



# Python NumPy

## NumPy

- Numeric Python
- Python library used for working with arrays.
- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

### ✓ 01. Import NumPy

```
import numpy as np
```

```
#Check Numpy version  
np.__version__
```

```
1.26.4
```

### ✓ 02. Create Arrays

```
#Convert List to array
```

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

```
print(type(A))
```

```
arr=np.array(A)
```

```
print(arr)  
print(type(arr))
```

```
<class 'list'>  
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]  
<class 'numpy.ndarray'>
```

#### 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.

This behavior is called **locality of reference** in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

```
np.arange(3.0)
```

```
→ array([0., 1., 2.])
```

```
# np.arange([start, ]stop, [step, ]dtype=None, *, device=None, like=None)
# start<stop
```

```
print(np.arange(15))
#[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

```
print(np.arange(3.0))
#[0. 1. 2.]
```

```
print(np.arange(3,7))
#[3 4 5 6] Starts with 3 to 7-1
```

```
print(np.arange(1,9,3))
#[1 4 7] Starts with 1 to 9 with step 3
```

```
→ [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
   [0. 1. 2.]
   [3 4 5 6]
   [1 4 7]
```

```
np.arange(-10,10,2)
```

```
→ array([-10, -8, -6, -4, -2,  0,  2,  4,  6,  8])
```

```
np.arange(1+1j,9+9j,2)
```

```
→ array([1.+1.j, 3.+1.j, 5.+1.j, 7.+1.j])
```

```
# 0D Array
A0=np.array(42)
print(A0)
print(type(A0))
print(A0.ndim, 'D')
print('\n')
```

```
# 1D Array
A1=np.array([1,2,3,4])
print(A1)
print(type(A1))
print(A1.ndim, 'D')
print('\n')
```

```
# 2D Array
A2=np.array([[1,2,3,4],[5,6,7,8]])
print(A2)
print(type(A2))
print(A2.ndim, 'D')
print('\n')
```

```
# 3D Array
A3=np.array([[[1,2,3],[4,5,6],[7,8,9]],[[10,11,12],[13,14,15],[16,17,18]],[[19,20,21],[22,23,24],[25,26,27]]])
print(A3)
print(type(A3))
print(A3.ndim, 'D')
```

```
→ 42
   <class 'numpy.ndarray'>
   0 D
```

```
[1 2 3 4]
<class 'numpy.ndarray'>
1 D
```

```
[[1 2 3 4]
 [5 6 7 8]]
<class 'numpy.ndarray'>
2 D
```

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

```
[ 4  5  6]
[ 7  8  9]]

[[10 11 12]
 [13 14 15]
 [16 17 18]]

[[19 20 21]
 [22 23 24]
 [25 26 27]]]
<class 'numpy.ndarray'>
3 D
```

```
# np.zeros(shape, dtype=float, order='C', *, like=None)
```

```
print(np.zeros(5,dtype=int))
#[0, 0, 0, 0, 0]
print('\n')
```

```
print(np.zeros(3,dtype=float))
#[0., 0., 0.]
print('\n')
```

```
print(np.zeros([2,3],dtype=int))
#[[0, 0, 0],
# [0, 0, 0]]
print('\n')
```

```
print(np.zeros([2,3,4],dtype=int))
```

```
↩ [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 0 0 0]]]
```

```
n=(2,2)
n1=(4,4)
```

```
print(np.zeros(n,dtype=int))
print('\n')
print(np.zeros(n1,dtype=float))
```

```
↩ [[0 0]
   [0 0]]
```

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

```
#np.ones(shape, dtype=None, order='C', *,device=None, like=None)
```

```
print(np.ones(5,dtype=int))
#[0, 0, 0, 0, 0]
print('\n')
print(np.ones(3,dtype=float))
#[1. 1. 1.]
print('\n')
print(np.ones([2,3],dtype=int))
```

```
↩ [1 1 1 1 1]
```

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

```
[[1 1 1]
 [1 1 1]]
```

```
n=(2,2)
n1=(4,4)
```

```
print(np.ones(n,dtype=int))
print('\n')
print(np.ones(n1,dtype=float))
```

```
↔ [[1 1]
    [1 1]]
```

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

# The shape can be provided either as a tuple (2, 3) or as a list [2, 3]

