In [1]:

```
#Numpy is an array like data structure used to calculate complex scientific op
erations

import numpy as np
```

In [2]:

```
#Initialize numpy with list

a = np.array([1, 7, 5, 2, 8])
print(a)
```

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

In [3]:

```
#Initialize numpy with arange function

a = np.arange(10)
print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

In [4]:

```
#Numpy VS list Comperison

L = range(1000)
%timeit([i**2 for i in L])
```

```
300 µs ± 25.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [5]:

```
a = np.arange(1000)
%timeit a**2
```

```
1.01 µs ± 7.65 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [6]:

```
#Here, numpy execute takes less then 1 micro-seconds, while list operation tak
es 271 micro-seconds
```

In [7]:

```
#Numpy array creation

#1-D array

a = np.array([5, 1, 7, 6, 4])
print(a)
```

```
[5 1 7 6 4]
```

In [8]:

```
#Array Dimension

a.ndim
```

Out[8]:

```
1
```

In [9]:

```
#Array Shape: Always return the dimensions

a.shape
```

Out[9]:

```
(5,)
```

In [10]:

```
#2-D Array

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

print('Array='+ str(b))
print('Dimension= ' + str(b.ndim))
print('Shape= ' + str(b.shape))
```

```
Array=[[1 3 5]
 [2 4 6]]
Dimension= 2
Shape= (2, 3)
```

In [11]:

```
#3-D Array

c = np.array([[[4, 7, 6, 1], [9, 2, 5, 7], [1, 0, 7, 9]], [[1, 3, 5, 7], [4, 6
, 8, 0], [4, 5, 8, 9]]])

print('Array='+ str(c))
print('Dimension= ' + str(c.ndim))
print('Shape= ' + str(c.shape))
```

```
Array=[[[4 7 6 1]
  [9 2 5 7]
  [1 0 7 9]]

 [[1 3 5 7]
  [4 6 8 0]
  [4 5 8 9]]]
Dimension= 3
Shape= (2, 3, 4)
```

In [12]:

```
#Creating numpy arrays: using arange function

#with 1 parement n -> array = [0, n-1]
arr = np.arange(10)
print(arr)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

In [13]:

```
#with 2 parement (start, end) -> array = [start to end-1 with uniform stem siz
e=1]
arr = np.arange(3, 10)
print(arr)
```

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

In [14]:

```
#with 3 parement (start, end, step) -> array = [start to end-1 with step size]
arr = np.arange(5, 10, 2)
print(arr)
```

```
[5 7 9]
```

In [15]:

```
#Creating numpy arrays: using linespace function paremeter(start, end, break-point)

arr = np.linspace(0, 1, 5)
print(arr)
```

```
[0.   0.25 0.5  0.75 1.  ]
```

In [16]:

```
#Creating ones array

arr = np.ones((3, 3)) #dimension

arr
```

Out[16]:

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

In [17]:

```
#Creating ones array

arr = np.zeros((3, 3)) #dimension

arr
```

Out[17]:

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

In [18]:

```
#Creating identity matrix (n, m) with n rows and m columns

arr = np.eye(3, 3)

arr
```

Out[18]:

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

In [19]:

```
#Creating diagonal matrix ([n1, n2, n3, ... nt]) with t diagonal values

arr = np.diag([7, 5, 6, 1])

arr
```

Out[19]:

```
array([[7, 0, 0, 0],
       [0, 5, 0, 0],
       [0, 0, 6, 0],
       [0, 0, 0, 1]])
```

In [20]:

```
#Creating random array

#for unifor random variable we use: rand(n)

arr = np.random.rand(5)
print(arr)

#for standard normal variant we use: randn(n)
brr = np.random.randn(5)
print(brr)
```

```
[0.88846198 0.17143381 0.45960649 0.28555434 0.64015709]
[-2.12660396  0.28841956  0.51255777  1.24473773  0.40968247]
```

