Data Science Workshop-1

ITER, SOA University

Ashis Kumar Pati

Centre for Data Science
SOA University
ashispati@soa.ac.in

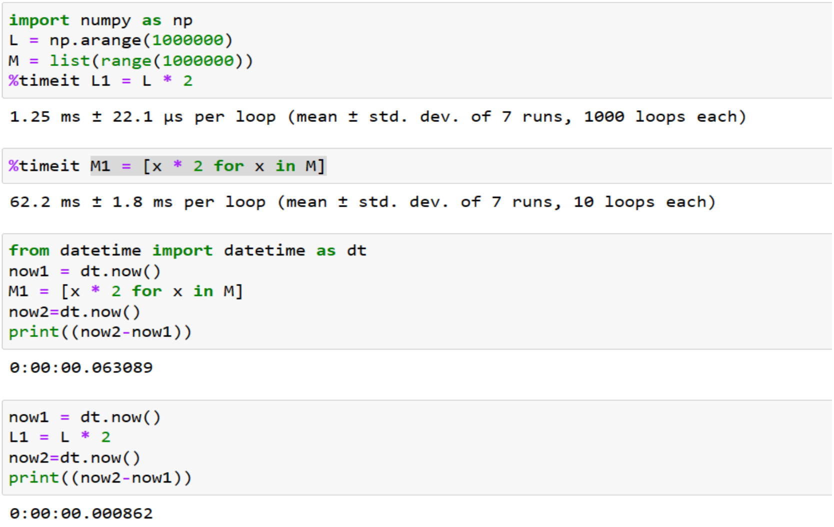
All about numpy

- ndarray:- Multidimensional array providing fast array oriented arithmetic operations
- Mathematical functions for fast operations on entire arrays of data(without writing loops)
- Tools for reading/writing array data to disk and working with memory mapped files.(A technique that allows a part of the virtual address space to be associated with a file logically, which will increase the performance)
- Linear algebra, random number generation
- A C API for connecting NumPy with libraries written in C,C++ or FORTRAN(interface means contract of service between two applications)

Contents

- Creating ndarrays(), Datatypes, shape and ndim in numpy
- Arithmetic Operations, Numpy Broadcasting
- indexing and slicing
- random module in numpy
- universal functions
- array oriented programming and conditional logic as array operation
- Mathematical and statistical methods
- sorting and unique other set logic
- file input and output

ndarrays are faster



```
import numpy as np
L = np.arange(1000000)
M = list(range(1000000))
%timeit L1 = L * 2
```

```
1.25 ms ± 22.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
%timeit M1 = [x * 2 for x in M]
```

```
62.2 ms ± 1.8 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
from datetime import datetime as dt
now1 = dt.now()
M1 = [x * 2 for x in M]
now2=dt.now()
print((now2-now1))
```

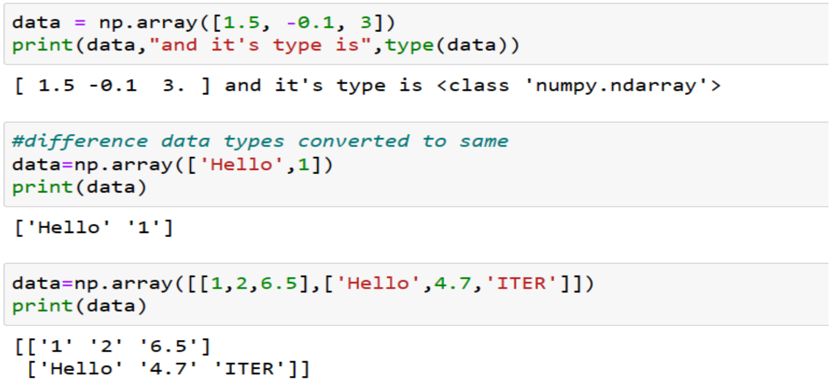
```
0:00:00.063089
```

```
now1 = dt.now()
L1 = L * 2
now2=dt.now()
print((now2-now1))
```

```
0:00:00.000862
```

Creating arrays in numpy

- An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type
- The easiest way to create an ndarray is to use the array function in numpy module.
- Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array



```
data = np.array([1.5, -0.1, 3])
print(data,"and it's type is",type(data))
```

```
[ 1.5 -0.1  3. ] and it's type is <class 'numpy.ndarray'>
```

```
#difference data types converted to same
data=np.array(['Hello',1])
print(data)
```

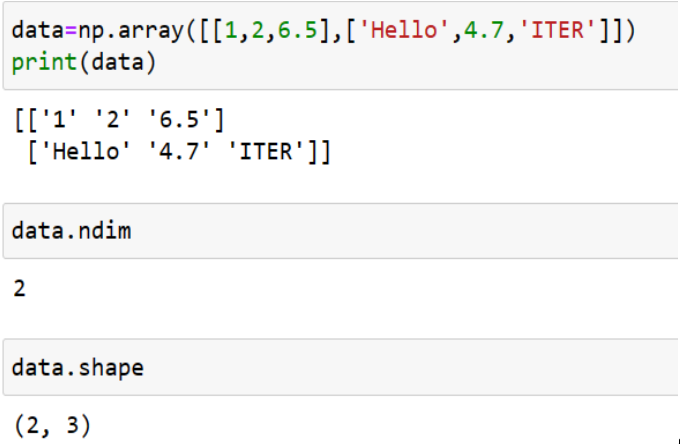
```
['Hello' '1']
```

```
data=np.array([[1,2,6.5],['Hello',4.7,'ITER']])
print(data)
```

```
[['1' '2' '6.5']
 ['Hello' '4.7' 'ITER']]
```

Shape and dimension of ndarray

- arr.ndim: function return the number of dimensions of an array.
- arr.shape: shape of an array is the number of elements in each dimension.
- arr.size: Try this and see what you are getting.



```
data=np.array([[1,2,6.5],['Hello',4.7,'ITER']])
print(data)
```

```
[['1' '2' '6.5']
 ['Hello' '4.7' 'ITER']]
```

```
data.ndim
```

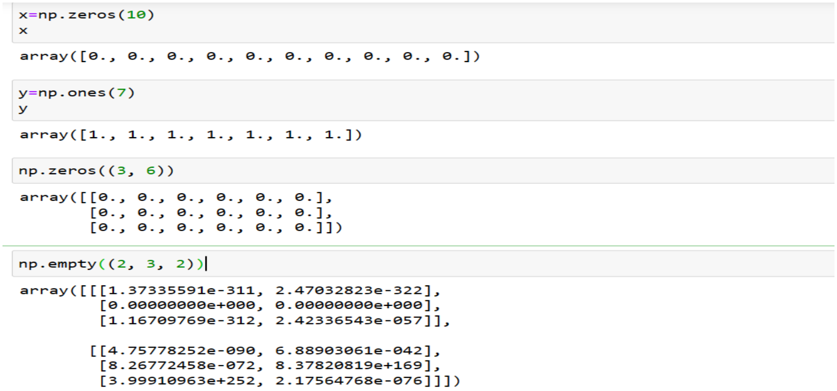
```
2
```

```
data.shape
```

```
(2, 3)
```

Other functions for creating neew arrays

- numpy.zeros and numpy.ones create arrays of 0s or 1s, respectively, with a given length or shape.
- numpy.empty creates an array without initializing its values to any particular value.
- To create a higher dimensional array with these methods, pass a tuple for the shape.



```
x=np.zeros(10)
x
```

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

```
y=np.ones(7)
y
```

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

```
np.zeros((3, 6))
```

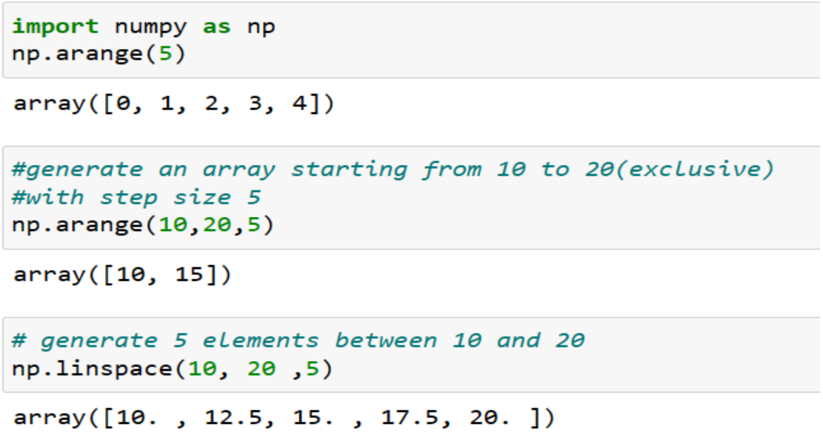
```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

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

```
array([[[1.37335591e-311, 2.47032823e-322],
        [0.00000000e+000, 0.00000000e+000],
        [1.16709769e-312, 2.42336543e-057]],

       [[4.75778252e-090, 6.88903061e-042],
        [8.26772458e-072, 8.37820819e+169],
        [3.99910963e+252, 2.17564768e-076]]])
```

Creating new arrays

- arange Like the built-in range but returns an ndarray instead of a list
- syntax: numpy.arange(start = 0, stop, step = 1, dtype = None)
- linspace() function is used to create an array of evenly spaced numbers within a specified range



```python
import numpy as np
np.arange(5)
```

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

```python
#generate an array starting from 10 to 20(exclusive)
#with step size 5
np.arange(10,20,5)
```

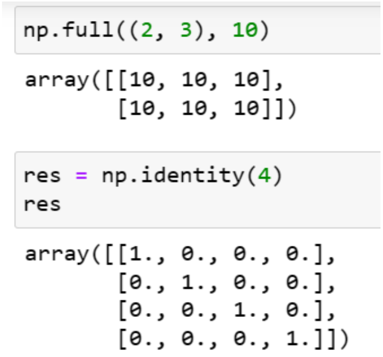
```
array([10, 15])
```