```
n1=(2,2) # tuple
n2=[2,2] # list
```

```
print(np.ones(n1,dtype=int))
print('\n')
print(np.ones(n2,dtype=int))
print('\n')
print(np.ones([3,3],dtype=int))
print('\n')
print(np.ones((3,3),dtype=int))
```

```
↔ [[1 1]
    [1 1]]
```

```
[[1 1]
 [1 1]]
```

```
[[1 1 1]
 [1 1 1]
 [1 1 1]]
```

```
[[1 1 1]
 [1 1 1]
 [1 1 1]]
```

```
# linspace-
# also called as Linearly space.
# Linearly separable in a given range at equal distance it creates points

# np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

```
print(np.linspace(1,5,5))
print('\n')
print(np.linspace(-10,10,5))
```

```
↔ [1. 2. 3. 4. 5.]
```

```
[-10. -5. 0. 5. 10.]
```

```
# Identity matrix
# np.identity(n,dtype=int)

print(np.identity(3,dtype=int))
print('\n')
print(np.identity(5))
print('\n')
print(np.identity(7,dtype=int))
```

```
↵ [[1 0 0]
   [0 1 0]
   [0 0 1]]
```

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

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

```
from numpy import random
```

```
#randint(low, high=None, size=None, dtype=int)
# Generate a random inter from 0 to 100
print(random.randint(100))
```

```
print('\n')
```

```
# rand(d0, d1, ..., dn)
# Generate a random float from 0 to 1
print(random.rand())
```

```
↵ 6
```

```
0.5104582090054129
```

```
# Generate a 1-D array containing 5 random integers from 0 to 100
print(random.randint(100, size=(5)))
```

```
print('\n')
```

```
# Generate a 2-D array with 3 rows, each row containing 5 random integers from 0 to 100
print(random.randint(100, size=(3,5)))
```

```
print('\n')
```

```
# Generate a 1-D array containing 5 random floats
print(random.rand(5))
```

```
print('\n')
```

```
# Generate a 2-D array with 3 rows, each row containing 5 random numbers
print(random.rand(3,5))
```

```
↵ [38 34 40 17 41]
```

```
[[31 83 55 37 39]
 [88 82 89 45 26]
 [52 88 48 92 80]]
```

```
[0.66889297 0.50057476 0.98407845 0.02101914 0.0166311 ]
```

```
[[0.81808709 0.6368088 0.84853468 0.47809559 0.3826904 ]
 [0.70964462 0.12782407 0.25329329 0.49406612 0.73285376]
 [0.87262205 0.4255038 0.71689912 0.76260816 0.53127176]]
```

```
import random
```

```
# prints a random value from the list
list_1 = [1, 2, 3, 4, 5, 6]
print(np.random.choice(list_1))
print(np.random.choice(list_1,size=(2,2)))
```

```
# prints a random item from the string
string = "World"
print(random.choice(string))
```

```
↩ 2
[[6 5]
 [6 6]]
o
```

