Numpy Array

In [1]:

```
import numpy as np
type(np.array(["vbhvh", 2, 3]))
```

Out[1]:

```
numpy.ndarray
```

In [4]:

```
l=[1,2,3,4,5,6,7]
type(l)
```

Out[4]:

```
list
```

In [5]:

```
type(np.array(l))
```

Out[5]:

```
numpy.ndarray
```

In [10]:

```
#Upcasting:
# if all are integer then all are taken as interger
# if anyone is string then all are taken as string
```

In [11]:

```
a=np.array([1, 2, 3.0])
```

In [17]:

```
a.shape  # shape of array
```

Out[17]:

```
(3,)
```

In [18]:

```
a.ndim   # number of dimension
```

Out[18]:

```
1
```

In [4]:

```
#two dimensions
```

In [21]:

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

In [23]:

```
d.ndim   # dimension is 2
```

Out[23]:

```
2
```

In [19]:

```
#Minimum dimensions 2:
```

In [26]:

```
np.array([1, 2, 3], ndmin=5)   # it is able to convert any number pf dimensions by usir
```

Out[26]:

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

In [27]:

```
#dtype
```

In [29]:

```
np.array([1, 2, 3], dtype=complex)   # change it in any format
```

Out[29]:

```
array([1.+0.j, 2.+0.j, 3.+0.j])
```

In [36]:

```
np.array([1, 2, 3], dtype=float)
```

Out[36]:

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

In [37]:

```
#Data-type consisting of more than one element:
```

In [49]:

```
x = np.array([(1,2),(3,4)],dtype=[('a','<i2'),('b','<i8')])
x
# for 1 datatype is 'a'  and for 2 datatype is '<i2'
# and so on for all the tuples
# '<i2'  is int16   and '<i8' is int 64
```

Out[49]:

```
array([(1, 2), (3, 4)], dtype=[('a', '<i2'), ('b', '<i8')])
```

In [50]:

```
x[0]
```

Out[50]:

```
(1, 2)
```

In [51]:

```
type(x[1][0])
```

Out[51]:

```
numpy.int16
```

In [52]:

```

#Creating an array from sub-classes:
```

In [53]:

```
np.mat(np.array([[1, 2],[4,7]]))
```

Out[53]:

```
matrix([[1, 2],
        [4, 7]])
```

In [55]:

```
m=np.mat('1 2; 3 4')
```

In [56]:

```
m[0]
```

Out[56]:

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

In [58]:

```
m[0, 1]
```

Out[58]:

```
2
```

numpy.asarray

In [59]:

```
#Convert the input to an array.
```

In [60]:

```
#Convert a list into an array:
```

In [61]:

```
 a = [1, 2]
type(a)
```

Out[61]:

```
list
```

In [63]:

```
type(np.asarray(a))
```

Out[63]:

```
numpy.ndarray
```

In [80]:

```
type(np.asanyarray([1,2,3,4,5]))   # convert data into array
```

Out[80]:

```
numpy.ndarray
```

In [81]:

```
a = np.array([1, 2]) #Existing arrays are not copied
type(a)
```

Out[81]:

```
numpy.ndarray
```

In [82]:

```
np.asarray((1,2))
```

Out[82]:

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

In [83]:

```
#If dtype is set, array is copied only if dtype does not match:
```

In [84]:

```
a = np.array([1, 2], dtype=np.float32)
a
```

Out[84]:

```
array([1., 2.], dtype=float32)
```

In [85]:

```
np.asarray([1,2]) is a
```

Out[85]:

```
False
```

In [86]:

```
1 np.asarray([1,2])
```

Out[86]:

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

In [87]:

```
1 np.asarray(a, dtype=np.float64) is a
```

Out[87]:

```
False
```

In [88]:

```
1 # ndarray subclasses are not passed through
```

In [89]:

```
1 issubclass(np.matrix, np.ndarray)
```

Out[89]:

```
True
```

In [90]:

```
1 a = np.matrix([[1, 2]])
2 np.asanyarray(a)
3
```

Out[90]:

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

In [91]:

```
1 np.asarray(a) is a
```

Out[91]:

```
False
```

In [92]:

```
1 np.asanyarray(a) is a
```

Out[92]:

```
True
```

In [93]:

```
1 type(np.asarray(a))
```

Out[93]:

```
numpy.ndarray
```

numpy.asanyarray

In [94]:

```
a = [1, 2]
np.asanyarray(a)
```

Out[94]:

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

In [95]:

```
a = np.matrix([1, 2])
a
```

Out[95]:

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

In [96]:

```
np.asanyarray([1,2]) is a
```

Out[96]:

```
False
```

```
#numpy.copy
```

In [97]:

```
np.array(a, copy=True)  # copy on another location
```

Out[97]:

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

In [40]:

```
#Create an array x, with a reference y and a copy z:
```

In [99]:

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

```

Out[99]:

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

In [100]:

```
y = x
```

In [101]:

```
z = np.copy(x)
```

In [102]:

```
y[0] = 100
x
```

Out[102]:

```
array([100,   2,   3])
```

In [112]:

```
z  # z not change
```

Out[112]:

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

In [106]:

```
id(x[0])
```

Out[106]:

```
2243456502864
```

In [107]:

```
id(y[0])
```

Out[107]:

```
2243456502608
```

In [108]:

```
id(x)
```

Out[108]:

```
2243456767360
```

In [109]:

```
id(y)
```

Out[109]:

```
2243456767360
```

numpy.fromfunction

In [113]:

```
#Construct an array by executing a function over each coordinate.
```

In [124]:

```
np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)   # write own external functior
# 3 dimension means (3,3,3) that array of 3 matrix of order 3X3 and same as on 4 dimens
```

Out[124]:

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

In [116]:

```
np.fromfunction(lambda i, j: i * j, (3, 3), dtype=int)
```

Out[116]:

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

In [61]:

```
#Create a new 1-dimensional array from an iterable object.
```

In [119]:

```
iterable = (x*x for x in range(5))
iterable
```

Out[119]:

```
<generator object <genexpr> at 0x0000020A586FFCF0>
```

In [125]:

```
np.fromiter(iterable, float)
```

Out[125]:

```
array([], dtype=float64)
```

In [126]:

```
#A new 1-D array initialized from text data in a string
```

In [127]:

```
a = np.fromstring('234 234',sep=' ')
a
```

Out[127]:

```
array([234., 234.])
```

In [128]:

```
np.fromstring('1, 2', dtype=int, sep=',')
```

Out[128]:

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

In [130]:

```
# How to create a record array from a (flat) list of arrays
```

In [131]:

```
x1=np.array([1,2,3,4])
```

In [132]:

```
x2=np.array(['a','dd','xyz','12'])
```

In [133]:

```
x3=np.array([1.1,2,3,4])
x4=np.array([1.1,2,3,4])
type(x4)
```

Out[133]:

```
numpy.ndarray
```

In [140]:

```
r = np.core.records.fromarrays([x1,x2,x3,x4],names='a,b,c,d')
r
#  shape of array are mismatch then give error
```

Out[140]:

```
rec.array([( 1, 'a', 1.1, 1.1), (34, 'dd', 2. , 2. ),
           ( 3, 'xyz', 3. , 3. ), ( 4, '12', 4. , 4. )],
          dtype=[('a', '<i4'), ('b', '<U3'), ('c', '<f8'), ('d', '<f8')])
```

In [141]:

```
print(r[1]["a"])
```

```
34
```

In [142]:

```
x1[1]=34
```

In [143]:

```
x1[1]
```

Out[143]:

```
34
```

data types

In [144]:

```
my_list = [1,2,3]
import numpy as np
arr = np.array(my_list)
print("Type/Class of this object:",type(arr))
print("Here is the vector\n--------------------\n",arr)
```

```
Type/Class of this object: <class 'numpy.ndarray'>
Here is the vector
--------------------
 [1 2 3]
```

In [147]:

```
my_mat = [[1,2,3],[4,5,6],[7,8,9]]
mat = np.array(my_mat)
print("Type/Class of this object:",type(mat))
print("Here is the matrix\n----------\n",mat,"\n----------")
print("Dimension of this matrix: ",mat.ndim,sep='') #ndim gives the dimensison, 2 for d
print("Size of this matrix: ", mat.size,sep='') #size gives the total number of element
print("Shape of this matrix: ", mat.shape,sep='') #shape gives the number of elements d
print("Data type of this matrix: ", mat.dtype,sep='') #dtype gives the data type conta
```

```
Type/Class of this object: <class 'numpy.ndarray'>
Here is the matrix
----------
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
----------
Dimension of this matrix: 2
Size of this matrix: 9
Shape of this matrix: (3, 3)
Data type of this matrix: int32
```

In [148]:

```
my_mat = [[1.1,2,3],[4,5.2,6],[7,8.3,9]]
mat = np.array(my_mat)
print("Data type of the modified matrix: ", mat.dtype,sep='') #dtype gives the data typ
print("\n\nEven tuples can be converted to ndarrays...")
```

```
Data type of the modified matrix: float64


Even tuples can be converted to ndarrays...
```

In [150]:

```
b = np.array([(1.5,2,3), (4,5,6)])
print("We write b = np.array([(1.5,2,3), (4,5,6)])")
print("Matrix made from tuples, not lists\n--------------------------------------")
print(b)
```

