np.power(x, 2))
```

```
[ 1  4  9 16]
```