In [21]:

```
#Basic Data types of numpy

arr = np.arange(10)

print(arr.dtype) #get the data type using dtype function
```

```
int32
```

In [22]:

```
arr = np.arange(10, dtype='float64')
print(arr)
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

In [23]:

```
#Array Accessing from numpy

#1-D array

arr = np.arange(10)

print(arr)
print(arr[5])
```

```
[0 1 2 3 4 5 6 7 8 9]
5
```

In [24]:

```
#2-D array

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

print(arr)
print(arr[1][2])
```

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

In [25]:

```
#Slicing numpy array

arr = np.arange(10)

print('Original Array= ', end='')
print(arr)

brr = arr[2:10:3] #here, arr[s:e:ss] means: s=start, e=end, and ss=step size
print('Modified Array= ', end='')
print(brr)
```

```
Original Array= [0 1 2 3 4 5 6 7 8 9]
Modified Array= [2 5 8]
```

In [26]:

```
#Assignment and slicing

arr = np.arange(10)

print('Old Array= ', end='')
print(arr)

#here, we assign -7 from 5 to rest of the index
arr[5:] = -7

print('New Array= ', end='')
print(arr)
```

```
Old Array= [0 1 2 3 4 5 6 7 8 9]
New Array= [ 0  1  2  3  4 -7 -7 -7 -7 -7]
```

In [27]:

```
#Assigment in reverse order

arr = np.arange(5)
print('Old Array= ', end='')
print(arr)

brr = arr[::-1]
print('New Array= ', end='')
print(brr)
```

```
Old Array= [0 1 2 3 4]
New Array= [4 3 2 1 0]
```

In [28]:

```
#Slicing share the memory. So if we update the slice part it also modify the m
ain array

a = np.arange(10)
b = a[::2]

print('Array a= ' + str(a))
print('Array b= ' + str(b))

#here, we assign 8 in the 0th index of b. and see the result

b[0] = 8

print()
print('Array a= ' + str(a))
print('Array b= ' + str(b))

"""This happend because of the memory share between slicing part and main par
t. Th check memory share we use the following technique"""

print('\na and b shared memory: {}'.format(np.shares_memory(a,b)))
```

```
Array a= [0 1 2 3 4 5 6 7 8 9]
Array b= [0 2 4 6 8]

Array a= [8 1 2 3 4 5 6 7 8 9]
Array b= [8 2 4 6 8]

a and b shared memory: True
```

In [29]:

```
#We can use copy to eleminate memory sharing

a = np.arange(10)
b = a[::2].copy()

print('Array a= ' + str(a))
print('Array b= ' + str(b))

#here, we assign 8 in the 0th index of b. and see the result

b[0] = 8

print()
print('Array a= ' + str(a))
print('Array b= ' + str(b))

"""Here, no memory will share between slicing part and main part"""

print('\na and b shared memory: {}'.format(np.shares_memory(a,b)))
```

```
Array a= [0 1 2 3 4 5 6 7 8 9]
Array b= [0 2 4 6 8]

Array a= [0 1 2 3 4 5 6 7 8 9]
Array b= [8 2 4 6 8]

a and b shared memory: False
```

In [30]:

```
#Fancy Indexing: copy from the main part. So, memory share not possible

a = np.random.randint(0, 20, 15) #randint(s, e, n) return a sequence of random
numbers which is 0<randNumber<20 and the length of sequence is n.
print('Old array= ' + str(a))

mask = (a%2 == 0)
b = a[mask]

print('New array= ' + str(b))
```

```
Old array= [17 19  4  9  5 16  2  8 11 14 11  6 16 18  2]
New array= [ 4 16  2  8 14  6 16 18  2]
```

In [31]:

```
#Numpy Operations: Elementwise operation

#Basic Operation

arr = np.array([2, 6, 1, 3, 8])

print('Old arr= ' + str(arr))

brr = arr + 1 #here, we add 1 to each elements of arr

print('New arr= ' + str(brr))
```

```
Old arr= [2 6 1 3 8]
New arr= [3 7 2 4 9]
```

In [32]:

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

b = np.ones(4) + 1

print(a-b) #here, a-b operation done in every elements of a and b
```

```
[-1.  0.  1.  2.]
```

In [33]:

```
c = np.ones(5)

print(a-c) #elementwise operation must place with same shape
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-0efba91f9fa4> in <module>()
      1 c = np.ones(5)
      2 
----> 3 print(a-c) #elementwise operation must place with same shape

ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

In [34]:

```
#Matrix Multiplication

a = np.diag([1, 3, 5])
b = np.ones((3,3)) + 2

print(a)
print()
print(b)

print('\nThe matrix multiplication of a and b is: ')
print(a * b) #same operation can be done by a.dot(b)
```

```
[[1 0 0]
 [0 3 0]
 [0 0 5]]

[[3. 3. 3.]
 [3. 3. 3.]
 [3. 3. 3.]]

The matrix multiplication of a and b is:
[[ 3.  0.  0.]
 [ 0.  9.  0.]
 [ 0.  0. 15.]]
```

In [35]:

```
a = np.array([1, 2, 3, 4])
b = np.array([7, 2, 2, 4])

print(a==b)
```

```
[False  True False  True]
```

In [36]:

```
#Array_equal check two array is elementwise same or not
print(np.array_equal(a,b))
```

```
False
```

In [37]:

```
#Logical Operators

a = np.array([0, 0, 1, 1], dtype=bool)
b = np.array([0, 1, 0, 1], dtype=bool)

print(np.logical_or(a, b))
print()
print(np.logical_and(a, b))
```

```
[False  True  True  True]

[False False False  True]
```

In [38]:

```
#Mathematical functions

a = (np.array([0., 30., 45., 60., 90.]) * np.pi / 180.) #angle should be in ra
dian

print(np.round(np.sin(a),3)) #sin function
print(np.round(np.cos(a),3)) #cos function
```

```
[0.    0.5   0.707 0.866 1.   ]
[1.    0.866 0.707 0.5   0.   ]
```

In [39]:

```
a = np.arange(5)

print(np.exp(a)) #here, exp in e^x in math
```

```
[ 1.          2.71828183  7.3890561  20.08553692 54.59815003]
```

In [40]:

```
#Basic reduction functions

x = np.array([1, 2, 3, 4])
print(np.sum(x))
```

```
10
```

In [41]:

```
#exmples of sum functions

x = np.array([[1, 2], [3, 4]])

print('Sum of all elements: ', x.sum())
print('Sum of all columns : ', x.sum(axis=0)) #summation of all columns, when
 axis = 0
print('Sum of all rows    : ', x.sum(axis=1)) #summation of all rows, when axi
s = 1
```

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

In [42]:

```
#exmples of min functions

x = np.array([[1, 2], [3, 4]])

print('Minimum of all elements: ', x.min())
print('Minimum of all columns : ', x.min(axis=0)) #minimum of all columns, when axis = 0
print('Minimum of all rows    : ', x.min(axis=1)) #minimum of all rows, when axis = 1
```

```
Minimum of all elements:  1
Minimum of all columns :  [1 2]
Minimum of all rows    :  [1 3]
```

In [43]:

```
#max function also work same as min. Now we have another two functions. one is argmin(), and another is argmax()

x = np.array([7, -2, 2, 1, 9, 0])

print('Index {} hold minimum value'. format(np.argmin(x)))
print('Index {} hold maximum value'. format(np.argmax(x)))
```

```
Index 1 hold minimum value
Index 4 hold maximum value
```

In [44]:

```
#Statistical Functions

#mean function

x = np.array([[1, 3, 5], [2, 4, 6]])
print('Mean of all elements: ', x.mean())
print('Mean of all columns : ', x.mean(axis=0)) #minimum of all columns, when axis = 0
print('Mean of all rows    : ', x.mean(axis=1)) #minimum of all rows, when axis = 1
```

```
Mean of all elements:  3.5
Mean of all columns :  [1.5 3.5 5.5]
Mean of all rows    :  [3. 4.]
```

In [45]:

```
'''The function median() and std() work for median and standard deviation'''
x = np.array([[1, 3, 5], [2, 4, 6]])

print('Median of all elements: ', np.median(x))
print('Standard deviation of all elements: ', x.std())
```

```
Median of all elements:  3.5
Standard deviation of all elements:  1.707825127659933
```

In [46]:
```
#Load data from text file in numpy array
import os

data = np.loadtxt('D:\Development\Applied AI and Machine Learning\Codes\popula
tions.txt')

print(data)
```

```
[[ 1900. 30000.  4000. 48300.]
 [ 1901. 47200.  6100. 48200.]
 [ 1902. 70200.  9800. 41500.]
 [ 1903. 77400. 35200. 38200.]
 [ 1904. 36300. 59400. 40600.]
 [ 1905. 20600. 41700. 39800.]
 [ 1906. 18100. 19000. 38600.]
 [ 1907. 21400. 13000. 42300.]
 [ 1908. 22000.  8300. 44500.]
 [ 1909. 25400.  9100. 42100.]
 [ 1910. 27100.  7400. 46000.]
 [ 1911. 40300.  8000. 46800.]
 [ 1912. 57000. 12300. 43800.]
 [ 1913. 76600. 19500. 40900.]
 [ 1914. 52300. 45700. 39400.]
 [ 1915. 19500. 51100. 39000.]
 [ 1916. 11200. 29700. 36700.]
 [ 1917.  7600. 15800. 41800.]
 [ 1918. 14600.  9700. 43300.]
 [ 1919. 16200. 10100. 41300.]
 [ 1920. 24700.  8600. 47300.]]
```

In [47]:
```
#Initialize all the variables form the data

year, hare, lynx, carrot = data.T

print(year)
```

```
[1900. 1901. 1902. 1903. 1904. 1905. 1906. 1907. 1908. 1909. 1910. 1911.
 1912. 1913. 1914. 1915. 1916. 1917. 1918. 1919. 1920.]
```

In [48]:
```
#Get only the population data

populations = data[:, 1:]
print(populations)
```

```
[[30000.  4000. 48300.]
 [47200.  6100. 48200.]
 [70200.  9800. 41500.]
 [77400. 35200. 38200.]
 [36300. 59400. 40600.]
 [20600. 41700. 39800.]
 [18100. 19000. 38600.]
 [21400. 13000. 42300.]
 [22000.  8300. 44500.]
 [25400.  9100. 42100.]
 [27100.  7400. 46000.]
 [40300.  8000. 46800.]
 [57000. 12300. 43800.]
 [76600. 19500. 40900.]
 [52300. 45700. 39400.]
 [19500. 51100. 39000.]
 [11200. 29700. 36700.]
 [ 7600. 15800. 41800.]
 [14600.  9700. 43300.]
 [16200. 10100. 41300.]
 [24700.  8600. 47300.]]
```

In [49]:
```
#Standard deviations of all the specises

print(populations.std(axis=0))
```

```
[20897.90645809 16254.59153691  3322.50622558]
```

In [50]:
```
#Whic specises has maximum population in the given years

max_population = np.argmax(populations, axis=1)

print(max_population)
```

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

In [51]:
```
#Broadcasting

a = np.arange(0, 40, 10)
print('Original Array= ' + str(a))
```

```
Original Array= [ 0 10 20 30]
```

In [52]:

```
#replicate same value of the row in 3 times

b = np.tile(a, (3,1))
print('New Array= \n', b)
```

```
New Array=
 [[ 0 10 20 30]
 [ 0 10 20 30]
 [ 0 10 20 30]]
```

In [53]:

```
#transpose the matrix

c = b.T
print('Transpose matrix= \n', c)
```

```
Transpose matrix=
 [[ 0  0  0]
 [10 10 10]
 [20 20 20]
 [30 30 30]]
```

In [54]:

```
d = np.array([0, 1, 2])

e = c+d #here, c and d doesn't have same number of row. But the column number
 is same.

print(e)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

In [55]:

```
a = np.arange(0, 40, 10)

print('a = ', a)
```

```
a =  [ 0 10 20 30]
```

In [56]:

```
print('Shape of a= ', a.shape)

a = a[:, np.newaxis]
print('Shape of a= ', a.shape)
```

```
Shape of a=  (4,)
Shape of a=  (4, 1)
```

In [57]:

```
print('a = ', a)
```

```
a =  [[ 0]
 [10]
 [20]
 [30]]
```

In [58]:

```
b = np.array([0, 1, 2])

print(a+b)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

In [59]:

```
'''if we want to add two differnt shape array. At first we have to transform t
he first matrix 2D from 1D'''
```

Out[59]:

```
'if we want to add two differnt shape array. At first we have to transform th
e first matrix 2D from 1D'
```

In [60]:

```
#Array Shape manipulation
```

In [61]:

```
#Flattening: Change a 2D matrix to a single stright line

a = np.array([[1, 3, 5], [2, 4, 6]])
print('Old array=\n', a)

b = a.ravel()
print('\nNew array= ', b)
```

```
Old array=
 [[1 3 5]
 [2 4 6]]

New array=  [1 3 5 2 4 6]
```

In [62]:

```
b = (a.T).ravel()
print('New array= ', b)
```

```
New array=  [1 2 3 4 5 6]
```

In [63]:

```
#Reshape: shape 1D array to a specific form of matrix

a = np.array([[1, 3, 5], [2, 4, 6]])
print('Old array[2x3]=\n', a)

b = a.ravel()
print('\nFlat 1D array= ', b)

a = b.reshape((3, 2))
print('\nNew array[3x2]=\n', a)
```

```
Old array[2x3]=
 [[1 3 5]
 [2 4 6]]

Flat 1D array=  [1 3 5 2 4 6]

New array[3x2]=
 [[1 3]
 [5 2]
 [4 6]]
```

In [64]:

```
#Resize: resize function modify the size of an array

a = np.arange(4)

print(a)

a.resize((8,))
print(a)

b = a
a.resize((4,))
```

```
[0 1 2 3]
[0 1 2 3 0 0 0 0]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-64-c9badd8f9363> in <module>()
      9
     10 b = a
---> 11 a.resize((4,))

ValueError: cannot resize an array that references or is referenced
by another array in this way.  Use the resize function
```

In [65]:

```
#If we assign one array to another. Then it is not possible to resize the orig
inal array
```

In [66]:

```
#Sorting: Eelement of array

a = np.array([3, 5, 1, 2, 7])
print('Old= ', a)

b = np.sort(a) #sort and assign in new variable
print('New= ', b)

print('\nOld= ', a)  #sort and assign in same variable
a.sort()
print('New= ', a)
```

```
Old=  [3 5 1 2 7]
New=  [1 2 3 5 7]

Old=  [3 5 1 2 7]
New=  [1 2 3 5 7]
```

In [67]:

```
#Sorting: fancy indexing

a = np.array([3, 1, 2, 6, 5])

sorted_index = np.argsort(a)
sorted_data = np.sort(a)

print('Unsorted data= ', a)
print('Sorted data=   ', sorted_data)
print('Sorted index=  ', sorted_index)
```

```
Unsorted data=  [3 1 2 6 5]
Sorted data=    [1 2 3 5 6]
Sorted index=   [1 2 0 4 3]
```

In [68]:

```
print(a[sorted_index])
```

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