```python
# generate 5 elements between 10 and 20
np.linspace(10, 20 ,5)
```

```
array([10. , 12.5, 15. , 17.5, 20. ])
```

Creating new arrays

- Produce an array of the given shape and data type with all values set to the indicated "fill value"
- numpy.full(shape, fill value)
- eye/identity Create a square N $\times$ N identity matrix (1s on the diagonal and 0s elsewhere)
- Return a new array of given shape and type, filled with fill value.

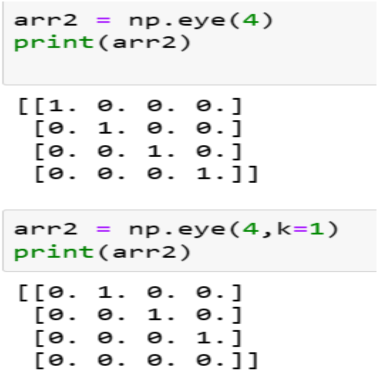


```
np.full((2, 3), 10)
```

```
array([[10, 10, 10],
       [10, 10, 10]])
```

```
res = np.identity(4)
res
```

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```



```
arr2 = np.eye(4)
print(arr2)
```

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

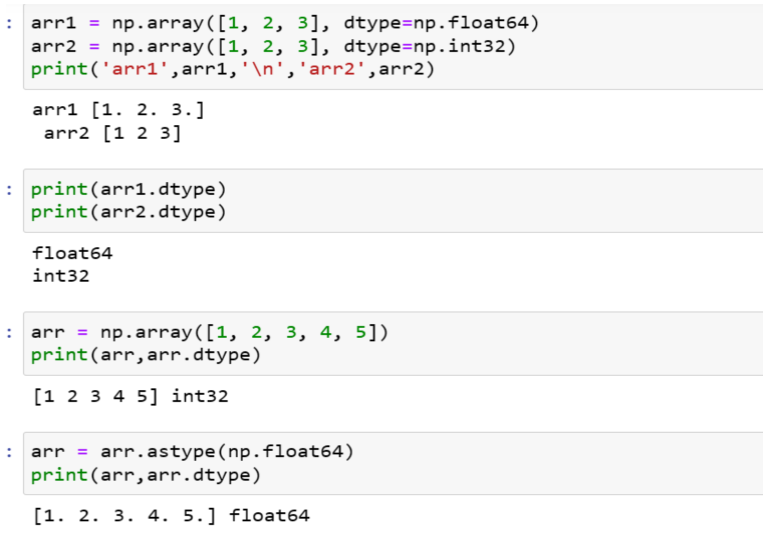
```
arr2 = np.eye(4,k=1)
print(arr2)
```

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

Data types for ndarrays

- The data type or dtype is a special object containing the information about data.
- numpy tries to infer a good data type for the array that it creates.
- You can explicitly convert or cast an array from one data type to another using ndarray's astype method.
- A string which cannot be converted to float64, if we use astype method, a Value Error will be raised

converting data types for ndarrays



```
arr1 = np.array([1, 2, 3], dtype=np.float64)
arr2 = np.array([1, 2, 3], dtype=np.int32)
print('arr1',arr1,'\n','arr2',arr2)
```

```
arr1 [1. 2. 3.]
 arr2 [1 2 3]
```

```
print(arr1.dtype)
print(arr2.dtype)
```

```
float64
int32
```

```
arr = np.array([1, 2, 3, 4, 5])
print(arr,arr.dtype)
```

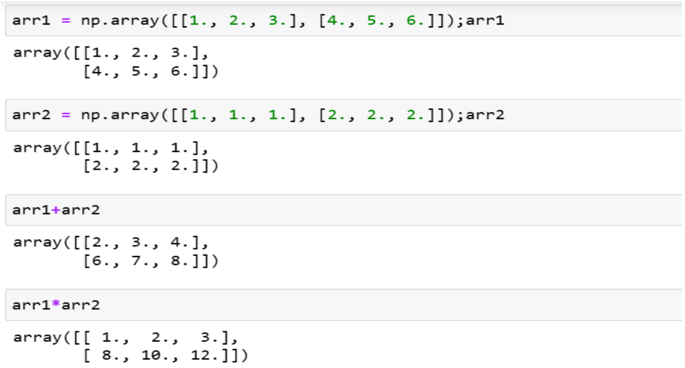
```
[1 2 3 4 5] int32
```

```
arr = arr.astype(np.float64)
print(arr,arr.dtype)
```

```
[1. 2. 3. 4. 5.] float64
```

Arithmetic With NumPy arrays

- Batch operations on data can be performed in numpy without writing any for loops. This is known as vectorization.
- Any arithmetic operations between equal-size arrays apply the operation element-wise:



```
arr1 = np.array([[1., 2., 3.], [4., 5., 6.]]);arr1
```

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

```
arr2 = np.array([[1., 1., 1.], [2., 2., 2.]]);arr2
```

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

```
arr1+arr2
```

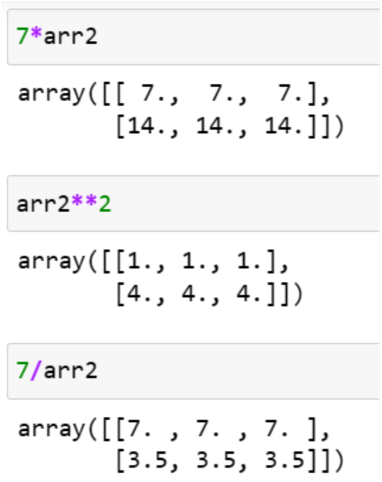
```
array([[2., 3., 4.],
       [6., 7., 8.]])
```

```
arr1*arr2
```

```
array([[ 1.,  2.,  3.],
       [ 8., 10., 12.]])
```

Arithmetic operations

- Arithmetic operations with scalars propagate the scalar argument to each element in the array
- Comparisons between arrays of the same size yield Boolean arrays.



```
7*arr2
```

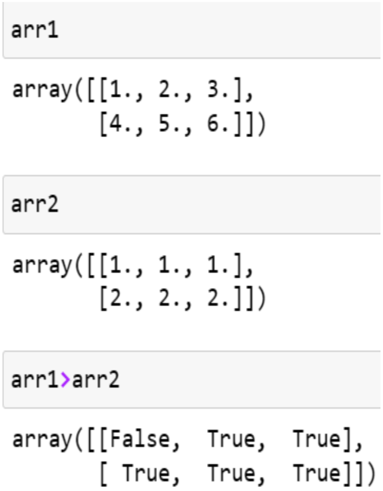
```
array([[ 7.,  7.,  7.],
       [14., 14., 14.]])
```

```
arr2**2
```

```
array([[1., 1., 1.],
       [4., 4., 4.]])
```

```
7/arr2
```

```
array([[7. , 7. , 7. ],
       [3.5, 3.5, 3.5]])
```



```
arr1
```

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

```
arr2
```

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

```
arr1>arr2
```

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

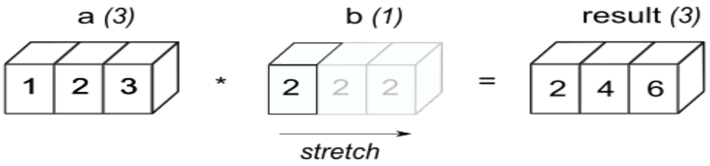
Let's solve

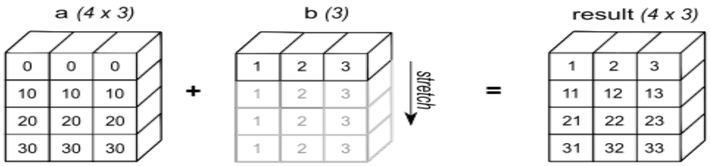
check point

- Create an ndarray object with float data type. Convert this to integer data type. Observe the changes in the data.
- Create identity matrix of order 4, and another matrix of order 4 with sub-diagonal elements 1 and super-diagonal elements 1, and other element zero. Compare the elements of each matrix.

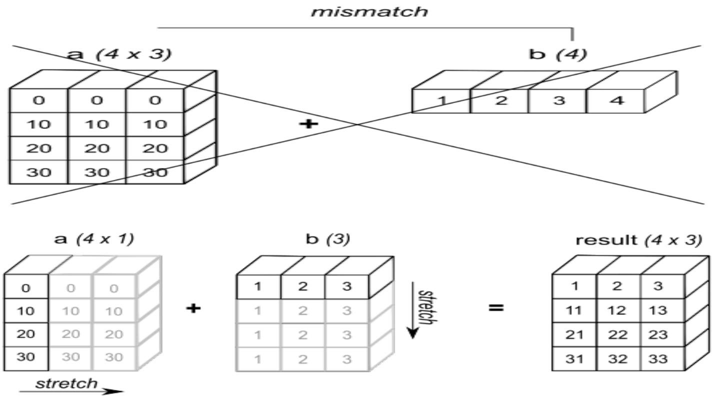
Numpy Broadcasting

- Broadcasting refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations.
- The smaller array is broadcast to the size of the larger array so that they have compatible shapes.





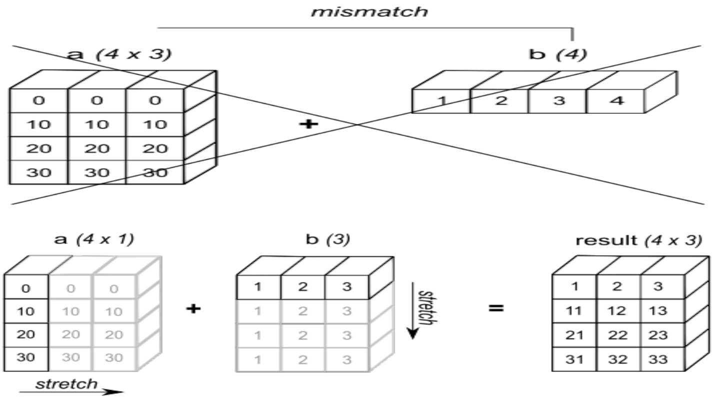
Numpy Broadcasting



Check Point

Take an 3 dimensional array of shape $4 \times 7 \times 1$. Find all possible 2-dimensional and 3-dimensional array shapes which are compartible with the above array.

Numpy Broadcasting

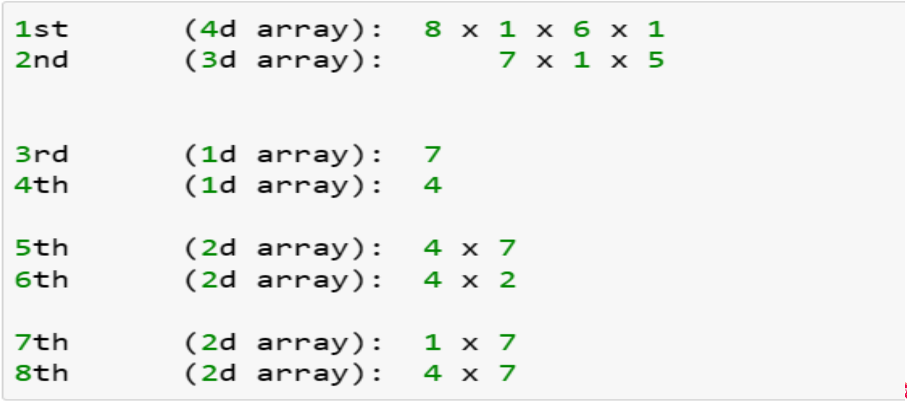


Check Point

Take an 3 dimensional array of shape $4 \times 7 \times 1$. Find all possible 2-dimensional and 3-dimensional array shapes which are compartible with the above array. Answer $1 \times 1 \times k$, $1 \times 7 \times k$, $4 \times 1 \times k$, 1, 1×1, 4×1, 4×7

NumPy Broadcasting

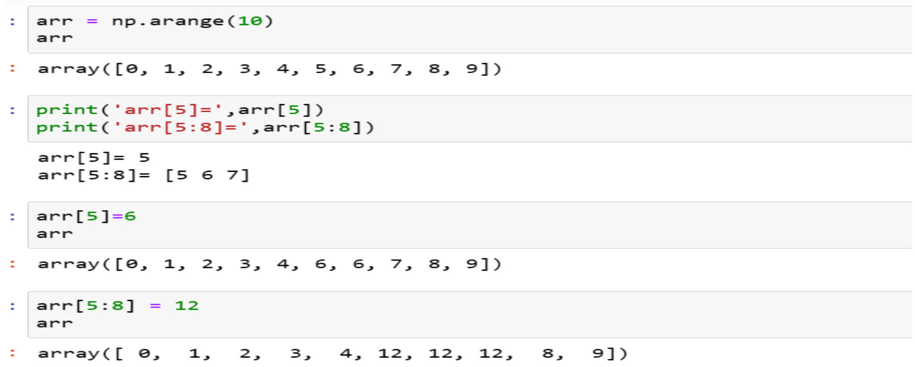
- Two dimensions are compatible(broadcast-able) when they are equal, or one of them is 1.