### ✓ 03. Array Operations

```
# Basic Arithmetic operations
```

```
A=[[2,4,6,8],[10,12,14,16],[18,20,22,24],[26,28,30,32]]
B=[[4,8,12,16],[20,24,28,32],[36,40,44,48],[52,56,60,64]]
```

```
A=np.array(A)
B=np.array(B)
```

```
print(type(A))
print(type(B))
```

```
print('\n')
```

```
# Array Addition
print('Addition',end='\n')
print(A+B)
```

```
print('\n')
```

```
# Array Subtraction
print('Subtraction',end='\n')
print(B-A)
```

```
print('\n')
```

```
# Array Multiplication
print('Multiplication',end='\n')
print(A*B)
```

```
print('\n')
```

```
# Array Division
print('Division',end='\n')
print(B/A)
```

```
↩ <class 'numpy.ndarray'>
  <class 'numpy.ndarray'>
```

```
Addition
[[ 6 12 18 24]
 [30 36 42 48]
 [54 60 66 72]
 [78 84 90 96]]
```

```
Subtraction
[[ 2  4  6  8]
 [10 12 14 16]
 [18 20 22 24]
 [26 28 30 32]]
```

```
Multiplication
[[  8  32  72 128]
 [200 288 392 512]
 [648 800 968 1152]
 [1352 1568 1800 2048]]
```

```
Division
[[2.  2.  2.  2.]
 [2.  2.  2.  2.]
 [2.  2.  2.  2.]
 [2.  2.  2.  2.]]
```

```
# Element-wise operations
```

```
print(2+A)
print('\n')
print(3*A)
print('\n')
print(100-A)
print('\n')
print(B/2)
print('\n')
print(np.square(A))
print('\n')
print(np.sqrt(B))
print('\n')
print(np.exp(A))
```

```
→ [[ 4  6  8 10]
    [12 14 16 18]
    [20 22 24 26]
    [28 30 32 34]]
```

```
[[ 6 12 18 24]
 [30 36 42 48]
 [54 60 66 72]
 [78 84 90 96]]
```

```
[[98 96 94 92]
 [90 88 86 84]
 [82 80 78 76]
 [74 72 70 68]]
```

```
[[ 2.  4.  6.  8.]
 [10. 12. 14. 16.]
 [18. 20. 22. 24.]
 [26. 28. 30. 32.]]
```

```
[[ 4  16  36  64]
 [100 144 196 256]
 [324 400 484 576]
 [676 784 900 1024]]
```

```
[[2.          2.82842712 3.46410162 4.          ]
 [4.47213595 4.89897949 5.29150262 5.65685425]
 [6.          6.32455532 6.63324958 6.92820323]
 [7.21110255 7.48331477 7.74596669 8.          ]]
```

```
[[7.38905610e+00 5.45981500e+01 4.03428793e+02 2.98095799e+03]
 [2.20264658e+04 1.62754791e+05 1.20260428e+06 8.88611052e+06]
 [6.56599691e+07 4.85165195e+08 3.58491285e+09 2.64891221e+10]
 [1.95729609e+11 1.44625706e+12 1.06864746e+13 7.89629602e+13]]
```

```
# Dot Product
```

```
np.dot(A,B)
```

```
→ array([[ 720,  800,  880,  960],
          [1616, 1824, 2032, 2240],
          [2512, 2848, 3184, 3520],
          [3408, 3872, 4336, 4800]])
```

## 04. Array Manipulation

```
# Shape
A=[[2,4,6,8],[10,12,14,16],[18,20,22,24],[26,28,30,32]]
A=np.array(A)
```

```
print(A.shape)
```

```
→ (4, 4)
```

```
# Reshape

# reshape 1D to 2D

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

print(arr.reshape(3,3))
print('\n')
#print(arr.reshape(2,3))

#Reshape is done only when the elements fullfill the criteria of rows and col mentioned

#Reshape from 1D to 3D
arr1=np.array([1,2,3,4,5,6,7,8,9,10,11,12])
print(arr1.reshape(2,3,2))
```

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

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

```
[[ 7  8]
 [ 9 10]
 [11 12]]]
```

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

```
print(arr.reshape(3,3,order='C')) # C-Type
# C-order (Row-major): This reshapes arr in row-major order (C-style), meaning it fills the new shape row by row.
print('\n')
```

```
print(arr.reshape(3,3,order='F'))
# F-order (Column-major or Fortran-style): This reshapes arr in column-major order, filling elements column by column.
print('\n')
```

```
print(arr.reshape(3,3,order='A'))
# A-order (Arbitrary): The behavior of this order depends on the memory layout of arr. If arr is stored in C-order, it acts like C; if store
```

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

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

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

```
# Transpse
```

```
print(A)
print('\n')
print(A.T)
```

```
↔ [[ 2  4  6  8]
    [10 12 14 16]
    [18 20 22 24]
    [26 28 30 32]]
```

```
[[ 2 10 18 26]
 [ 4 12 20 28]
 [ 6 14 22 30]
 [ 8 16 24 32]]
```

```
# Flatten
```

```
print(A)
```



```
print('\n')
print(A.flatten())
```

```
[[ 2  4  6  8]
 [10 12 14 16]
 [18 20 22 24]
 [26 28 30 32]]
```

```
[ 2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32]
```

