Numpy Array

NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed. This tutorial explains the basics of NumPy such as its architecture and environment. It also discusses the various array functions, types of indexing, etc. An introduction to Matplotlib is also provided. All this is explained with the help of examples for better understanding.

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

Out[2]:
```
numpy.ndarray
```

In [2]:
```
#Upcasting:
```

In [3]:
```
np.array([1, 2, 3.0])
```

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

In [4]:
```
#two dimensions
```

In [4]:
```
 np.array([[1, 2], [3, 4]])
```

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

In [6]:
```
#Minimum dimensions 2:
```

In [6]:
```
np.array([1, 2, 3], ndmin=5)
```

Out[6]:
```
array([[[[[1, 2, 3]]]]])
```

In [8]:
```
#dtype
```

In [7]:
```
 np.array([1, 2, 3], dtype=complex)
```

Out[7]:
```
array([1.+0.j, 2.+0.j, 3.+0.j])
```

In [12]:
```
#Data-type consisting of more than one element:
```

In [15]:
```
x = np.array([(1,2),(3,4)],dtype=[('a','<i2'),('b','<i8')])
x
```

Out[15]:
```
array([(1, 2), (3, 4)], dtype=[('a', '<i2'), ('b', '<i8')])
```

In [39]:
```
type(x[1][0])
```

Out[39]:
```
numpy.int16
```

In [13]:
```
#Creating an array from sub-classes:
```

In [17]:
```
np.mat(np.array([[1, 2],[4,7]]))
```

Out[17]:
```
matrix([[1, 2],
        [4, 7]])
```

In [21]:
```
np.mat('1 2; 3 4')
```

Out[21]:
```
matrix([[1, 2],
        [3, 4]])
```

numpy.asarray

In [16]:
```
#Convert the input to an array.
```

In [18]:
```
#Convert a list into an array:
```

In [27]:
```
 a = [1, 2]
type(a)
```

Out[27]:
```
list
```

In [25]:
```
type(np.asarray(a))
```

Out[25]:
```
numpy.ndarray
```

In [31]:
```
a = np.array([1, 2]) #Existing arrays are not copied
type(a)
```

Out[31]:
```
numpy.ndarray
```

In [42]:
```
np.asarray((1,2))
```

Out[42]:
```
array([1, 2])
```

In [25]:
```
#If dtype is set, array is copied only if dtype does not match:
```

In [43]:
```
a = np.array([1, 2], dtype=np.float32)
a
```

Out[43]:
```
array([1., 2.], dtype=float32)
```

In [33]:
```
np.asarray([1,2]) is a
```

Out[33]:
```
False
```

In [36]:
```
np.asarray([1,2])
```

Out[36]:
```
array([1, 2])
```

In [34]:
```
np.asarray(a, dtype=np.float64) is a
```

Out[34]:
```
False
```

In [29]:
```
# ndarray subclasses are not passed through
```

In [44]:
```
issubclass(np.matrix, np.ndarray)
```

Out[44]:
```
True
```

In [48]:
```
a = np.matrix([[1, 2]])
np.asanyarray(a)
```

Out[48]:
```
matrix([[1, 2]])
```

In [49]:
```
np.asarray(a) is a
```

Out[49]:
```
False
```

In [50]:
```
np.asanyarray(a) is a
```

Out[50]:
```
True
```

In [51]:
```
type(np.asarray(a))
```

Out[51]:
```
numpy.ndarray
```

numpy.asanyarray

In [50]:
```
a = [1, 2]
np.asanyarray(a)
```

Out[50]:
```
array([1, 2])
```

In [86]:
```
a = np.matrix([1, 2])
a
```

Out[86]:
```
matrix([[1, 2]])
```

In [52]:
```
np.asanyarray([1,2]) is a
```

Out[52]:
```
True
```

#numpy.copy

In [52]:
```
np.array(a, copy=True)
```

Out[52]:
```
array([[1, 2]])
```

In [40]:
```
#Create an array x, with a reference y and a copy z:
```

In [56]:
```
x = np.array([1, 2, 3])
x
y
```

Out[56]:
```
array([1, 2, 3])
```

In [62]:
```
y = x
```

In [63]:
```
z = np.copy(x)
```

In [64]:
```
y[0] = 100
x
```

Out[64]:
```
array([100,   2,   3])
```

In [78]:
```
x
```

Out[78]:
```
array([100,   2,   3])
```

In [89]:
```
id(x[0])
```

Out[89]:
```
4767545360
```

In [91]:
```
id(y[0])
```

Out[91]:
```
4767545408
```

In [ ]:

In [92]:
```
id(x)
```

Out[92]:
```
4768156128
```

In [93]:
```
id(y)
```

Out[93]:
```
4768156128
```

numpy.fromfunction

In [58]:
```
#Construct an array by executing a function over each coordinate.
```

In [94]:
```
np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
```

Out[94]:
```
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]])
```

In [119…
```
np.fromfunction(lambda i, j: i * j, (3, 3), dtype=int)
```

Out[119…
```
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
```

In [61]:
```
#Create a new 1-dimensional array from an iterable object.
```

In [97]:
```
iterable = (x*x for x in range(5))
iterable
```

Out[97]:
```
<generator object <genexpr> at 0x11c326b10>
```

In [98]:
```
np.fromiter(iterable, float)
```

Out[98]:
```
array([ 0.,  1.,  4.,  9., 16.])
```

In [64]:
```
#A new 1-D array initialized from text data in a string
```

In [101…
```
a = np.fromstring('234 234',sep=' ')
a
```

Out[101…
```
array([234., 234.])
```

In [102…
```
np.fromstring('1, 2', dtype=int, sep=',')
```

Out[102…
```
array([1, 2])
```

In [79]:
```
# How to create create a record array from a (flat) list of arrays
```

In [103…
```
x1=np.array([1,2,3,4])
```

In [116…
```
x2=np.array(['a','dd','xyz','12'])
```

In [117…
```
x3=np.array([1.1,2,3,4])
x4=np.array([1.1,2,3,4])
type(x4)
```

Out[117…
```
numpy.ndarray
```

In [118…
```
r = np.core.records.fromarrays([x1,x2,x3,x4],names='a,b,c,d')
r
```

Out[118…
```
rec.array([(1, 'a', 1.1, 1.1), (2, 'dd', 2. , 2. ), (3, 'xyz', 3. , 3. ),
           (4, '12', 4. , 4. )],
          dtype=[('a', '<i8'), ('b', '<U3'), ('c', '<f8'), ('d', '<f8')])
```

In [112…
```
print(r[1]["a"])
```

```
2
```

x1[1]=34

In [130…
```
x1[1]
```

Out[130…
```
2
```

data types

In [120…
```
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 [121…
```
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 a
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 a
print("Data type of this matrix: ", mat.dtype,sep='') #dtype gives the data type contai
```

```
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: int64
```

In [122…
```
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 [111…
```
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 [127…
```
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[127…
```
array([ 5. ,  7.3,  9.6, 11.9, 14.2])
```

In [128…
```
list(range(5,16,2))
```

Out[128…
```
[5, 7, 9, 11, 13, 15]
```

In [129…
```
list(range(50,-1,5))
```

Out[129…
```
[]
```

In [135…
```
print("Numbers spaced apart by 2:",np.arange(50,-1,5)) # Numbers spaced apart by 2
```

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

In [131…
```
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 [132…
```
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 [134…
```
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 [136…
```
print("Vector of zeroes\n---------------------")
print(np.zeros(5))
```

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

In [139…
```
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 [127…
```
print("Vector of ones\n---------------------")
print(np.ones(5))
```

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

In [140…
```
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 [147…
```
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 [148…
```
print("Empty matrix\n-------------\n", np.empty((3,5)))
```

```
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 [132…
```
np.arange(3)
```

Out[132…
```
array([0, 1, 2])
```

In [80]:
```
np.arange(3.0)
```

Out[80]:
```
array([0., 1., 2.])
```

In [81]:
```
 np.arange(3,7)
```

Out[81]:
```
array([3, 4, 5, 6])
```

In [82]:
```
np.arange(3,7,2)
```

Out[82]:
```
array([3, 5])
```

In [150…
```
np.linspace(2.0, 3.0, num=5)
```

Out[150…
```
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
```

In [151…
```
np.linspace(2.0, 3.0, num=5, endpoint=False)
```

Out[151… array([2. , 2.2, 2.4, 2.6, 2.8])

In [153…
```
np.linspace(2.0, 3.0, num=9, retstep=True)
```

Out[153… (array([2. , 2.125, 2.25 , 2.375, 2.5 , 2.625, 2.75 , 2.875, 3.]), 0.125)

In [147…
```
#Return numbers spaced evenly on a log scale.
np.linspace(2.0, 3.0, num=4)
```

Out[147… array([2. , 2.33333333, 2.66666667, 3.])

In [154…
```
np.logspace(2.0, 3.0, num=4,base = 10)
```

Out[154… array([100. , 215.443469 , 464.15888336, 1000.])

In [113…
```
np.logspace(2.0, 3.0, num=4, endpoint=False)
```

Out[113… array([100. , 177.827941 , 316.22776602, 562.34132519])

In [155…
```
np.logspace(2.0, 3.0, num=4, base=2.0)
```

Out[155… array([4. , 5.0396842 , 6.34960421, 8.])

In [115…
```
#Extract a diagonal or construct a diagonal array.
```

In [179…
```
x = np.arange(16).reshape((-1,4))
```

In [180…
```
x
```

Out[180…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [172…
```
np.diag(x)
```

Out[172… array([0, 5, 10, 15])

In [182…
```
np.diag(x, k=2
        )
```

Out[182… array([2, 7])

In [183…
```
np.diag(x, k=-1)
```

Out[183… array([4, 9, 14])

In [184…
```
np.diag(np.diag(x))
```

Out[184…
```
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 [180…
```
np.diagflat([[1,2], [3,4]])
```