```
1st        (4d array):   8 x 1 x 6 x 1
2nd        (3d array):       7 x 1 x 5


3rd        (1d array):   7
4th        (1d array):   4

5th        (2d array):   4 x 7
6th        (2d array):   4 x 2

7th        (2d array):   1 x 7
8th        (2d array):   4 x 7
```

Basic Indexing and Slicing

- One-dimensional ndarrays indexing are similar to Python lists indexing.



```
:  arr = np.arange(10)
   arr

:  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

:  print('arr[5]=',arr[5])
   print('arr[5:8]=',arr[5:8])

   arr[5]= 5
   arr[5:8]= [5 6 7]

:  arr[5]=6
   arr

:  array([0, 1, 2, 3, 4, 6, 6, 7, 8, 9])

:  arr[5:8] = 12
   arr

:  array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

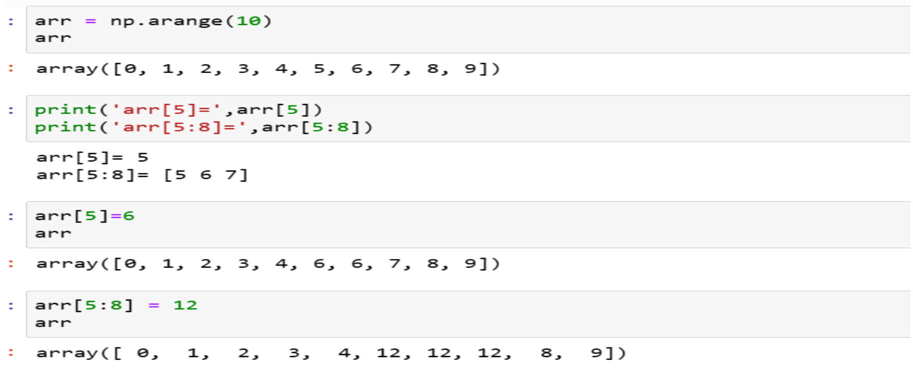
Check Point

1.Try to change each alternative element to 1000.

2.Try to reverse the array

Basic Indexing and Slicing

- One-dimensional ndarrays indexing are similar to Python lists indexing.



```
: arr = np.arange(10)
  arr

: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

: print('arr[5]=',arr[5])
  print('arr[5:8]=',arr[5:8])

  arr[5]= 5
  arr[5:8]= [5 6 7]

: arr[5]=6
  arr

: array([0, 1, 2, 3, 4, 6, 6, 7, 8, 9])

: arr[5:8] = 12
  arr

: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

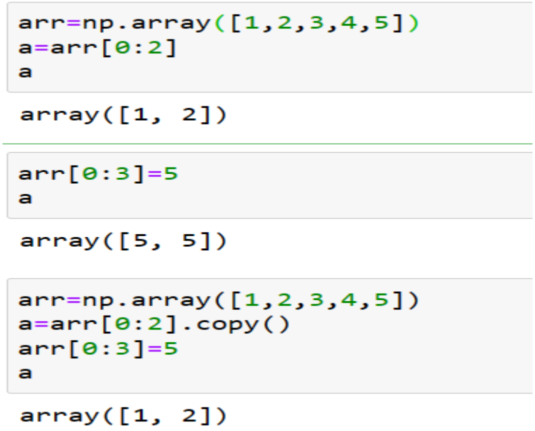
Check Point

1. Try to change each alternative element to 1000.
2. Try to reverse the array

Answer 1- a[::]=1000 2- a[::-1]

More on indexing

- If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array—for example, arr[5:8].copy()



```
arr=np.array([1,2,3,4,5])
a=arr[0:2]
a
```

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

```
arr[0:3]=5
a
```

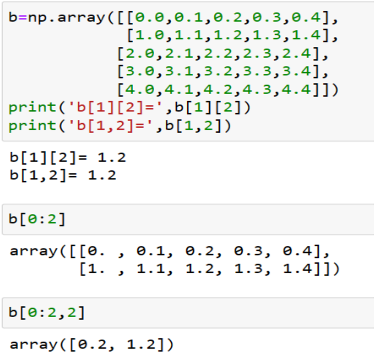
```
array([5, 5])
```

```
arr=np.array([1,2,3,4,5])
a=arr[0:2].copy()
arr[0:3]=5
a
```

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

Multidimensional arrays indexing

- Multidimensional arrays can have one index per axis.
- Individual elements can be accessed recursively.(The way we have done for 2-d python lists). But we can pass a comma-separated list of indices to select individual elements.
- In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional



```
b=np.array([[0.0,0.1,0.2,0.3,0.4],
            [1.0,1.1,1.2,1.3,1.4],
           [2.0,2.1,2.2,2.3,2.4],
           [3.0,3.1,3.2,3.3,3.4],
           [4.0,4.1,4.2,4.3,4.4]])
print('b[1][2]=',b[1][2])
print('b[1,2]=',b[1,2])
```

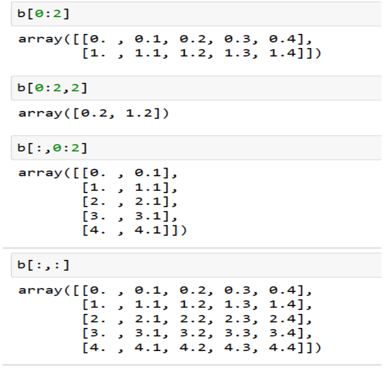
```
b[1][2]= 1.2
b[1,2]= 1.2
```

```
b[0:2]
```

```
array([[0. , 0.1, 0.2, 0.3, 0.4],
       [1. , 1.1, 1.2, 1.3, 1.4]])
```

```
b[0:2,2]
```

```
array([0.2, 1.2])
```



```
b[0:2]
```

```
array([[0. , 0.1, 0.2, 0.3, 0.4],
       [1. , 1.1, 1.2, 1.3, 1.4]])
```

```
b[0:2,2]
```

```
array([0.2, 1.2])
```

```
b[:,0:2]
```

```
array([[0. , 0.1],
       [1. , 1.1],
       [2. , 2.1],
       [3. , 3.1],
       [4. , 4.1]])
```

```
b[:,:]
```

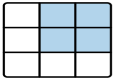
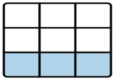
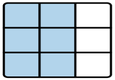
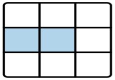
```
array([[0. , 0.1, 0.2, 0.3, 0.4],
       [1. , 1.1, 1.2, 1.3, 1.4],
       [2. , 2.1, 2.2, 2.3, 2.4],
       [3. , 3.1, 3.2, 3.3, 3.4],
       [4. , 4.1, 4.2, 4.3, 4.4]])
```

Multidimensional Indexing

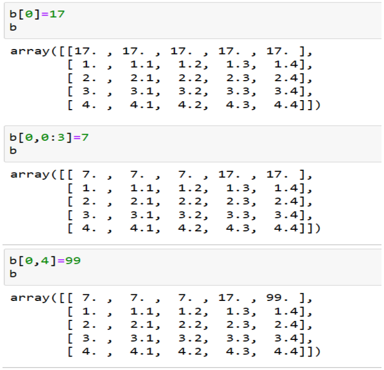
check point

Use array slicing to select the second row but only the first two columns.

- Both scalar values and arrays can be assigned.
- Note, slicing changes the shape of the array.

Expression	Shape
`arr[:2,1:]`	`(2,2)`
`arr[2]`	`(3,)`
`arr[2, :]`	`(3,)`
`arr[2:, :]`	`(1,3)`
`arr[:, :2]`	`(3,2)`
`arr[1, :2]`	`(2,)`
`arr[1:2, :2]`	`(1,2)`