```
We write b = np.array([(1.5,2,3), (4,5,6)])
Matrix made from tuples, not lists
--------------------------------------
[[1.5 2.  3. ]
 [4.  5.  6. ]]
```

arange and linspace

In [151]:

```
print("A series of numbers:",type(np.arange(5,16)))
np.arange(5,16,2.3)# A series of numbers from low to high
```

```
A series of numbers: <class 'numpy.ndarray'>
```

Out[151]:

```
array([ 5. ,  7.3,  9.6, 11.9, 14.2])
```

In [152]:

```
list(range(5,16,2))
```

Out[152]:

```
[5, 7, 9, 11, 13, 15]
```

In [153]:

```
list(range(50,-1,5))
```

Out[153]:

```
[]
```

In [155]:

```
list(range(50,-1,-5))
```

Out[155]:

```
[50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 0]
```

In [156]:

```
print("Numbers spaced apart by 2:",np.arange(50,-1,5)) # Numbers spaced apart by 2
```

```
Numbers spaced apart by 2: []
```

In [157]:

```
print("Numbers spaced apart by float:",np.arange(0,11,2.5)) # Numbers spaced apart by 2
```

```
Numbers spaced apart by float: [ 0.   2.5  5.   7.5 10. ]
```

In [158]:

```
print("Every 5th number from 50 in reverse order\n",np.arange(5.0,-1,-5))
```

```
Every 5th number from 50 in reverse order
 [5. 0.]
```

In [159]:

```
print("21 linearly spaced numbers between 1 and 5\n----------------------------------")
print((np.linspace(1,5,50)))
```

```
21 linearly spaced numbers between 1 and 5
----------------------------------------
[1.         1.08163265 1.16326531 1.24489796 1.32653061 1.40816327
 1.48979592 1.57142857 1.65306122 1.73469388 1.81632653 1.89795918
 1.97959184 2.06122449 2.14285714 2.2244898  2.30612245 2.3877551
 2.46938776 2.55102041 2.63265306 2.71428571 2.79591837 2.87755102
 2.95918367 3.04081633 3.12244898 3.20408163 3.28571429 3.36734694
 3.44897959 3.53061224 3.6122449  3.69387755 3.7755102  3.85714286
 3.93877551 4.02040816 4.10204082 4.18367347 4.26530612 4.34693878
 4.42857143 4.51020408 4.59183673 4.67346939 4.75510204 4.83673469
 4.91836735 5.        ]
```

Matrix creation

In [160]:

```
print("Vector of zeroes\n---------------------")
print(np.zeros(5))
```

```
Vector of zeroes
---------------------
[0. 0. 0. 0. 0.]
```

In [161]:

```
print("Matrix of zeroes\n--------------------")
print(np.zeros((3,4))) # Notice Tuples
```

```
Matrix of zeroes
--------------------
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

In [162]:

```
print("Vector of ones\n---------------------")
print(np.ones(5))
```

```
Vector of ones
---------------------
[1. 1. 1. 1. 1.]
```

In [163]:

```
print("Matrix of ones\n---------------------")
print(np.ones((5,2,8))) # Note matrix dimension specified by Tuples

```

```
Matrix of ones
---------------------
[[[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. 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. 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. 1. 1.]]]
```

In [165]:

```
print("Matrix of 5's\n---------------------")
print(5+np.ones((3,5)))
```

```
Matrix of 5's
---------------------
[[6. 6. 6. 6. 6.]
 [6. 6. 6. 6. 6.]
 [6. 6. 6. 6. 6.]]
```

In [193]:

```
print("Empty matrix\n-------------\n", np.empty((3,5)))   # matrix of dummies value
```

```
Empty matrix
-------------
 [[6. 6. 6. 6. 6.]
 [6. 6. 6. 6. 6.]
 [6. 6. 6. 6. 6.]]
```

In [149]:

```
mat1 = np.eye(4)
print("Identity matrix of dimension", mat1.shape)
print(mat1)
```

```
Identity matrix of dimension (4, 4)
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

In [176]:

```
np.eye(5)
```

Out[176]:

```
array([[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.]])
```

In [177]:

```
np.arange(3)
```

Out[177]:

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

In [178]:

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

Out[178]:

```
array([0., 1., 2.])
```

In [179]:

```
 np.arange(3,7)
```

Out[179]:

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

In [180]:

```
np.arange(3,7,2)
```

Out[180]:

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

In [185]:

```
np.linspace(2.0, 3.0, num=5)
```

Out[185]:

```
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
```

In [186]:

```
np.linspace(2.0, 3.0, num=5, endpoint=False)
```

Out[186]:

```
array([2. , 2.2, 2.4, 2.6, 2.8])
```

In [187]:

```
np.linspace(2.0, 3.0, num=9, retstep=True)
```

Out[187]:

```
(array([2.   , 2.125, 2.25 , 2.375, 2.5  , 2.625, 2.75 , 2.875, 3.   ]), 0.1
25)
```

In [188]:

```
#Return numbers spaced evenly on a log scale.
np.linspace(2.0, 3.0, num=4)
```

Out[188]:

```
array([2.        , 2.33333333, 2.66666667, 3.        ])
```

Logspace

In [195]:

```
np.logspace(2.0, 3.0, num=4,base = 10)  # it give logarithmic value of that number
```

Out[195]:

```
array([ 100.        ,  215.443469  ,  464.15888336, 1000.        ])
```

In [197]:

```
np.logspace(2.0, 3.0, num=4, endpoint=False)
```

Out[197]:

```
array([100.        , 177.827941  , 316.22776602, 562.34132519])
```

In [198]:

```
np.logspace(2.0, 3.0, num=4, base=2.0)
```

Out[198]:

```
array([4.        , 5.0396842 , 6.34960421, 8.        ])
```

In [115]:

```
#Extract a diagonal or construct a diagonal array.
```

In [205]:

```
x = np.arange(16).reshape((4,4))
```

In [206]:

```
x
```

Out[206]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [210]:

```
np.arange(16).reshape((4,-2))  # -2 is nothing numpy automaticaly understand
# what is the order with respect top first element  of order
```

Out[210]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [211]:

```
np.diag(x) # give all diagonal element
```

Out[211]:

```
array([ 0,  5, 10, 15])
```

In [218]:

```
np.diag(x, k=2)
# k>0 means diagonal above digonal
# k<0 it means diagonal of below the longest diagonal
```

Out[218]:

```
array([2, 7])
```

In [219]:

```
np.diag(x, k=-1)
```

Out[219]:

```
array([ 4,  9, 14])
```

In [220]:

```
np.diag(np.diag(x))
```

Out[220]:

```
array([[ 0,  0,  0,  0],
       [ 0,  5,  0,  0],
       [ 0,  0, 10,  0],
       [ 0,  0,  0, 15]])
```

In [122]:

```
#Create a two-dimensional array with the flattened input as a diagonal.
```

In [221]:

```
np.diagflat([[1,2], [3,4]])
```

Out[221]:

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

In [222]:

```
np.diagflat([1,2], 1)
```

Out[222]:

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

In [223]:

```
#An array with ones at and below the given diagonal and zeros elsewhere.
```

In [224]:

```
np.tri(3, 5, 1, dtype=int)
```

Out[224]:

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

In [225]:

```
np.tri(3, 5,k=-1)
```

Out[225]:

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

In [226]:

```
#return a Lower triangle of an array.
```

In [227]:

```
np.tril([[1,2,3],[4,5,6],[7,8,9]], 0)
```

Out[227]:

```
array([[1, 0, 0],
       [4, 5, 0],
       [7, 8, 9]])
```

In [228]:

```
#return Upper triangle of an array.
```

In [229]:

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

Out[229]:

```
array([[0, 2, 3],
       [0, 0, 6],
       [0, 0, 0],
       [0, 0, 0]])
```

Random number generation

In [230]:

```
print("Random number generation (from Uniform distribution)")
print(np.random.rand(2,3)) # 2 by 3 matrix with random numbers ranging from 0 to 1, Not
```

```
Random number generation (from Uniform distribution)
[[0.89421746 0.33738014 0.11568204]
 [0.25877632 0.96071706 0.58659781]]
```

In [232]:

```
print("Numbers from Normal distribution with zero mean and standard deviation 1 i.e. st
print(np.random.randn(4,3))
```

```
Numbers from Normal distribution with zero mean and standard deviation 1 i.
e. standard normal
[[ 0.10140966 -1.03476451 -0.33294421]
 [ 0.36281919  0.118804    0.79776546]
 [ 0.81253271 -0.36563605 -0.07826843]
 [ 0.00535548 -0.19104988  1.07780411]]
```

In [233]:

```
print("Random integer vector:",np.random.randint(1,10)) #randint (low, high, # of sampl
print ("\nRandom integer matrix")

```

```
Random integer vector: 3

Random integer matrix
```

In [234]:

```
print(np.random.randint(1,100,(4,4))) #randint (low, high, # of samples to be drawn in
print("\n20 samples drawn from a dice throw:",np.random.randint(1,7,20)) # 20 samples d
```

```
[[67 14 41 13]
 [20 10 85 16]
 [49 54 50 96]
 [81 75 87 60]]