Out[180…
```
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

In [124…
```
np.diagflat([1,2], 1)
```

Out[124…
```
array([[0, 1, 0],
       [0, 0, 2],
       [0, 0, 0]])
```

In [125…
```
#An array with ones at and below the given diagonal and zeros elsewhere.
```

In [187…
```
np.tri(3, 5, 1, dtype=int)
```

Out[187…
```
array([[1, 1, 0, 0, 0],
       [1, 1, 1, 0, 0],
       [1, 1, 1, 1, 0]])
```

In [191…
```
np.tri(3, 5,k=-1)
```

Out[191…
```
array([[0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [1., 1., 0., 0., 0.]])
```

In [128…
```
#return a Lower triangle of an array.
```

In [205…
```
np.tril([[1,2,3],[4,5,6],[7,8,9]], 0)
```

Out[205…
```
array([[1, 0, 0],
       [4, 5, 0],
       [7, 8, 9]])
```

In [130…
```
#return Upper triangle of an array.
```

In [206…
```
np.triu([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], 1)
```

Out[206…
```
array([[0, 2, 3],
       [0, 0, 6],
       [0, 0, 0],
       [0, 0, 0]])
```

Random number generation

In [197…
```
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)
[[0.79234365 0.26636652 0.84876143]
 [0.78766627 0.06627396 0.8829639 ]]
```

In [209…
```
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 n
ormal
[[ 0.08550888  0.73651756  0.08329424]
```

```
[ 0.75179279  0.42441618  0.47752353]
[ 0.23835886  1.06582108 -0.01790105]
[ 0.48190152 -0.24940656 -0.44992115]]
```

In [202…
```
print("Random integer vector:",np.random.randint(1,10)) #randint (low, high, # of sampl
print ("\nRandom integer matrix")
```

```
Random integer vector: 4

Random integer matrix
```

In [208…
```
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
```

```
[[28 90 28 25]
 [14 51 32 98]
 [10 98 11 71]
 [12 42 54 77]]

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

Reshaping

In [211…
```
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[211…
```
array([[51, 63, 51, 54, 96],
       [11, 93,  2, 51, 34],
       [62, 35, 42, 95, 58],
       [88, 44, 36, 93, 83],
       [86, 56, 30,  2, 15],
       [68, 40, 77, 48, 57]])
```

In [209…
```
print ("Shape of a:", a.shape)
print ("Shape of b:", b.shape)
print ("Shape of c:", c.shape)
```

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

In [219…
```
print("\na looks like\n",'-'*20,"\n",a,"\n",'-'*20)
print("\nb looks like\n",'-'*20,"\n",b,"\n",'-'*20)
print("\nc looks like\n",'-'*20,"\n",c,"\n",'-'*20)
```

```
a looks like
 --------------------
 [72 69 61 67 42  2 92 42 38 56 22 71 21 81 81 47  2  1  6 94 52 14 87 71
 38 57 99 87 62 88]
 --------------------

b looks like
 --------------------
 [[[72 69 61 67 42]
  [ 2 92 42 38 56]
  [22 71 21 81 81]]