```
b[0]=17
b
```

```
array([[17. , 17. , 17. , 17. , 17. ],
       [ 1. ,  1.1,  1.2,  1.3,  1.4],
       [ 2. ,  2.1,  2.2,  2.3,  2.4],
       [ 3. ,  3.1,  3.2,  3.3,  3.4],
       [ 4. ,  4.1,  4.2,  4.3,  4.4]])
```

```
b[0,0:3]=7
b
```

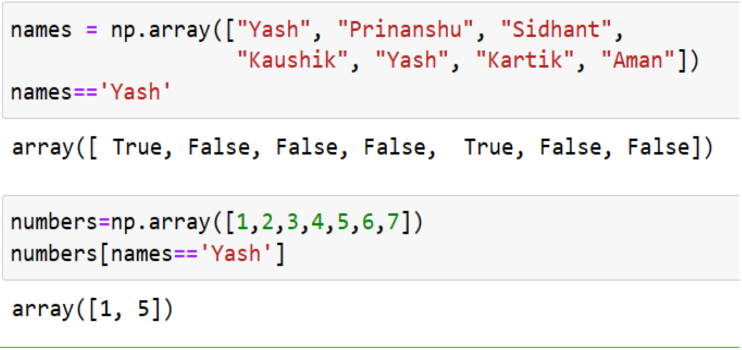
```
array([[ 7. ,  7. ,  7. , 17. , 17. ],
       [ 1. ,  1.1,  1.2,  1.3,  1.4],
       [ 2. ,  2.1,  2.2,  2.3,  2.4],
       [ 3. ,  3.1,  3.2,  3.3,  3.4],
       [ 4. ,  4.1,  4.2,  4.3,  4.4]])
```

```
b[0,4]=99
b
```

```
array([[ 7. ,  7. ,  7. , 17. , 99. ],
       [ 1. ,  1.1,  1.2,  1.3,  1.4],
       [ 2. ,  2.1,  2.2,  2.3,  2.4],
       [ 3. ,  3.1,  3.2,  3.3,  3.4],
       [ 4. ,  4.1,  4.2,  4.3,  4.4]])
```

Boolean Indexing

- Create a Boolean array. That we can do by giving a condition.
- If we pass the Boolean array as the index to an other array of same size. We will get the values, where the Boolean array has value True.



```
names = np.array(["Yash", "Prinanshu", "Sidhant",
                  "Kaushik", "Yash", "Kartik", "Aman"])
names=='Yash'
```

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

```
numbers=np.array([1,2,3,4,5,6,7])
numbers[names=='Yash']
```

```
array([1, 5])
```

Boolean Indexing

Checkpoint

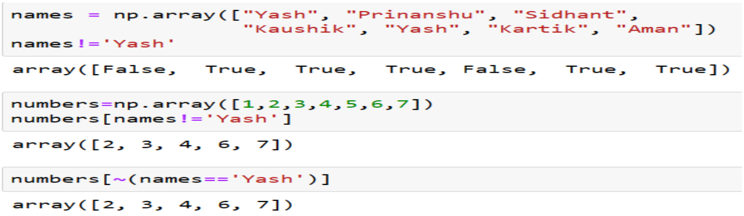
Create two lists one with the name of shoppers and another list withing shopping cart values. Let us assume Abinsh has shopped 3 times, with shopping cart value 17.50Rs, 99.45Rs and 56.34Rs. Add 5more shoppers and theirs values. Find the average shopping value of Abinash.

Boolean Indexing

Checkpoint

Create two lists one with the name of shoppers and another list withing shopping cart values. Let us assume Abinsh has shopped 3 times, with shopping cart value 17.50Rs, 99.45Rs and 56.34Rs. Add 5more shoppers and theirs values. Find the average shopping value of Abinash.

- To select everything but "Yash" you can either use != or negate the condition using ~



```
names = np.array(["Yash", "Prinanshu", "Sidhant",
                  "Kaushik", "Yash", "Kartik", "Aman"])
names!='Yash'
```

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

```
numbers=np.array([1,2,3,4,5,6,7])
numbers[names!='Yash']
```

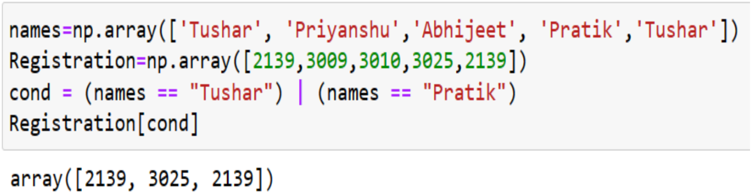
```
array([2, 3, 4, 6, 7])
```

```
numbers[~(names=='Yash')]
```

```
array([2, 3, 4, 6, 7])
```

Boolean Indexing

- To combine multiple Boolean conditions, use Boolean arithmetic operators like & (and) and |(or)



```
names=np.array(['Tushar', 'Priyanshu','Abhijeet', 'Pratik','Tushar'])
Registration=np.array([2139,3009,3010,3025,2139])
cond = (names == "Tushar") | (names == "Pratik")
Registration[cond]
```

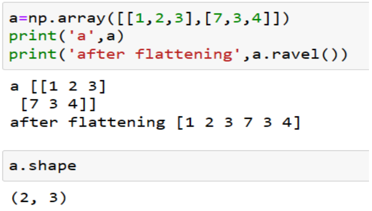
```
array([2139, 3025, 2139])
```

Check point

Store the following three lists into numpy arrays. Find Mark secured by Nisith with roll number 13.
Name=['Satyanistha', 'Nisith','Chandran', 'Swayam', 'Nisith']
Roll=[100,13,39,41,43] Marks=[95,97,96,91,94]

Reshaping the array

- arr.ravel(): returns the array, flattened
- arr.reshape(m,n): returns the array with a modified shape if there exists $m \times n$ elements
- arr.T: returns the array, transposed
- Note that the three commands all return a modified array, but do not change the original array
- Simple transposing with .T is a special case of swapping axes. ndarray has the method arr.swapaxes(m,n), which takes a pair of axis numbers and switches the indicated axes to rearrange the data.

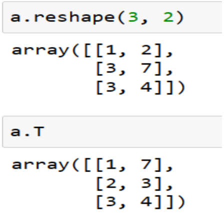


```
a=np.array([[1,2,3],[7,3,4]])
print('a',a)
print('after flattening',a.ravel())
```

```
a [[1 2 3]
 [7 3 4]]
after flattening [1 2 3 7 3 4]
```

```
a.shape
```

```
(2, 3)
```

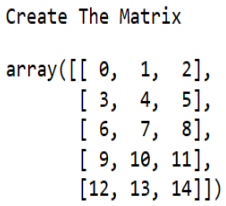


```
a.reshape(3, 2)
```

```
array([[1, 2],
       [3, 7],
       [3, 4]])
```

```
a.T
```

```
array([[1, 7],
       [2, 3],
       [3, 4]])
```



Create The Matrix

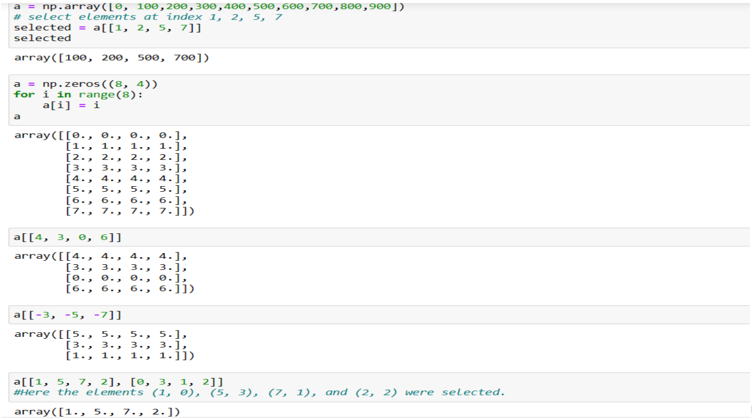
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Fancy Indexing

Check point

Check the resize() function in numpy.

- Rather than using a scalar or slice as an index, an axis can be indexed with an array



```
a = np.array([0, 100,200,300,400,500,600,700,800,900])
# select elements at index 1, 2, 5, 7
selected = a[[1, 2, 5, 7]]
selected
```

```
array([100, 200, 500, 700])
```

```
a = np.zeros((8, 4))
for i in range(8):
    a[i] = i
a
```

```
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

```
a[[4, 3, 0, 6]]
```

```
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

```
a[[-3, -5, -7]]
```

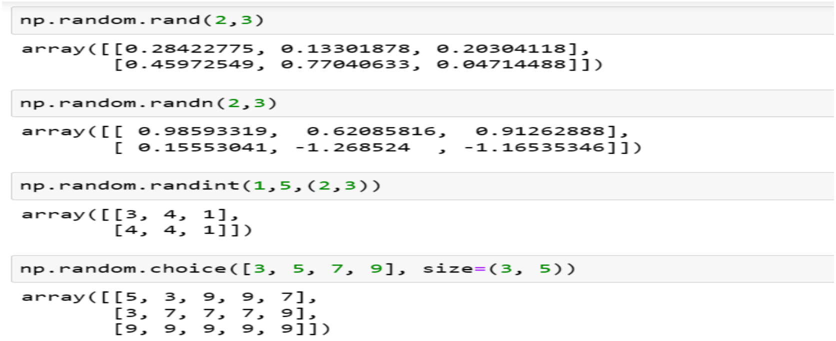
```
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

```
a[[1, 5, 7, 2], [0, 3, 1, 2]]
#Here the elements (1, 0), (5, 3), (7, 1), and (2, 2) were selected.
```

```
array([1., 5., 7., 2.])
```

Numpy random

- numpy.random.rand(d0, d1, ..., dn) Random values in a given shape.
- numpy.random.randn(d0, d1, ..., dn) Return a sample (or samples) from the "standard normal" distribution
- numpy.random.randint(low, high=None, size) Return random integers from low (inclusive) to high (exclusive).
- The choice() method takes an array as a parameter and randomly returns one of the values.