20 samples drawn from a dice throw: [3 5 4 3 6 5 4 1 6 2 4 2 1 6 5 4 6 5 3
6]
```

Reshaping

In [239]:

```
from numpy.random import randint as ri
a = ri(1,100,30)
b = a.reshape(2,3,5)
c = a.reshape(6,-19878)
c
```

Out[239]:

```
array([[92, 14, 94, 37, 57],
       [38, 20, 79, 61, 94],
       [20, 86,  1, 26, 74],
       [16, 23, 89, 25, 20],
       [51, 11, 47, 28, 61],
       [53, 92, 86, 26, 81]])
```

In [240]:

```
a
```

Out[240]:

```
array([92, 14, 94, 37, 57, 38, 20, 79, 61, 94, 20, 86,  1, 26, 74, 16, 23,
       89, 25, 20, 51, 11, 47, 28, 61, 53, 92, 86, 26, 81])
```

In [241]:

```
1 b
```

Out[241]:

```
array([[[92, 14, 94, 37, 57],
        [38, 20, 79, 61, 94],
        [20, 86,  1, 26, 74]],

       [[16, 23, 89, 25, 20],
        [51, 11, 47, 28, 61],
        [53, 92, 86, 26, 81]]])
```

In [242]:

```
1 print ("Shape of a:", a.shape)
2 print ("Shape of b:", b.shape)
3 print ("Shape of c:", c.shape)
4
```

```
Shape of a: (30,)
Shape of b: (2, 3, 5)
Shape of c: (6, 5)
```

In [243]:

```
1 print("\na looks like\n",'-'*20,"\n",a,"\n",'-'*20)
2 print("\nb looks like\n",'-'*20,"\n",b,"\n",'-'*20)
3 print("\nc looks like\n",'-'*20,"\n",c,"\n",'-'*20)
4
```

```
a looks like
 --------------------
 [92 14 94 37 57 38 20 79 61 94 20 86  1 26 74 16 23 89 25 20 51 11 47 28
 61 53 92 86 26 81]
 --------------------

b looks like
 --------------------
 [[[92 14 94 37 57]
  [38 20 79 61 94]
  [20 86  1 26 74]]

 [[16 23 89 25 20]
  [51 11 47 28 61]
  [53 92 86 26 81]]]
 --------------------

c looks like
```

In [244]:

```
A = ri(1,100,10) # Vector of random interegrs
print("\nVector of random integers\n",'-'*50,"\n",A)
print("\nHere is the sorted vector\n",'-'*50,"\n",np.sort(A))

```

```
Vector of random integers
 --------------------------------------------------
 [80 15 40 42 45 41 10 77 32 12]

Here is the sorted vector
 --------------------------------------------------
 [10 12 15 32 40 41 42 45 77 80]
```

In [252]:

```
M = ri(1,100,25).reshape(5,5) # Matrix of random interegrs
#print("\nHere is the sorted matrix along each row\n",'-'*50,"\n",np.sort(M, kind='merg
print("\nHere is the sorted matrix along each column\n",'-'*50,"\n",np.sort(M, axis=1,
M
#  here axis=1 means it is doing according to rows, it is doing actually columnwise but
# and work column wise by axis=0
```

```
Here is the sorted matrix along each column
 --------------------------------------------------
 [[29 36 53 85 87]
 [14 32 80 89 90]
 [ 5 10 56 63 99]
 [11 24 34 73 77]
 [ 6 14 14 71 76]]
```

Out[252]:

```
array([[85, 36, 29, 87, 53],
       [14, 32, 90, 89, 80],
       [63, 99, 56, 10,  5],
       [34, 77, 73, 11, 24],
       [71, 76,  6, 14, 14]])
```

In [253]:

```
print(np.sort(M, axis=0, kind='mergesort'))
M
```

```
[[14 32  6 10  5]
 [34 36 29 11 14]
 [63 76 56 14 24]
 [71 77 73 87 53]
 [85 99 90 89 80]]
```

Out[253]:

```
array([[85, 36, 29, 87, 53],
       [14, 32, 90, 89, 80],
       [63, 99, 56, 10,  5],
       [34, 77, 73, 11, 24],
       [71, 76,  6, 14, 14]])
```

In [257]:

```
print("Max of a:", M.max())
print("Max of b:", b.max())
print(a)
b

```

```
Max of a: 99
Max of b: 94
[92 14 94 37 57 38 20 79 61 94 20 86  1 26 74 16 23 89 25 20 51 11 47 28
 61 53 92 86 26 81]
```

Out[257]:

```
array([[[92, 14, 94, 37, 57],
        [38, 20, 79, 61, 94],
        [20, 86,  1, 26, 74]],