## 05. Statistical Operations

```
print(A)
```

```
print("Max value:",A.max())
print("Min value:",A.min())
print("Sum value:",A.sum())
print("Mean value:",A.mean()) #avg
print("Std value:",A.std())
print("Median value:",A.median())
```

```
[[ 2  4  6  8]
 [10 12 14 16]
 [18 20 22 24]
 [26 28 30 32]]
```

```
Max value: 32
Min value: 2
Sum value: 272
Mean value: 17.0
Std value: 9.219544457292887
```

```
AttributeError                                Traceback (most recent call last)
<ipython-input-32-e3e392763da3> in <cell line: 8>()
      6 print("Mean value:",A.mean()) #avg
      7 print("Std value:",A.std())
----> 8 print("Median value:",A.median())
```

```
AttributeError: 'numpy.ndarray' object has no attribute 'median'
```

Next steps: [Explain error](#)

```
# Without importing import* we cant get median, mode
```

```
from numpy import *
```

```
print("Median value:",median(A))
```

```
Median value: 17.0
```

```
print(A)
```

```
print("Max value:",np.max(A))
print("Min value:",np.min(A))
print("Sum value:",np.sum(A))
print("Mean value:",np.mean(A)) #avg
print("Std value:",np.std(A))
print("Median value:",np.median(A))
print("25th Percentile value:",np.percentile(A,25)) #finds the 25th percentile
print("Product value:",np.prod(A))
print("Variance:", np.var(A))
print("Cumulative Sum:", np.cumsum(A))
print("Cumulative Product:", np.cumprod(A))
```

```
[[ 2  4  6  8]
 [10 12 14 16]
 [18 20 22 24]
 [26 28 30 32]]
```

```
Max value: 32
Min value: 2
Sum value: 272
Mean value: 17.0
Std value: 9.219544457292887
Median value: 17.0
```

```

25th Percentile value: 9.5
Product value: 1371195958099968000
Variance: 85.0
Cumulative Sum: [ 2  6 12 20 30 42 56 72 90 110 132 156 182 210 240 272]
Cumulative Product: [ 2 384 3840 46080 645120 10321920 185794560 3715891200 81749606400 1961990553600 51011754393600 1428329123020800 42849873690624000 1371195958099968000]

```

## 06. Indexing and Slicing

```

A=[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
A=np.array(A)
print(A)

```

```

print(A[3][1])
print(A[-1][-1])

```

```

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
14
16

```

```

print(A[::-1]) # Reverse the array
print('\n')
print(A[::-2]) # last line then prints -2 leaving the printed line
print('\n')
print(A[::-3]) # last line then prints -3 leaving the printed line
print('\n')
print(A[::-4])

```

```

[[13 14 15 16]
 [ 9 10 11 12]
 [ 5  6  7  8]
 [ 1  2  3  4]]

```

```

[[13 14 15 16]
 [ 5  6  7  8]]

```

```

[[13 14 15 16]
 [ 1  2  3  4]]

```

```

[[13 14 15 16]]

```

```

print(A[:-2]) # Index 0 to -2-1
print('\n')
print(A[:-3]) # Index 0 to -3-1

```

```

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

```

```

[[1 2 3 4]]

```

```

# Create a random matrix between 1 to 20 in 5*4 dim

```

```

mat=np.arange(1,21).reshape(5,4)
print(mat)

```

```

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

```

```
mat_1=np.arange(0,100).reshape(10,10)
print(mat_1)
```

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]
 [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]]
```

```
mat_1[:,2]
```

```
array([ 2, 12, 22, 32, 42, 52, 62, 72, 82, 92])
```

```
mat_1[3,:]
```

```
array([30, 31, 32, 33, 34, 35, 36, 37, 38, 39])
```

```
mat_1[0:10:3]
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

```
mat_1[2:6,2:4] # 1 to 5 row, 1 to 3 col
```

```
array([[22, 23],
       [32, 33],
       [42, 43],
       [52, 53]])
```

## ✓ 07. Logical operations

```
# Masking
```

```
mat_1<50
```


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

```
mat_1[mat_1<50]
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49])
```

```
mat_2=mat_1[mat_1<50]

print(mat_2)
print(type(mat_2))
```