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

```
array([[0.28422775, 0.13301878, 0.20304118],
       [0.45972549, 0.77040633, 0.04714488]])
```

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

```
array([[ 0.98593319,  0.62085816,  0.91262888],
       [ 0.15553041, -1.268524  , -1.16535346]])
```

```
np.random.randint(1,5,(2,3))
```

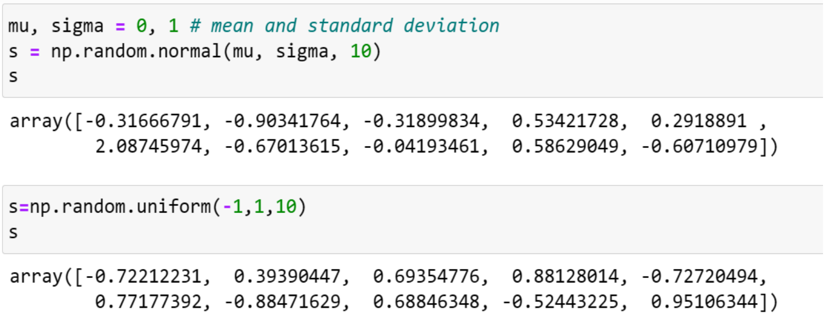
```
array([[3, 4, 1],
       [4, 4, 1]])
```

```
np.random.choice([3, 5, 7, 9], size=(3, 5))
```

```
array([[5, 3, 9, 9, 7],
       [3, 7, 7, 7, 9],
       [9, 9, 9, 9, 9]])
```

NumPy random for sampling

- for efficiently generating whole arrays of sample values from many kinds of probability distributions.
- numpy.random.uniform(low=0.0, high=1.0, size)
- numpy.random.normal(mean=0.0, standard deviation=1.0, size)



```
mu, sigma = 0, 1 # mean and standard deviation
s = np.random.normal(mu, sigma, 10)
s
```

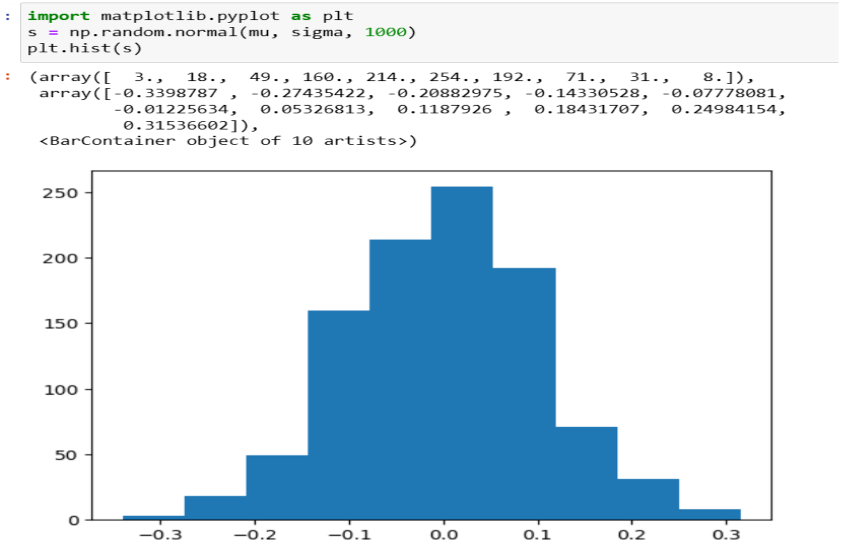
```
array([-0.31666791, -0.90341764, -0.31899834,  0.53421728,  0.2918891 ,
        2.08745974, -0.67013615, -0.04193461,  0.58629049, -0.60710979])
```

```
s=np.random.uniform(-1,1,10)
s
```

```
array([-0.72212231,  0.39390447,  0.69354776,  0.88128014, -0.72720494,
        0.77177392, -0.88471629,  0.68846348, -0.52443225,  0.95106344])
```

Use np.random.standard_normal() to create a 4 × 4 matrix, whose entries are from standard normal distributions.

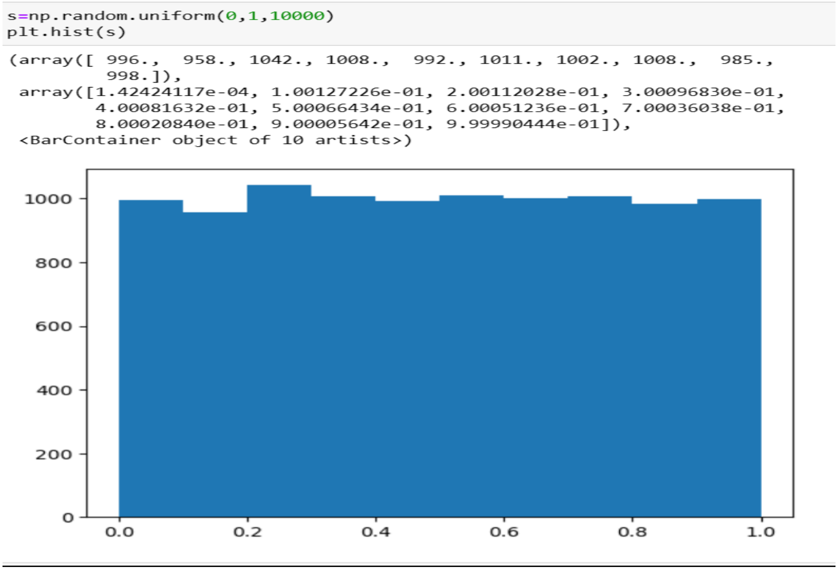
Visualizing the Normal Distribution



```
import matplotlib.pyplot as plt
s = np.random.normal(mu, sigma, 1000)
plt.hist(s)
```

```
(array([  3.,  18.,  49., 160., 214., 254., 192.,  71.,  31.,   8.]),
 array([-0.3398787 , -0.27435422, -0.20882975, -0.14330528, -0.07778081,
        -0.01225634,  0.05326813,  0.1187926 ,  0.18431707,  0.24984154,
         0.31536602]),
 <BarContainer object of 10 artists>)
```

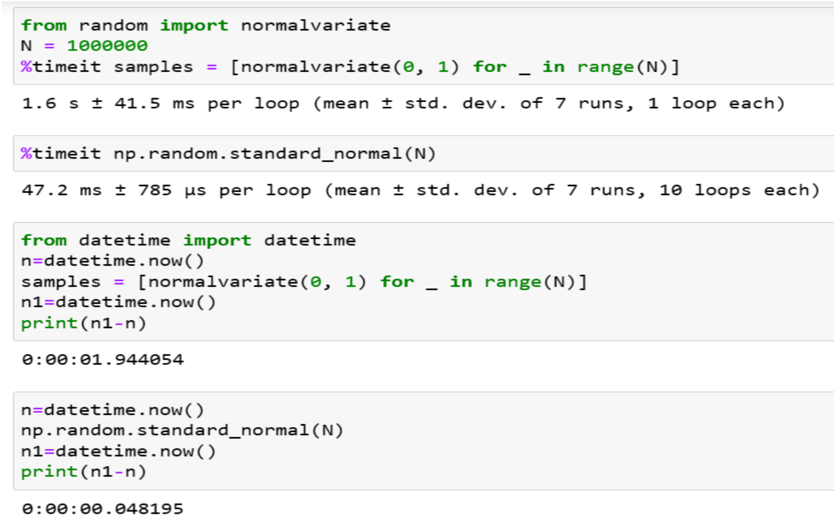
Visualizing the Uniform distribution



```
s=np.random.uniform(0,1,10000)
plt.hist(s)
```

```
(array([ 996.,  958., 1042., 1008.,  992., 1011., 1002., 1008.,  985.,
         998.]),
 array([1.42424117e-04, 1.00127226e-01, 2.00112028e-01, 3.00096830e-01,
        4.00081632e-01, 5.00066434e-01, 6.00051236e-01, 7.00036038e-01,
        8.00020840e-01, 9.00005642e-01, 9.99990444e-01]),
 <BarContainer object of 10 artists>)
```

python random vs NumPy random



```
from random import normalvariate
N = 1000000
%timeit samples = [normalvariate(0, 1) for _ in range(N)]
```

```
1.6 s ± 41.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit np.random.standard_normal(N)
```

```
47.2 ms ± 785 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
from datetime import datetime
n=datetime.now()
samples = [normalvariate(0, 1) for _ in range(N)]
n1=datetime.now()
print(n1-n)
```

```
0:00:01.944054
```

```
n=datetime.now()
np.random.standard_normal(N)
n1=datetime.now()
print(n1-n)
```

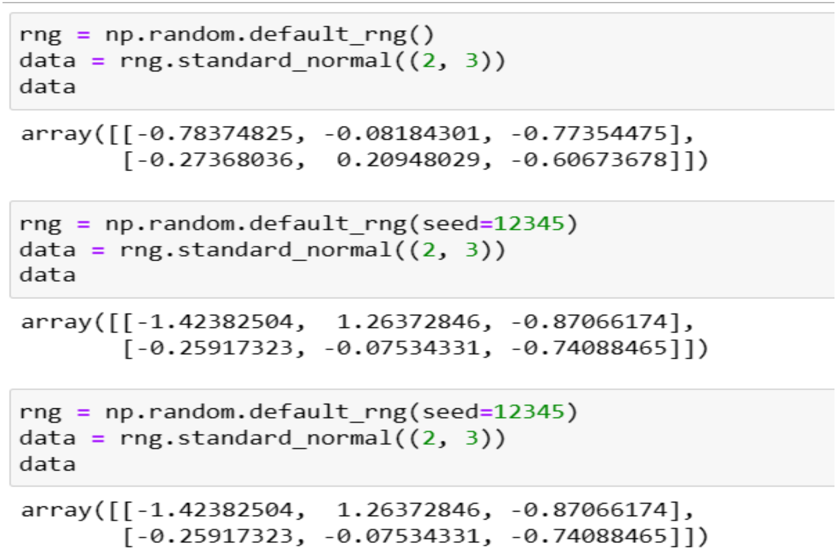
```
0:00:00.048195
```

NumPy Random Number generator

Read only taken from RealPython

- Python uses the random module, which generates numbers using the Mersenne twister algorithm. While this is still widely used in Python code, it's possible to predict the numbers that it generates, and it requires significant computing power.
- Since version 1.17, NumPy uses the more efficient permuted congruential generator-64 (PCG64) algorithm. This produces less-predictable numbers. PCG64 is also faster and requires fewer resources to work.
- PRNGs require a seed number to initialize their number generation. PRNGs that use the same seed will generate the same numbers.
- Because seeds should be random, you need one random number to generate another. For this purpose, PRNGs use the computer hardware clock's time as their default seed.

