NumPy, short for Numerical Python, is the fundamental package required for high performance scientific computing and data analysis.

Numpy Provides:

1. **Extension package to Python for multi-dimensional arrays**
2. **Highly Efficient**
3. **Designed for Scientific Computation**

In [23]:

```
# Importing Numpy

import numpy as np
```

In [24]:

```
# Initializing an array in Numpy

a = np.array([2,3,4,5])
```

In [25]:

```
a
```

Out[25]:

```
array([2, 3, 4, 5])
```

In [26]:

```
type(a)
```

Out[26]:

```
numpy.ndarray
```

In [27]:

```
a.ndim
```

Out[27]:

```
1
```

In [28]:

```
a.shape
```

Out[28]:

```
(4,)
```

In [29]:

```
#Upcasting:

a = np.array([2,3,4,5.0])
```

In [30]:

```
a
```

Out[30]:

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

In [31]:

```
# Initializing 2D and 3D array

b = np.array([[2,4,5], [6,8,10]])

b
```

Out[31]:

```
array([[ 2,  4,  5],
       [ 6,  8, 10]])
```

In [32]:

```
b.ndim
```

Out[32]:

```
2
```

In [33]:

```
b.shape  # returns the shape of the matrix
```

Out[33]:

```
(2, 3)
```

In [34]:

```
len(b)   #returns the first dimension of the matrix
```

Out[34]:

```
2
```

In [35]:

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

In [36]:

```
c.ndim
```

Out[36]:

```
3
```

In [37]:

```
c
```

Out[37]:

```
array([[[1, 2],
        [3, 5]],

       [[4, 6],
        [7, 9]]])
```

In [38]:

```
c.shape
```

Out[38]:

```
(2, 2, 2)
```

Swallow Copy and Deep Copy

In [39]:

```
a = np.array([4,5,6,7,8])
```

```
a = np.array([4,5,6,7,8])
```

In [40]:

```
b = a  #swallow copy
```

In [41]:

```
b
```

Out[41]:

```
array([4, 5, 6, 7, 8])
```

In [42]:

```
b[1] = 50
```

In [43]:

```
b, a
```

Out[43]:

```
(array([ 4, 50,  6,  7,  8]), array([ 4, 50,  6,  7,  8]))
```

In [44]:

```
c = np.copy(a)  #deep copy
```

In [45]:

```
c
```

Out[45]:

```
array([ 4, 50,  6,  7,  8])
```

In [46]:

```
c[1] = 20
```

In [47]:

```
c
```

Out[47]:

```
array([ 4, 20,  6,  7,  8])
```

In [48]:

```
a
```

Out[48]:

```
array([ 4, 50,  6,  7,  8])
```

Numpy.fromfunction

In [49]:

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

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

Out[49]:

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

In [50]:

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

Out[50]:

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

In [51]:

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

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

Out[51]:

```
<generator object <genexpr> at 0x00000284D4B90848>
```

In [52]:

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

Out[52]:

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

arange and linspace

In [53]:

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

Out[53]:

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

In [54]:

```
np.arange(2,11,2)   #start, end and step
```

Out[54]:

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

In [55]:

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

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

Difference between range and arange function is in range function you cannot use floating point number but in arange function you can use floating point number.

In [56]:

```
# Now let's check why Numpy is more efficient

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

```
838 µs ± 113 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [57]:

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

```
4.39 µs ± 280 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [58]:

```
# linspace

l = np.linspace(0, 1, 6) #start, end, number of points

l
```

Out[58]:

```
array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

In [59]:

```
np.linspace(1,5,20, retstep = True)
```

Out[59]:

```
(array([1.        , 1.21052632, 1.42105263, 1.63157895, 1.84210526,
        2.05263158, 2.26315789, 2.47368421, 2.68421053, 2.89473684,
        3.10526316, 3.31578947, 3.52631579, 3.73684211, 3.94736842,
        4.15789474, 4.36842105, 4.57894737, 4.78947368, 5.        ]),
 0.21052631578947367)
```

Some Common Arrays

In [60]:

```
z = np.zeros((3,3))
```

In [61]:

```
z
```

Out[61]:

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

In [62]:

```
o = np.ones((3,4))

o
```

Out[62]:

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

In [63]:

```
e = np.eye(3)    # Return a 2D array with ones on the diagnal and zeros elsewhere
```