 [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23  
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47  
48 49]  
<class 'numpy.ndarray'>


```
#1D
import numpy as np
arr=np.array([1,2,3,4,5,6])
```

```
for i in arr:
    print(i)
```

 1  
2  
3  
4  
5  
6


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

```
for i in arr:
    print(i)
```

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

```
#3D
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for i in arr:
    print(i)
print('\n')
#print each element of 3D array
```


```
for x in arr:
    for y in x:
        for z in y:
            print(z)
```

 [[1 2 3]  
[4 5 6]]  
[[ 7 8 9]  
[10 11 12]]

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12

```
# nditer
```

```
for x in np.nditer(arr):
    print(x)
```

 1  
2  
3  
4  
5  
6  
7  
8

```
9
10
11
12
```

## 08. Broadcasting

```
print(A)
print('\n')
```

```
print(A+5)
```

```
↔ [[ 1  2  3  4]
    [ 5  6  7  8]
    [ 9 10 11 12]
    [13 14 15 16]]
```

```
[[ 6  7  8  9]
 [10 11 12 13]
 [14 15 16 17]
 [18 19 20 21]]
```

## 09. Concatenation

- Concatenate a sequence of arrays along an existing axis.

```
print(A)
print('\n')
print(B)
print('\n')
```

```
↔ [[ 1  2  3  4]
    [ 5  6  7  8]
    [ 9 10 11 12]
    [13 14 15 16]]
```

```
[[ 4  8 12 16]
 [20 24 28 32]
 [36 40 44 48]
 [52 56 60 64]]
```

```
print(np.concatenate((A,B),axis=0))
print('\n')
print(np.concatenate((A,B),axis=1))
print('\n')
print(np.concatenate((A,B),axis=None))
```

```
↔ [[ 1  2  3  4]
    [ 5  6  7  8]
    [ 9 10 11 12]
    [13 14 15 16]
    [ 4  8 12 16]
    [20 24 28 32]
    [36 40 44 48]
    [52 56 60 64]]
```

```
[[ 1  2  3  4  4  8 12 16]
 [ 5  6  7  8 20 24 28 32]
 [ 9 10 11 12 36 40 44 48]
 [13 14 15 16 52 56 60 64]]
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16  4  8 12 16 20 24 28 32
 36 40 44 48 52 56 60 64]
```

```
print(np.concatenate((A,B.T),axis=1))
```

```

[[ 1  2  3  4  4 20 36 52]
 [ 5  6  7  8  8 24 40 56]
 [ 9 10 11 12 12 28 44 60]
 [13 14 15 16 16 32 48 64]]

```

## 10. Stacking

- Used for joining multiple NumPy arrays. Unlike, concatenate(), it joins arrays along a new axis. It returns a NumPy array.
- to join 2 arrays, they must have the same shape and dimensions. (e.g. both (2,3)→ 2 rows,3 columns)
- stack() creates a new array which has 1 more dimension than the input arrays. If we stack 2 1-D arrays, the resultant array will have 2 dimensions.

```

# input array
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Stacking 2 1-d arrays
c = np.stack((a, b),axis=0)
print(c)
print('\n')
print('Shape of a:',a.shape)
print('Shape of b:',b.shape)
print('Shape of c:',c.shape)

```

```

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

```

```

Shape of a: (3,)
Shape of b: (3,)
Shape of c: (2, 3)

```

## 11. Linear Algebra

- Rank, Determinant, Trace, etc of an array.
- Eigen Values of matrices
- Matrix and vector products(dot, inner, outer, etc. product), matrix exponential.
- solve linear or tensor equations and much more!

```

A = np.array([[6, 1, 1],[4, -2, 5],[2, 8, 7]])
A

```

```

array([[ 6,  1,  1],
       [ 4, -2,  5],
       [ 2,  8,  7]])

```

```

# Determinant of a Matrix
det=np.linalg.det(A)
det

```

```

-306.0

```

```

# Inverse of matrix A
print('\nInverse of A:\n',np.linalg.inv(A))

```

```

Inverse of A:
[[ 0.17647059 -0.00326797 -0.02287582]
 [ 0.05882353 -0.13071895  0.08496732]
 [-0.11764706  0.1503268  0.05228758]]