Way to Use Random Module



```
rng = np.random.default_rng()
data = rng.standard_normal((2, 3))
data
```

```
array([[-0.78374825, -0.08184301, -0.77354475],
       [-0.27368036,  0.20948029, -0.60673678]])
```

```
rng = np.random.default_rng(seed=12345)
data = rng.standard_normal((2, 3))
data
```

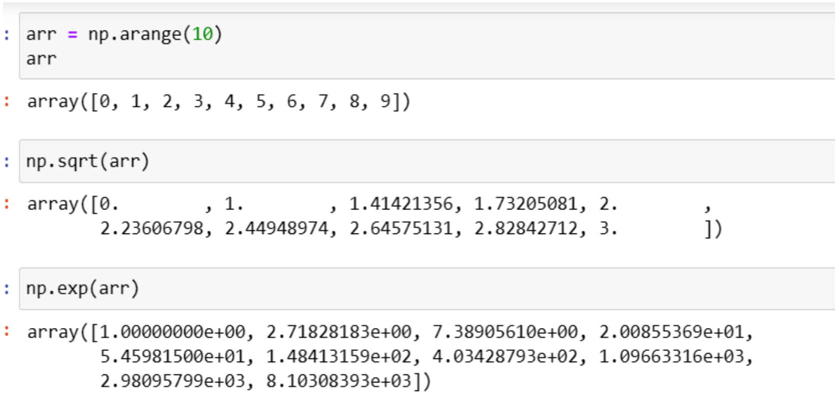
```
array([[-1.42382504,  1.26372846, -0.87066174],
       [-0.25917323, -0.07534331, -0.74088465]])
```

```
rng = np.random.default_rng(seed=12345)
data = rng.standard_normal((2, 3))
data
```

```
array([[-1.42382504,  1.26372846, -0.87066174],
       [-0.25917323, -0.07534331, -0.74088465]])
```

Universal Functions: Fast Element-Wise Array Functions

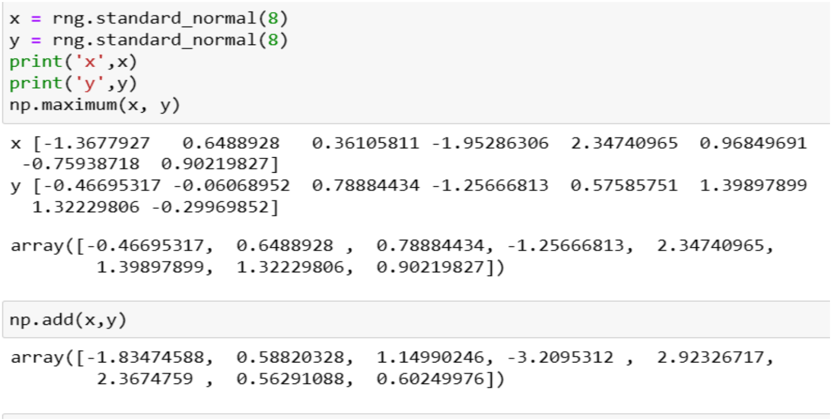
- Popularly known as *ufunc*, is a function that performs element-wise operations on data in ndarrays
- some functions take one array as their argument, These are referred to as unary ufuncs. For example exponential, square root.



```
: arr = np.arange(10)
  arr

: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

: np.sqrt(arr)

: array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
         2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])

: np.exp(arr)

: array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
         5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
         2.98095799e+03, 8.10308393e+03])
```

Ufunc

- some takes two arrays(they are called binary ufunc) as their arguments, and return a single array as the output. Like add, multiply.



```
x = rng.standard_normal(8)
y = rng.standard_normal(8)
print('x',x)
print('y',y)
np.maximum(x, y)
```

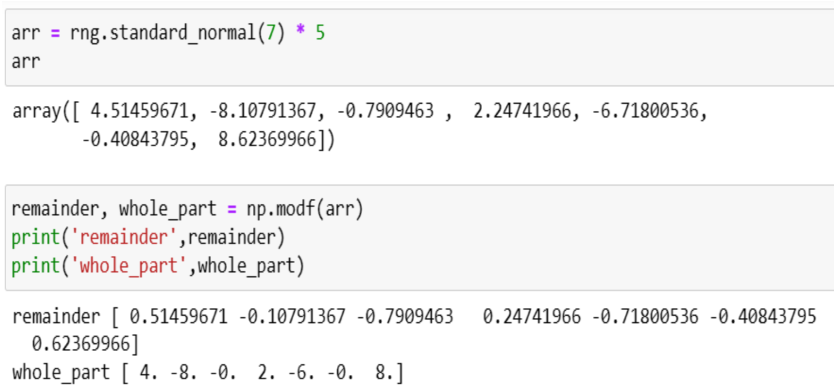
```
x [-1.3677927   0.6488928   0.36105811 -1.95286306  2.34740965  0.96849691
 -0.75938718  0.90219827]
y [-0.46695317 -0.06068952  0.78884434 -1.25666813  0.57585751  1.39897899
  1.32229806 -0.29969852]

array([-0.46695317,  0.6488928 ,  0.78884434, -1.25666813,  2.34740965,
        1.39897899,  1.32229806,  0.90219827])
```

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

```
array([-1.83474588,  0.58820328,  1.14990246, -3.2095312 ,  2.92326717,
        2.3674759 ,  0.56291088,  0.60249976])
```

Ufunc

- In somecases ufunc can return multiple arrays. numpy.modf is one example: it returns the fractional and integral parts of a floating-point array



```
arr = rng.standard_normal(7) * 5
arr
```

```
array([ 4.51459671, -8.10791367, -0.7909463 ,  2.24741966, -6.71800536,
       -0.40843795,  8.62369966])
```

```
remainder, whole_part = np.modf(arr)
print('remainder',remainder)
print('whole_part',whole_part)
```

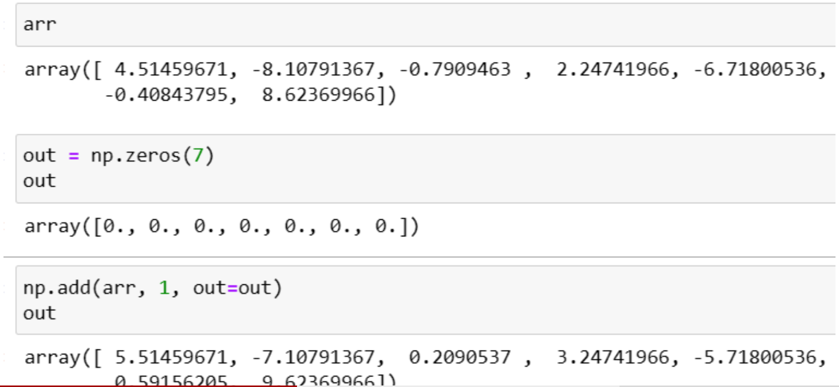
```
remainder [ 0.51459671 -0.10791367 -0.7909463   0.24741966 -0.71800536 -0.40843795
  0.62369966]
whole_part [ 4. -8. -0.  2. -6. -0.  8.]
```

Ufunc

- Ufuncs accept an optional **out** argument that allows them to assign their results(output) into an existing array rather than create a new one.

Check point

Find out 10 unary ufunc and 10 binary ufunc in NumPy.



```
arr
```

```
array([ 4.51459671, -8.10791367, -0.7909463 ,  2.24741966, -6.71800536,
       -0.40843795,  8.62369966])
```

```
out = np.zeros(7)
out
```

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

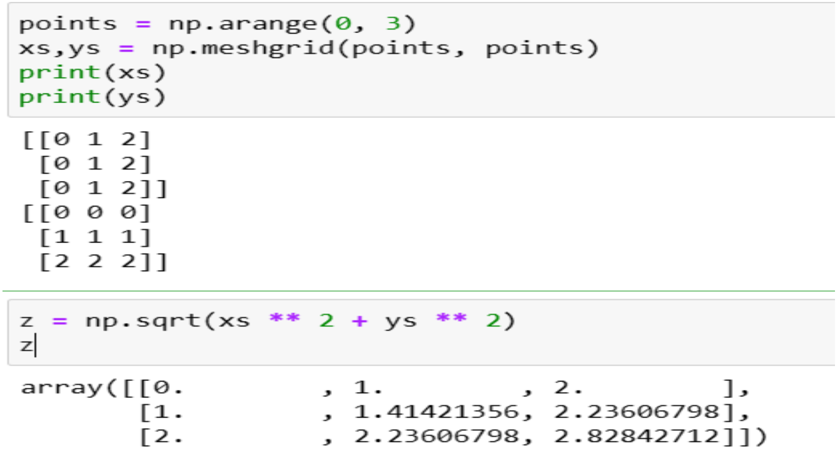
```
np.add(arr, 1, out=out)
out
```

```
array([ 5.51459671, -7.10791367,  0.2090537 ,  3.24741966, -5.71800536,
        0.59156205,  9.62369966])
```

Array-Oriented Programming with Arrays

- Meshgrid: The purpose of meshgrid is to create a rectangular grid out of an array of x values and an array of y values.
- it's primary purpose is to create a coordinates matrices.
- Suppose we have a function of two variables, $f(x,y) = \sqrt{x^2 + y^2}$, and you want to see how is the graph(function) behaving in the interval $[0,2] \times [0,2]$.
- Then you want all possible of combinations such as (0,0),(0,1)...(2,2). Here meshgrid comes into help.