       [[16, 23, 89, 25, 20],
        [51, 11, 47, 28, 61],
        [53, 92, 86, 26, 81]]])
```

In [258]:

```
M
```

Out[258]:

```
array([[85, 36, 29, 87, 53],
       [14, 32, 90, 89, 80],
       [63, 99, 56, 10,  5],
       [34, 77, 73, 11, 24],
       [71, 76,  6, 14, 14]])
```

In [270]:

```
print("Max of a location:", M.argmax(axis= 1 ))    # give index of max element of all r
print("Max of b location:", b.argmax())


```

```
Max of a location: [3 2 1 1 1]
Max of b location: 2
```

Indexing and slicing

In [271]:

```
arr = np.arange(0,11)
print("Array:",arr)

```

```
Array: [ 0  1  2  3  4  5  6  7  8  9 10]
```

In [273]:

```
print("Element at 7th index is:", arr[7])

```

```
Element at 7th index is: 7
```

In [274]:

```
print("Elements from 3rd to 5th index are:", arr[3:6:2])

```

```
Elements from 3rd to 5th index are: [3 5]
```

In [275]:

```
print("Elements up to 4th index are:", arr[:4])
arr
```

```
Elements up to 4th index are: [0 1 2 3]
```

Out[275]:

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

In [276]:

```
print("Elements from last backwards are:", arr[-1:7:-1])

```

```
Elements from last backwards are: [10  9  8]
```

In [277]:

```
print("3 Elements from last backwards are:", arr[-1:-6:2])

```

```
3 Elements from last backwards are: []
```

In [278]:

```
arr = np.arange(0,21,2)
print("New array:",arr)

```

```
New array: [ 0  2  4  6  8 10 12 14 16 18 20]
```

In [279]:

```
print("Elements at 2nd, 4th, and 9th index are:", arr[[2,4,9]]) # Pass a list as a inde
```

```
Elements at 2nd, 4th, and 9th index are: [ 4  8 18]
```

In [280]:

```
import numpy as np
mat = np.array(ri(10,100,15)).reshape(3,5)
print("Matrix of random 2-digit numbers\n--------------------------------\n",mat)
mat[1:4,3:5]
# mat[0:3,[1,3]]
```

```
Matrix of random 2-digit numbers
--------------------------------
 [[97 32 31 33 69]
 [81 97 65 13 74]
 [17 70 21 45 37]]
```

Out[280]:

```
array([[13, 74],
       [45, 37]])
```

In [283]:

```
mat[0:3,[1,3]]
```

Out[283]:

```
array([[32, 33],
       [97, 13],
       [70, 45]])
```

In [284]:

```
mat[0:3,[1,3]]
mat
```

Out[284]:

```
array([[97, 32, 31, 33, 69],
       [81, 97, 65, 13, 74],
       [17, 70, 21, 45, 37]])
```

In [286]:

```
print("\nDouble bracket indexing\n-------------------------")
print("Element in row index 1 and column index 2:", mat[1][1])
mat
```

```
Double bracket indexing
-----------------------
Element in row index 1 and column index 2: 97
```

Out[286]:

```
array([[97, 32, 31, 33, 69],
       [81, 97, 65, 13, 74],
       [17, 70, 21, 45, 37]])
```

In [287]:

```
print("\nSingle bracket with comma indexing\n---------------------------------")
print("Element in row index 1 and column index 2:", mat[1,2])
print("\nRow or column extract\n----------------------")

```

```
Single bracket with comma indexing
---------------------------------
Element in row index 1 and column index 2: 65

Row or column extract
----------------------
```

In [288]:

```
print("Entire row at index 2:", mat[2])
print("Entire column at index 3:", mat[:,3])

```

```
Entire row at index 2: [17 70 21 45 37]
Entire column at index 3: [33 13 45]
```

In [289]:

```
print("\nSubsetting sub-matrices\n-------------------------")
print("Matrix with row indices 1 and 2 and column indices 3 and 4\n", mat[1:3,3:5])

```

```
Subsetting sub-matrices
-------------------------
Matrix with row indices 1 and 2 and column indices 3 and 4
 [[13 74]
 [45 37]]
```

In [290]:

```
print("Matrix with row indices 0 and 1 and column indices 1 and 3\n", mat[0:2,[1,3]])
```

```
Matrix with row indices 0 and 1 and column indices 1 and 3
 [[32 33]
 [97 13]]
```

Subseting

In [291]:

```
mat = np.array(ri(10,100,15)).reshape(3,5)
print("Matrix of random 2-digit numbers\n------------------------------\n",mat)
mat>50
```

```
Matrix of random 2-digit numbers
------------------------------
 [[26 89 37 37 96]
 [37 34 39 84 35]
 [53 48 99 20 22]]
```

Out[291]:

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

In [292]:

```
print ("Elements greater than 50\n", mat[mat>50])
```

```
Elements greater than 50
 [89 96 84 53 99]
```

Slicing

In [293]:

```
mat = np.array([[11,12,13],[21,22,23],[31,32,33]])
print("Original matrix")
print(mat)

```

```
Original matrix
[[11 12 13]
 [21 22 23]
 [31 32 33]]
```

In [295]:

```
mat_slice = mat[:2,:2]
print ("\nSliced matrix")
print(mat_slice)
print ("\nChange the sliced matrix")
```

```
Sliced matrix
[[11 12]
 [21 22]]

Change the sliced matrix
```

In [296]:

```
mat_slice[0,0] = 1000
print (mat_slice)
```

```
[[1000   12]
 [  21   22]]
```

In [298]:

```
print("\nBut the original matrix? WHOA! It got changed too!")
print(mat)

```

```
But the original matrix? WHOA! It got changed too!
[[1000   12   13]
 [  21   22   23]
 [  31   32   33]]
```

In [177]:

```
# Little different way to create a copy of the slixed matrix
print ("\nDoing it again little differently now...\n")
mat = np.array([[11,12,13],[21,22,23],[31,32,33]])
print("Original matrix")
print(mat)

```

```
Doing it again little differently now...

Original matrix
[[11 12 13]
 [21 22 23]
 [31 32 33]]
```

In [302]:

```
mat_slice = np.array(mat[:2,:2]) # Notice the np.array command to create a new array nc
print ("\nSliced matrix")
print(mat_slice)

```

```
Sliced matrix
[[1000   12]
 [  21   22]]
```

In [303]:

```
print ("\nChange the sliced matrix")
mat_slice[0,0] = 1000
print (mat_slice)

```

```
Change the sliced matrix
[[1000   12]
 [  21   22]]
```

In [304]:

```
print("\nBut the original matrix? NO CHANGE this time:)")
print(mat)
```

```
But the original matrix? NO CHANGE this time:)
[[1000   12   13]
 [  21   22   23]
 [  31   32   33]]
```

Universal Functions

In [306]:

```
mat1 = np.array(ri(1,10,9)).reshape(3,3)
mat2 = np.array(ri(1,10,9)).reshape(3,3)
print("\n1st Matrix of random single-digit numbers\n----------------------------------
print("\n2nd Matrix of random single-digit numbers\n----------------------------------

```

```
1st Matrix of random single-digit numbers
---------------------------------------
 [[6 8 8]
 [4 1 7]
 [9 1 7]]

2nd Matrix of random single-digit numbers
---------------------------------------
 [[2 2 3]
 [3 4 3]
 [5 5 9]]
```

In [316]:

```
mat1+mat2   # element wise multiplication
```

Out[316]:

```
array([[ 8, 10, 11],
       [ 7,  5, 10],
       [14,  6, 16]])
```

In [318]:

```
# actual multiplication of matrix
mat1@mat2
```

Out[318]:

```
array([[ 76,  84, 114],
       [ 46,  47,  78],
       [ 56,  57,  93]])
```

In [319]:

```
mat1+mat2
```

Out[319]:

```
array([[ 8, 10, 11],
       [ 7,  5, 10],
       [14,  6, 16]])
```

In [320]:

```
print("\nDivision\n------------------\n", mat1/0)
#print("\nLineaer combination: 3*A - 2*B\n----------------------------\n", 3*mat1-2*mc

```

```
Division
------------------
 [[inf inf inf]
 [inf inf inf]
 [inf inf inf]]

<ipython-input-320-ea15cab37cb9>:1: RuntimeWarning: divide by zero encounter
ed in true_divide
  print("\nDivision\n------------------\n", mat1/0)
```

In [321]:

```
print("\nAddition of a scalar (100)\n-------------------------\n", 100+mat1)

```

```
Addition of a scalar (100)
-------------------------
 [[106 108 108]
 [104 101 107]
 [109 101 107]]
```

In [322]:

```
print("\nExponentiation, matrix cubed here\n---------------------------------------\n"
print("\nExponentiation, sq-root using pow function\n-----------------------------------
```

```
Exponentiation, matrix cubed here
---------------------------------------
 [[216 512 512]
 [ 64   1 343]
 [729   1 343]]

Exponentiation, sq-root using pow function
-----------------------------------------
 [[216 512 512]
 [ 64   1 343]
 [729   1 343]]
```

Broadcasting

In [324]:

```
#NumPy operations are usually done on pairs of arrays on an element-by-element basis.
#In the simplest case, the two arrays must have exactly the same shape.
#NumPy's broadcasting rule relaxes this constraint when the arrays' shapes meet certain
#When operating on two arrays, NumPy compares their shapes element-wise. It starts with
#dimensions, and works its way forward. Two dimensions are compatible when
#they are equal, or one of them is 1
```

In [323]:

```
start = np.zeros((4,4))
start= start+100
start
```

Out[323]:

```
array([[100., 100., 100., 100.],
       [100., 100., 100., 100.],
       [100., 100., 100., 100.],
       [100., 100., 100., 100.]])
```

In [325]:

```
# create a rank 1 ndarray with 3 values
add_rows = np.array([1, 0, 2,5])
print(add_rows)
```

```
[1 0 2 5]
```

In [326]:

```
y = start + add_rows  # add to each row of 'start' using broadcasting
print(y)
```

```
[[101. 100. 102. 105.]
 [101. 100. 102. 105.]
 [101. 100. 102. 105.]
 [101. 100. 102. 105.]]
```

In [328]:

```
# create an ndarray which is 4 x 1 to broadcast across columns
add_cols = np.array([[0,1,2,3]])
add_cols = add_cols.T
print(add_cols)
```

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

In [329]:

```
# add to each column of 'start' using broadcasting
y = start + add_cols
print(y)
```

```
[[100. 100. 100. 100.]
 [101. 101. 101. 101.]
 [102. 102. 102. 102.]
 [103. 103. 103. 103.]]
```

In [330]:

```
# this will just broadcast in both dimensions
add_scalar = np.array([100])
print(start+y)
```

```
[[200. 200. 200. 200.]
 [201. 201. 201. 201.]
 [202. 202. 202. 202.]
 [203. 203. 203. 203.]]
```

Array Math

In [331]:

```
mat1 = np.array(ri(1,10,9)).reshape(3,3)
mat2 = np.array(ri(1,10,9)).reshape(3,3)
print("\n1st Matrix of random single-digit numbers\n\n",mat1)
print("\n2nd Matrix of random single-digit numbers\n------------\n",mat2)

```

```
1st Matrix of random single-digit numbers

 [[1 9 3]
 [9 1 9]
 [7 3 7]]

2nd Matrix of random single-digit numbers
------------
 [[3 1 6]
 [6 2 4]
 [5 3 7]]
```

In [335]:

```
np.sqrt(3)
```

Out[335]:

```
1.7320508075688772
```

In [336]:

```
print("\nSq-root of 1st matrix using np\n------------------\n", np.sqrt(mat1))

```

```
Sq-root of 1st matrix using np
------------------
 [[1.         3.         1.73205081]
 [3.         1.         3.        ]
 [2.64575131 1.73205081 2.64575131]]
```

In [338]:

```
print("\nExponential power of 1st matrix using np\n",'-'*50,"\n", np.exp(mat1))

```

```
Exponential power of 1st matrix using np
 --------------------------------------------------
 [[2.71828183e+00 8.10308393e+03 2.00855369e+01]
 [8.10308393e+03 2.71828183e+00 8.10308393e+03]
 [1.09663316e+03 2.00855369e+01 1.09663316e+03]]
```

In [339]:

```
print("\n10-base logarithm on 1st matrix using np\n",'-'*50,"\n", np.log10(mat1))
print(mat1)
print(mat2)
```

```
10-base logarithm on 1st matrix using np
 --------------------------------------------------
 [[0.         0.95424251 0.47712125]
 [0.95424251 0.         0.95424251]
 [0.84509804 0.47712125 0.84509804]]
[[1 9 3]
 [9 1 9]
 [7 3 7]]
[[3 1 6]
 [6 2 4]
 [5 3 7]]
```

In [341]:

```
print("\nModulo reminder using np\n",'-'*50,"\n", np.fmod(mat1,mat2))
mat1%mat2
```

```
Modulo reminder using np
 --------------------------------------------------
 [[1 0 3]
 [3 1 1]
 [2 0 0]]
```

Out[341]:

```
array([[1, 0, 3],
       [3, 1, 1],
       [2, 0, 0]], dtype=int32)
```

In [186]:

```
print("\nCombination of functions by shwoing exponetial decay of a sine wave\n",'-'*70)

```

```
Combination of functions by shwoing exponetial decay of a sine wave
 ----------------------------------------------------------------------
```

In [207]:

```
A = np.linspace(0,12*np.pi,1001)

```

In [208]:

```
A
```

Out[208]:

```
array([ 0.        ,  0.03769911,  0.07539822, ..., 37.62371362,
       37.66141273, 37.69911184])
```

In [343]:

```
np.pi  # It is the value of pie
```

Out[343]:

```
3.141592653589793
```

In [ ]:

```

```