```

```

# Eigen Values and Eigen Vectors
eigenvalues,eigenvectors=np.linalg.eig(A)
print('Eigen Values:\n',eigenvalues)
print('\n')
print('Eigen Vectors:\n',eigenvectors)

```

```

↳ Eigen Values:
  [11.24862343  5.09285054 -5.34147398]

```

```

Eigen Vectors:
[[ 0.24511338  0.75669314  0.02645665]
 [ 0.40622202 -0.03352363 -0.84078293]
 [ 0.88028581 -0.65291014  0.54072554]]

```

```

# Rank of matrix A
print("Rank of A:", np.linalg.matrix_rank(A))

```

```

↳ Rank of A: 3

```

```

# Trace of matrix A
print('\nTrace of A:', np.trace(A))

```

```

↳ Trace of A: 11

```

```

# Power of matrix A
print('\nMatrix A raised to power 3:', np.linalg.matrix_power(A,3))

```

```

↳ Matrix A raised to power 3: [[336 162 228]
 [406 162 469]
 [698 702 905]]

```

```

# Solving Linear Equation
# Solve Ax=b for x

```

```

A=np.array([[3,1],[1,-2]])
b=np.array([9,3])

```

```

x=np.linalg.solve(A,b)
x

```

```

↳ array([ 3., -0.])

```

```

# Norms of a Vector
# Measures the "length" or "magnitude" of a vector.
# The most common is the Euclidean (or 2-norm), which is calculated as the square root of the sum of the squares of the vector's elements.

```

```

vector=np.array([1,2,3,4,5])
norms=np.linalg.norm(vector)
norms

```

```

↳ 7.416198487095663

```

```

# Singular Value Decomposition (SVD)
# A matrix factorization method that decomposes a matrix A into three matrices
#U, S, and V, where U and V are orthogonal matrices, and S contains the singular values.
#SVD is useful in data reduction and noise reduction.

```

```

U, S, V = np.linalg.svd(A)
print("U:\n", U)
print("S:\n", S)
print("V:\n", V)

```

```

↳ U:
  [[-0.98195639 -0.18910752]
   [-0.18910752  0.98195639]]
S:
  [3.1925824 2.1925824]
V:
  [[-0.98195639 -0.18910752]
   [ 0.18910752 -0.98195639]]

```

```

# QR Decomposition
# Decompose a matrix A into an orthogonal matrix Q and an upper traingle matrix R.
# QR decomposition is useful for solving linear systems and eigen pproblems

```

```

Q, R = np.linalg.qr(A)

```

```
print("Q:\n", Q)
print("R:\n", R)
```

```
Q:
[[-0.9486833 -0.31622777]
 [-0.31622777  0.9486833 ]]
R:
[[-3.16227766 -0.31622777]
 [ 0.          -2.21359436]]
```

## 12. Random Sampling

```
# Random Sampling from a Normal Distribution
```

```
# Normal Distribution (mean=0, standard deviation=1)
normal_sample = np.random.randn(5) # Generates 5 samples from a standard normal distribution
print(normal_sample)
```

```
print('\n')
```

```
#Normal Distribution with specific mean and standard deviation
normal_custom = np.random.normal(loc=5, scale=2, size=5) # Mean=5, StdDev=2
print(normal_custom)
```

```
[-1.41825583  0.59394642 -2.21025368  0.41355399 -0.96940032]

[ 6.17546499  6.30183893 -0.12000794  0.98798837  7.19710738]
```

```
# Random Sampling from a Uniform Distribution
```

```
#Uniform Distribution between two values
uniform_sample = np.random.uniform(low=0, high=1, size=5) # Generates 5 samples from a uniform distribution between 0 and 1
print(uniform_sample)
```

```
[0.02374098 0.28542258 0.74485629 0.41604621 0.67612603]
```

```
# Shuffling an Array
```