Meshgrid example



```python
points = np.arange(0, 3)
xs,ys = np.meshgrid(points, points)
print(xs)
print(ys)
```

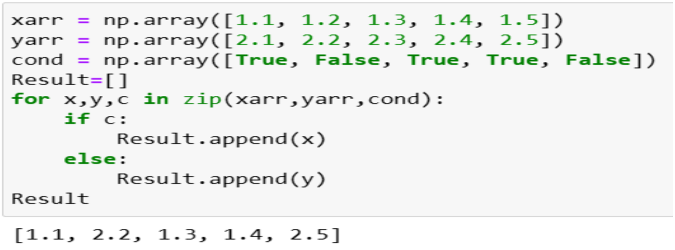
```
[[0 1 2]
 [0 1 2]
 [0 1 2]]
[[0 0 0]
 [1 1 1]
 [2 2 2]]
```

```python
z = np.sqrt(xs ** 2 + ys ** 2)
z
```

```
array([[0.        , 1.        , 2.        ],
       [1.        , 1.41421356, 2.23606798],
       [2.        , 2.23606798, 2.82842712]])
```

Expressing Conditional Logic as Array Operations

- Suppose we had a Boolean array (COND) and two arrays of values (XARR and YARR).
- Suppose we wanted to take a value from xarr whenever the corresponding value in cond is True, and otherwise take the value from yarr



```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])
Result=[]
for x,y,c in zip(xarr,yarr,cond):
     if c:
          Result.append(x)
     else:
          Result.append(y)
Result
```

```
[1.1, 2.2, 1.3, 1.4, 2.5]
```

But How we will such a problem in Multi dimension?

Conditional Logic as Array Operations

- The numpy.where function is a vectorized version of the ternary expression x if condition else y
- The second and third arguments to numpy.where don't need to be arrays; one or both of them can be scalars.

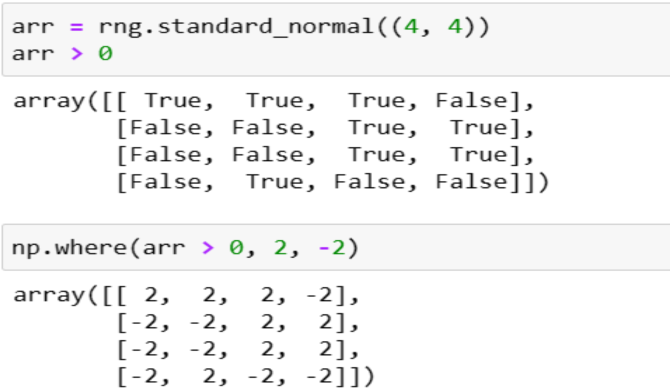
```
Result=np.where(cond,xarr,yarr)
Result
```

```
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Check Point

Create a randomly generated 4×4 matrix, where each data is from standard normal distribution. Replace all positive values with 2 and all negative values with –2.

Answer to above



```python
arr = rng.standard_normal((4, 4))
arr > 0
```

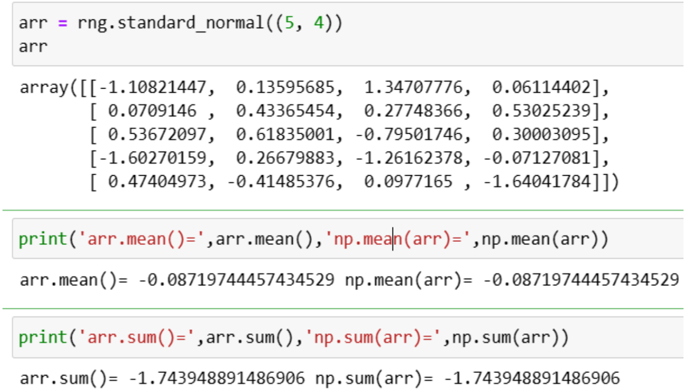
```
array([[ True,  True,  True, False],
       [False, False,  True,  True],
       [False, False,  True,  True],
       [False,  True, False, False]])
```

```python
np.where(arr > 0, 2, -2)
```

```
array([[ 2,  2,  2, -2],
       [-2, -2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2, -2, -2]])
```

Mathematical and Statistical Methods

- Some mathematical tasks can be performed by calling the array instance method or using the NumPy function



```
arr = rng.standard_normal((5, 4))
arr
```

```
array([[-1.10821447,  0.13595685,  1.34707776,  0.06114402],
       [ 0.0709146 ,  0.43365454,  0.27748366,  0.53025239],
       [ 0.53672097,  0.61835001, -0.79501746,  0.30003095],
       [-1.60270159,  0.26679883, -1.26162378, -0.07127081],
       [ 0.47404973, -0.41485376,  0.0977165 , -1.64041784]])
```

```
print('arr.mean()=',arr.mean(),'np.mean(arr)=',np.mean(arr))
```

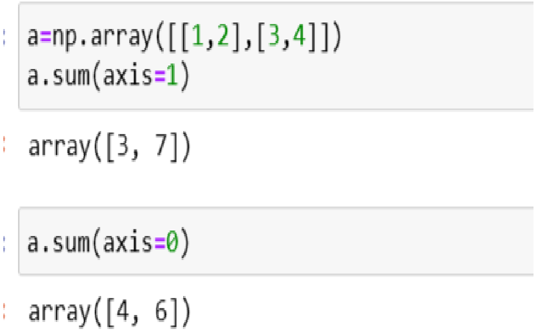
```
arr.mean()= -0.08719744457434529 np.mean(arr)= -0.08719744457434529
```

```
print('arr.sum()=',arr.sum(),'np.sum(arr)=',np.sum(arr))
```

```
arr.sum()= -1.743948891486906 np.sum(arr)= -1.743948891486906
```

Mathematical Methods

- Functions like mean and sum take an optional axis argument that computes the statistic over the given axis, resulting in an array with one less dimension
- axis=1 means find sums across columns, and axis=0 means find sum over rows.



```
a=np.array([[1,2],[3,4]])
a.sum(axis=1)
```

```
array([3, 7])
```

```
a.sum(axis=0)
```

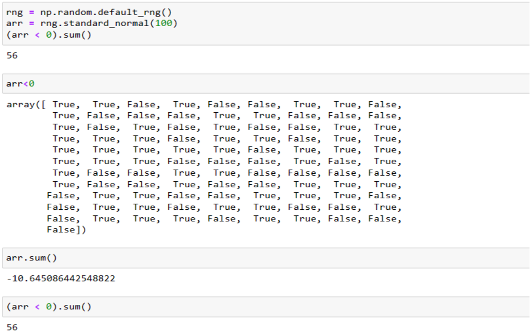
```
array([4, 6])
```

Check point

Explore statistical methods like mean, std, var, min, max

Methods for Boolean Arrays

- sum() is often used as a means of counting True values in a Boolean array



```
rng = np.random.default_rng()
arr = rng.standard_normal(100)
(arr < 0).sum()
```

```
56
```

```
arr<0
```

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

```
arr.sum()
```

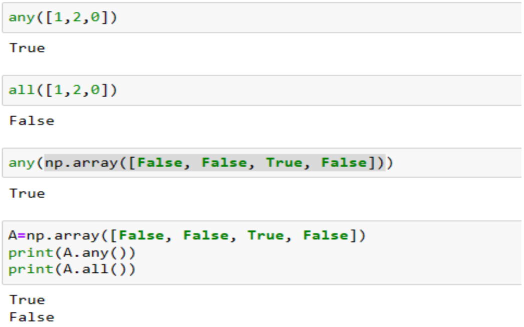
```
-10.645086442548822
```

```
(arr < 0).sum()
```

```
56
```

any all methods

- any() tests whether one or more values in an array is True
- all() checks if every value is True



```
any([1,2,0])
```

```
True
```

```
all([1,2,0])
```

```
False
```

```
any(np.array([False, False, True, False]))
```

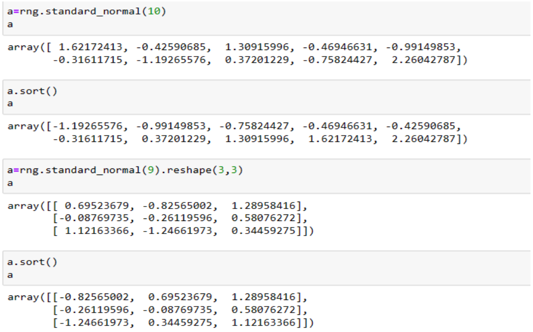
```
True
```

```
A=np.array([False, False, True, False])
print(A.any())
print(A.all())
```

```
True
False
```

Sorting

- NumPy arrays can be sorted using the sort method.
- We can sort each one-dimensional section of values in a multidimensional array in place along an axis by passing the axis number to sort



```
a=rng.standard_normal(10)
a
```

```
array([ 1.62172413, -0.42590685,  1.30915996, -0.46946631, -0.99149853,
       -0.31611715, -1.19265576,  0.37201229, -0.75824427,  2.26042787])
```

```
a.sort()
a
```

```
array([-1.19265576, -0.99149853, -0.75824427, -0.46946631, -0.42590685,
       -0.31611715,  0.37201229,  1.30915996,  1.62172413,  2.26042787])
```

```
a=rng.standard_normal(9).reshape(3,3)
a
```

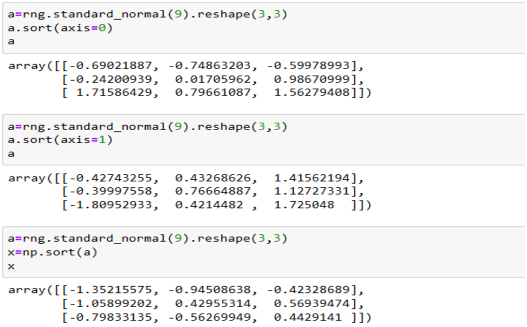
```
array([[ 0.69523679, -0.82565002,  1.28958416],
       [-0.08769735, -0.26119596,  0.58076272],
       [ 1.12163366, -1.24661973,  0.34459275]])
```

```
a.sort()
a
```

```
array([[-0.82565002,  0.69523679,  1.28958416],
       [-0.26119596, -0.08769735,  0.58076272],
       [-1.24661973,  0.34459275,  1.12163366]])
```

Sorting

- arr.sort(axis=0) sorts the values within each column, while arr.sort(axis=1) sorts across each row
- numpy.sort() returns a sorted copy of an array (like the Python built-in function sorted) instead of modifying the original array.