 [[47  2  1  6 94]
```

```
 [52 14 87 71 38]
 [57 99 87 62 88]]]
 --------------------

c looks like
 --------------------
 [[72 69 61 67 42]
 [ 2 92 42 38 56]
 [22 71 21 81 81]
 [47  2  1  6 94]
 [52 14 87 71 38]
 [57 99 87 62 88]]
 --------------------
```

In [212…
```
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
 --------------------------------------------------
 [69 99 45 72 41 16 50 80 80 22]

Here is the sorted vector
 --------------------------------------------------
 [16 22 41 45 50 69 72 80 80 99]
```

In [221…
```
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 is the sorted matrix along each column
 --------------------------------------------------
 [[18 25 38 73 94]
 [18 42 75 79 91]
 [21 44 62 78 85]
 [15 17 44 89 91]
 [ 3 24 38 62 86]]
```

Out[221…
```
array([[18, 94, 25, 38, 73],
       [75, 42, 79, 18, 91],
       [21, 44, 85, 78, 62],
       [91, 15, 17, 44, 89],
       [62, 38,  3, 24, 86]])
```

In [214…
```
print("Max of a:", M.max())
print("Max of b:", b.max())
b
```

```
Max of a: 99
Max of b: 96
```

Out[214…
```
array([[[51, 63, 51, 54, 96],
        [11, 93,  2, 51, 34],
        [62, 35, 42, 95, 58]],

       [[88, 44, 36, 93, 83],
        [86, 56, 30,  2, 15],
        [68, 40, 77, 48, 57]]])
```

In [217…
```
M
```

Out[217…
```
array([[46, 88, 23, 66, 57],
       [50, 90, 30, 46, 36],
```

```
       [31, 19, 42, 23, 31],
       [ 4, 44, 86, 10, 81],
       [73, 35, 30, 99, 43]])
```

In [218…
```
print("Max of a location:", M.argmax(axis= 1 ))
print("Max of b location:", b.argmax())
print("Max of c location:", b.argmax())
```

```
Max of a location: [1 1 2 2 3]
Max of b location: 4
Max of c location: 4
```

Indexing and slicing

In [229…
```
arr = np.arange(0,11)
print("Array:",arr)
```

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

In [161…
```
print("Element at 7th index is:", arr[7])
```

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

In [230…
```
print("Elements from 3rd to 5th index are:", arr[3:6:2])
```

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

In [231…
```
print("Elements up to 4th index are:", arr[:4])
arr
```

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

Out[231…
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [233…
```
print("Elements from last backwards are:", arr[-1:7:-1])
```

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

In [164…
```
print("3 Elements from last backwards are:", arr[-1:-6:2])
```

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

In [165…
```
arr = np.arange(0,21,2)
print("New array:",arr)
```

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

In [64]:
```
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 [237…
```
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
--------------------------------
 [[35 75 44 59 85]
 [71 58 75 72 69]
 [76 97 49 63 53]]
```

Out[237…
```
array([[75, 59],
       [58, 72],
       [97, 63]])
```

In [238…
```
mat[0:3,[1,3]]
mat
```

Out[238…
```
array([[35, 75, 44, 59, 85],
       [71, 58, 75, 72, 69],
       [76, 97, 49, 63, 53]])
```

In [241…
```
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: 58
```

Out[241…
```
array([[35, 75, 44, 59, 85],
       [71, 58, 75, 72, 69],
       [76, 97, 49, 63, 53]])
```

In [242…
```
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: 75

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

In [172…
```
print("Entire row at index 2:", mat[2])
print("Entire column at index 3:", mat[:,3])
```

```
Entire row at index 2: [22 10 57 59 61]
Entire column at index 3: [92 88 59]
```

In [7]:
```
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
 [[14 32]
 [62 27]]
```

In [156…
```
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
 [[77 69]
 [64 17]]
```

Subseting

In [173…
```
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
```

```
--------------------------------
 [[51 99 67 32 43]
 [53 61 80 97 51]
 [82 76 18 26 11]]
```

Out[173…
```
array([[ True,  True,  True, False, False],
       [ True,  True,  True,  True,  True],
       [ True,  True, False, False, False]])
```

In [174…
```
print ("Elements greater than 50\n", mat[mat>50])
```

```
Elements greater than 50
 [51 99 67 53 61 80 97 51 82 76]
```

Slicing

In [175…
```
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 [176…
```
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 [18]:
```
mat_slice[0,0] = 1000
print (mat_slice)
```

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

In [75]:
```
print("\nBut the original matrix? WHOA! It got changed too!")
print(mat)
```