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

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

## 13. Avoiding Copy

### Use Views Instead of Copies

```
# Slicing in NumPy generally creates a view rather than a copy,
# meaning changes in the sliced array reflect in the original array.
```

```
arr = np.array([1, 2, 3, 4, 5])
view = arr[1:4] # This is a view, not a copy
view[0] = 10   # This changes arr as well
```

```
print(arr)
```

```
[ 1 10  3  4  5]
```

```
# Reshaping an array can also produce a view when possible
```

```
arr = np.array([1, 2, 3, 4, 5])
view = arr.reshape(5,1)
print(view)
print('\n')
view[0][0]=10
print(arr)
```



```

→ [[1]
   [2]
   [3]
   [4]
   [5]]

```

```
[10  2  3  4  5]
```

### Use .flat or .ravel() Instead of .flatten()

# .flatten() always returns a copy, while .ravel() will return a view if possible.

```

arr = np.array([[1, 2, 3], [4, 5, 6]])
flattened = arr.flatten()
flattened[0] = 10
print(flattened)

```

```

print('\n')
ravelled = arr.ravel()
ravelled[0] = 10
print(arr)

```

```

→ [10  2  3  4  5  6]

```

```

[[10  2  3]
 [ 4  5  6]]

```

## ✓ 14. Handling NAN

In NumPy, NaN ("Not a Number") values can appear when there are undefined or missing values in an array. Here are some common ways to handle them:

# Check for NaNs in an array using np.isnan():

```

arr = np.array([1, 2, np.nan, 4, 5])
has_nan = np.isnan(arr)
print(has_nan)

```

```

→ [False False  True False False]

```

# Remove NaNs from an array using masking or np.isnan()

```

arr = np.array([1, 2, np.nan, 4, 5])
non_nan = arr[~np.isnan(arr)]
print(non_nan)

```

```

→ [1.  2.  4.  5.]

```

# Replace NaNs with a specific value using np.nan\_to\_num() or np.where()

```
arr = np.array([1, 2, np.nan, 4, 5])
```

```

# Replace NaNs with 0
arr_filled = np.nan_to_num(arr, nan=0)
print(arr_filled)

```

```

→ [1.  2.  0.  4.  5.]

```

```
arr = np.array([1, 2, np.nan, 4, 5])
```

```

# Sum ignoring NaNs
sum_ignoring_nan = np.nansum(arr)
print(sum_ignoring_nan)

```

```
print('\n')
```

```
# Mean ignoring NaNs
```

```

mean_ignoring_nan = np.nanmean(arr)
print(mean_ignoring_nan)

print('\n')

# Max/Min ignoring NaNs
max_ignoring_nan = np.nanmax(arr)
min_ignoring_nan = np.nanmin(arr)
print(min_ignoring_nan)
print(max_ignoring_nan)

print('\n')

# Standard Deviation ignoring NaNs
std_ignoring_nan = np.nanstd(arr)
print(std_ignoring_nan)

```

```

↔ 12.0

3.0

1.0
5.0

1.5811388300841898

```

## ✓ 15. Save and Load

### Saving and Loading with .npy Format

```

# Save an array to a .npy file
arr = np.array([1, 2, 3, 4, 5])
np.save('array.npy', arr)

# Load an array from a .npy file:
loaded_arr = np.load('array.npy')

```

### Saving and Loading Multiple Arrays with .npz Format

```

# Save multiple arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
np.savez('arrays.npz', arr1=arr1, arr2=arr2)

# Load multiple arrays
loaded_arrays = np.load('arrays.npz')
loaded_arr1 = loaded_arrays['arr1']
loaded_arr2 = loaded_arrays['arr2']

```

### Saving and Loading as Text Files (.txt or .csv)

```

# Save an array to a text file:
np.savetxt('array.txt', arr, delimiter=',')

# Load an array from a text file:
loaded_arr = np.loadtxt('array.txt', delimiter=',')

```

### Saving and Loading Arrays in .csv Format

```

# Save to CSV
np.savetxt('array.csv', arr, delimiter=',', fmt='%d')

# Load from CSV
loaded_arr = np.loadtxt('array.csv', delimiter=',')

```

## ✓ 16. Memory Usage

```
# size_in_bytes=arr.nbytes
```

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

```
↔ 96
```

```
arr = np.array([1.1, 2.1, 3.1])  
print(arr.nbytes)
```

```
newarr = arr.astype(int)  
print(newarr)  
print(newarr.nbytes)
```

```
↔ 24  
[1 2 3]  
24
```

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.