```
a=rng.standard_normal(9).reshape(3,3)
a.sort(axis=0)
a
```

```
array([[-0.69021887, -0.74863203, -0.59978993],
       [-0.24200939,  0.01705962,  0.98670999],
       [ 1.71586429,  0.79661087,  1.56279408]])
```

```
a=rng.standard_normal(9).reshape(3,3)
a.sort(axis=1)
a
```

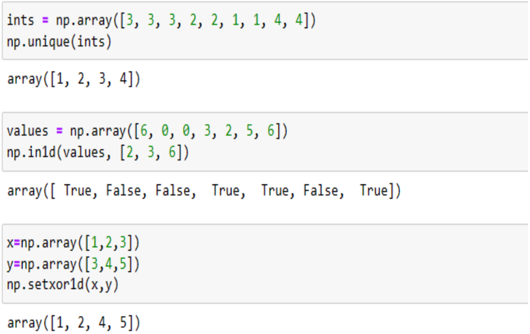
```
array([[-0.42743255,  0.43268626,  1.41562194],
       [-0.39997558,  0.76664887,  1.12727331],
       [-1.80952933,  0.4214482 ,  1.725048  ]])
```

```
a=rng.standard_normal(9).reshape(3,3)
x=np.sort(a)
x
```

```
array([[-1.35215575, -0.94508638, -0.42328689],
       [-1.05899202,  0.42955314,  0.56939474],
       [-0.79833135, -0.56269949,  0.4429141 ]])
```

Unique values

- numpy.unique(arr), which returns the sorted unique values in an array
- numpy.in1d(arr1,arr2), tests membership of the values in arr1 in arr2, returning a Boolean array
- Explore other set methods likes intersect1d(x, y), union1d(x, y), setdiff1d(x, y), setxor1d(x, y): Set symmetric differences



```
ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
np.unique(ints)
```

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

```
values = np.array([6, 0, 0, 3, 2, 5, 6])
np.in1d(values, [2, 3, 6])
```

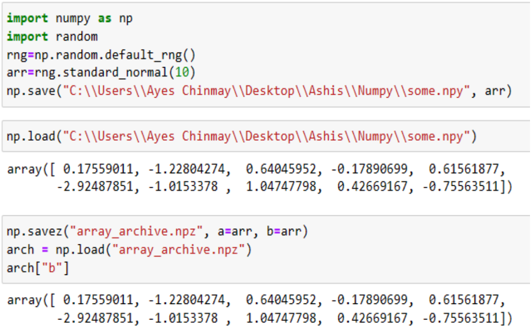
```
array([ True, False, False,  True,  True, False,  True])
```

```
x=np.array([1,2,3])
y=np.array([3,4,5])
np.setxor1d(x,y)
```

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

File handling

- np.save(name of the file, contents)
- np.load(name of the file)
- You can save multiple arrays in an uncompressed archive using numpy.savez and passing the arrays as keyword arguments
- When loading an .npz file, you get back a dictionary-like object that loads the individual arrays.
- Instead of using numpy.savez we can also use numpy.savez_compressed



```
import numpy as np
import random
rng=np.random.default_rng()
arr=rng.standard_normal(10)
np.save("C:\\Users\\Ayes Chinmay\\Desktop\\Ashis\\Numpy\\some.npy", arr)
```

```
np.load("C:\\Users\\Ayes Chinmay\\Desktop\\Ashis\\Numpy\\some.npy")
```

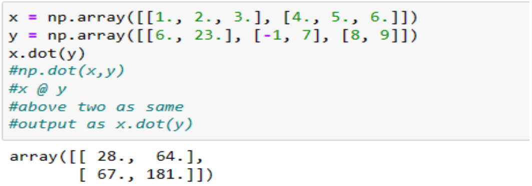
```
array([ 0.17559011, -1.22804274,  0.64045952, -0.17890699,  0.61561877,
       -2.92487851, -1.0153378 ,  1.04747798,  0.42669167, -0.75563511])
```

```
np.savez("array_archive.npz", a=arr, b=arr)
arch = np.load("array_archive.npz")
arch["b"]
```

```
array([ 0.17559011, -1.22804274,  0.64045952, -0.17890699,  0.61561877,
       -2.92487851, -1.0153378 ,  1.04747798,  0.42669167, -0.75563511])
```

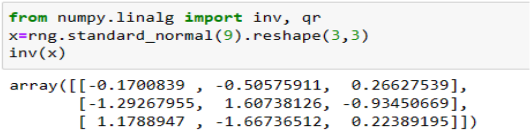
Linear Algebra

- We have seen the use of "*" between matrices is for element wise product.
- matrix multiplications uses the dot function. Either x.dot(y) or np.dot(x, y). Similar work is also done by @ operator.
- numpy.linalg has a standard set of matrix decomposition's and things like inverse and determinant



```
x = np.array([[1., 2., 3.], [4., 5., 6.]])
y = np.array([[6., 23.], [-1, 7], [8, 9]])
x.dot(y)
#np.dot(x,y)
#x @ y
#above two as same
#output as x.dot(y)
```

```
array([[ 28.,  64.],
       [ 67., 181.]])
```



```
from numpy.linalg import inv, qr
x=rng.standard_normal(9).reshape(3,3)
inv(x)
```

```
array([[-0.1700839 , -0.50575911,  0.26627539],
       [-1.29267955,  1.60738126, -0.93450669],
       [ 1.1788947 , -1.66736512,  0.22389195]])
```

Other functions in linear Algebra

- Create examples for other functions in linear algebra

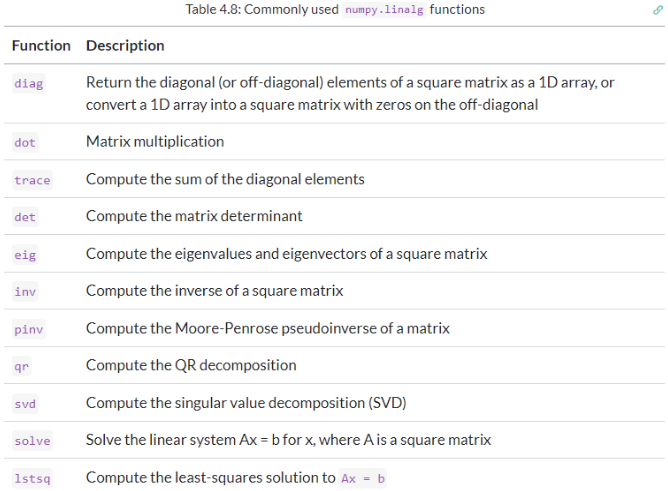


Table 4.8: Commonly used `numpy.linalg` functions

Function	Description
`diag`	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
`dot`	Matrix multiplication
`trace`	Compute the sum of the diagonal elements
`det`	Compute the matrix determinant
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudoinverse of a matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system Ax = b for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to `Ax = b`

Random Walk

- Let's first consider a simple random walk. starting at 0 with steps of 1 and -1 occurring with equal probability.
- Imagine tossing a coin if head comes I win 1 rupee from you and if tail I give you 1rupee.
- see after 1000 tosses what is my gain(or loss).

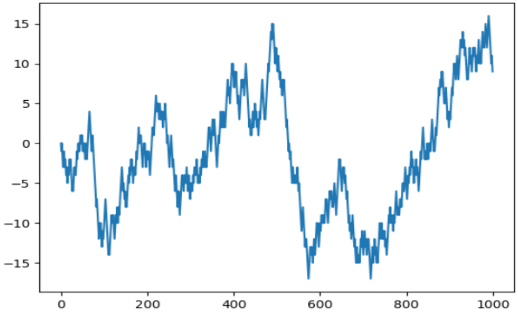
Random Walk

- Let's first consider a simple random walk. starting at 0 with steps of 1 and −1 occurring with equal probability.
- Imagine tossing a coin if head comes I win 1 rupee from you and if tail I give you 1rupee.
- see after 1000 tosses what is my gain(or loss).
- Hint
    - Initiate start=0, and walk to be list containing start and the cumulative sums
    - randomly generate an integer between -1 or 1.
    - update the start, that is the cumulative sum
    - Append the cumulative sums to walk
    - print the walk

Random walk

```
import random
import matplotlib.pyplot as plt
position = 0
walk = [position]
nsteps = 1000
for _ in range(nsteps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
plt.plot(walk[:1000])
```

```
[<matplotlib.lines.Line2D at 0x240214bbc10>]
```

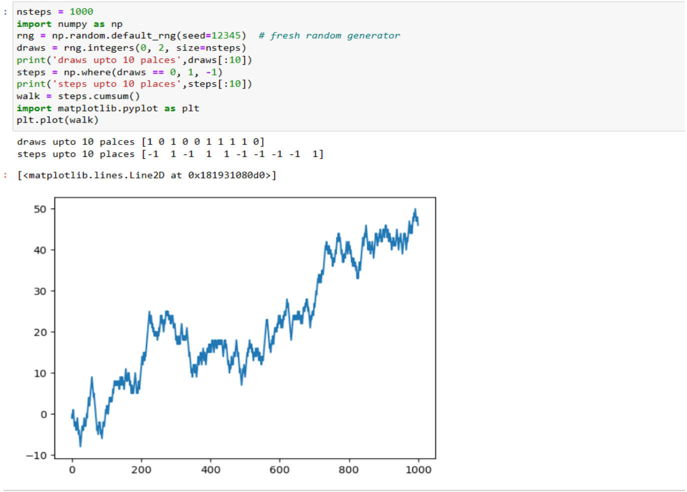


Random Walk using numpy

Let us try to do the above problem using numpy.

- Let us generate 100 random integers in the range of 0 and 1 store it to draws.
- Replace 1s with 0s and 0s with 1s in the draws array and store it to steps.
- define another array named walks, which is the cumulative sum of steps.
- Plot the walks array.

Numpy Random Walk



```
nsteps = 1000
import numpy as np
rng = np.random.default_rng(seed=12345)  # fresh random generator
draws = rng.integers(0, 2, size=nsteps)
print('draws upto 10 palces',draws[:10])
steps = np.where(draws == 0, 1, -1)
print('steps upto 10 places',steps[:10])
walk = steps.cumsum()
import matplotlib.pyplot as plt
plt.plot(walk)
```

```
draws upto 10 palces [1 0 1 0 0 1 1 1 1 0]
steps upto 10 places [-1  1 -1  1  1 -1 -1 -1 -1  1]
```

```
[<matplotlib.lines.Line2D at 0x181931080d0>]
```