```
But the original matrix? WHOA! It got changed too!
[[11 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 [21]:
```
mat_slice = np.array(mat[:2,:2]) # Notice the np.array command to create a new array no
print ("\nSliced matrix")
print(mat_slice)
```

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

In [178…
```
print ("\nChange the sliced matrix")
mat_slice[0,0] = 1000
print (mat_slice)
```

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

In [22]:
```
print("\nBut the original matrix? NO CHANGE this time:)")
print(mat)
```

```
But the original matrix? NO CHANGE this time:)
[[11 12 13]
 [21 22 23]
 [31 32 33]]
```

Universal Functions

In [247…
```
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
----------------------------------------
 [[3 3 9]
 [6 1 8]
 [5 9 1]]

2nd Matrix of random single-digit numbers
----------------------------------------
 [[3 5 7]
 [4 4 9]
 [1 2 5]]
```

In [248…
```
mat1*mat2
```

Out[248…
```
array([[ 9, 15, 63],
       [24,  4, 72],
       [ 5, 18,  5]])
```

In [ ]:
```

```

In [249…
```
#print("\nAddition\n------------------\n", mat1+mat2)
print("\nMultiplication\n------------------\n", mat1@mat2)
```

```
Multiplication
------------------
 [[ 30  45  93]
 [ 30  50  91]
 [ 52  63 121]]
```

In [251…
```
print("\nDivision\n------------------\n", mat1/0)
#print("\nLineaer combination: 3*A - 2*B\n----------------------------\n", 3*mat1-2*ma
```

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

```
/Users/sudhanshukumar/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1: Run
timeWarning: divide by zero encountered in true_divide
  """Entry point for launching an IPython kernel.
```

In [252…
```
print("\nAddition of a scalar (100)\n------------------------\n", 100+mat1)
```

```
Addition of a scalar (100)
------------------------
 [[103 103 109]
 [106 101 108]
 [105 109 101]]
```

In [253…
```
print("\nExponentiation, matrix cubed here\n---------------------------------------\n"
print("\nExponentiation, sq-root using pow function\n-----------------------------------
```

```
Exponentiation, matrix cubed here
---------------------------------------
 [[ 27  27 729]
 [216   1 512]
 [125 729   1]]

Exponentiation, sq-root using pow function
-------------------------------------------
 [[ 27  27 729]
 [216   1 512]
 [125 729   1]]
```

Broadcasting

In [174…
```
#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 [254…
```
start = np.zeros((4,4))
start= start+100
start
```

Out[254…
```
array([[100., 100., 100., 100.],
       [100., 100., 100., 100.],
       [100., 100., 100., 100.],
       [100., 100., 100., 100.]])
```

In [256…
```
# create a rank 1 ndarray with 3 values
add_rows = np.array([1, 0, 2,5])
print(add_rows)
```

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

In [257…
```
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 [262…
```
# 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 [263…
```
# 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 [264…
```
# 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 [265…
```
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

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

2nd Matrix of random single-digit numbers
------------
 [[4 8 6]
 [7 6 8]
 [4 6 5]]
```

In [266…
```
print("\nSq-root of 1st matrix using np\n-----------------\n", np.sqrt(mat1))
```

```
Sq-root of 1st matrix using np
------------------
 [[3.         2.23606798 2.        ]
 [1.         2.64575131 1.73205081]
 [2.82842712 1.73205081 2.23606798]]
```

In [267…
```
print("\nExponential power of 1st matrix using np\n",'-'*50,"\n", np.exp(mat1))
```

```
Exponential power of 1st matrix using np
 --------------------------------------------------
 [[8.10308393e+03 1.48413159e+02 5.45981500e+01]
 [2.71828183e+00 1.09663316e+03 2.00855369e+01]
 [2.98095799e+03 2.00855369e+01 1.48413159e+02]]
```

In [268…
```
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
 --------------------------------------------------
 [[0.95424251 0.69897    0.60205999]
 [0.         0.84509804 0.47712125]
 [0.90308999 0.47712125 0.69897   ]]
[[9 5 4]
 [1 7 3]
 [8 3 5]]
[[4 8 6]
 [7 6 8]
 [4 6 5]]
```

In [270…
```
print("\nModulo reminder using np\n",'-'*50,"\n", np.fmod(mat1,mat2))
mat1%mat2
```

```
Modulo reminder using np
 --------------------------------------------------
 [[1 5 4]
 [1 1 3]
 [0 3 0]]
```

Out[270…
```
array([[1, 5, 4],
       [1, 1, 3],
       [0, 3, 0]])
```

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])
```