In [64]:

```
e
```

Out[64]:

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

In [65]:

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

Out[65]:

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

In [66]:

```
# Create array with diag function

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

In [67]:

```
a
```

Out[67]:

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

Random Number Generation

In [68]:

```
ru = np.random.rand(4,3)   # 4 by 3 matrix with random numbers from uniform distribution
ranging from 0 to 1
ru
```

Out[68]:

```
array([[0.43071054, 0.18309457, 0.6763637 ],
       [0.2324718 , 0.76673858, 0.21594217],
       [0.85737872, 0.19450292, 0.25749114],
       [0.24194685, 0.44762387, 0.16450667]])
```

In [69]:

```
rs = np.random.randn(4,3)   # 4 by 3 matrix with random numbers from standard normal dist
ribution
rs
```

Out[69]:

```
array([[ 1.16404869, -0.70384643,  0.9427874 ],
       [-0.16482234, -0.64323706,  2.05347351],
       [ 2.16266688,  0.8376387 ,  0.37162964],
       [ 0.74644728, -0.81054121,  0.32492798]])
```

In [70]:

```
ri = np.random.randint(1,25,(4,3))   # (low, high, # of samples to be drawn in a tuple to
form a matrix) generating random integers

ri
```

Out[70]:

```
array([[12, 12, 13],
       [22, 17, 13],
       [ 3, 20, 24],
       [12, 11,  7]])
```

Indexing and slicing

In [71]:

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

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

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

In [72]:

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

Element at 7th index is: 7

In [73]:

```
print("Elements from 3rd to 11th index are:", arr[3:10])
```

Elements from 3rd to 11th index are: [3 4 5 6 7 8 9]

In [74]:

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

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

Out[74]:

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

In [75]:

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

Elements from last backwards are: [10 9 8 7 6 5]

In [76]:

```
arr1 = np.random.randint(1,25,(4,3))
```

In [77]:

```
arr1
```

Out[77]:

```
array([[12,  4, 11],
       [ 5, 16, 15],
       [18,  9, 21],
       [ 2,  3, 22]])
```

In [78]:

```
arr1[:2,:2]
```

Out[78]:

```
array([[12,  4],
       [ 5, 16]])
```

In [79]:

```
arr1[2:4,::3]
```

Out[79]:

```
array([[18],
       [ 2]])
```

Boolean Masking

In [80]:

```
b = np.random.randint(1,10,(4,3))

b
```

Out[80]:

```
array([[1, 1, 7],
       [8, 9, 4],
       [7, 1, 1],
       [2, 4, 4]])
```

In [81]:

```
b>5
```

Out[81]:

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

In [82]:

```
b[b>5]
```

Out[82]:

```
array([7, 8, 9, 7])
```

Reshaping

In [83]:

```
r = np.random.randint(1,10,(5,4))

r
```

Out[83]:

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

In [84]:

```
r.shape
```

Out[84]:

```
(5, 4)
```

In [85]:

```
r.reshape(4,5)
```

Out[85]:

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

In [86]:

```
r.reshape(10,2)
```

Out[86]:

```
array([[9, 7],
       [9, 9],
       [7, 2],
       [6, 5],
       [2, 1],
       [2, 2],
```

```
       [4, 9],
       [8, 8],
       [9, 9],
       [8, 9]])
```

In [87]:

```
r.reshape(20,1)
```

Out[87]:

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

Array Math

In [88]:

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

In [89]:

```
x+y
```

Out[89]:

```
array([[ 6,  8],
       [10, 12]])
```

In [90]:

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

Out[90]:

```
array([[ 6,  8],
       [10, 12]])
```

In [91]:

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

Out[91]:

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

In [92]:

```
x*y                 #returns the result of element wise multiplication
```

Out[92]:

```
array([[ 5, 12],
```

```
        [21, 32]])
```

In [93]:

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

Out[93]:

```
array([[ 5, 12],
       [21, 32]])
```

In [94]:

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

Out[94]:

```
array([[0.2       , 0.33333333],
       [0.42857143, 0.5       ]])
```

In [95]:

```
x%y
```

Out[95]:

```
array([[1, 2],
       [3, 4]], dtype=int32)
```

In [96]:

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

Out[96]:

```
array([[1, 2],
       [3, 4]], dtype=int32)
```

In [97]:

```
np.sqrt(x)
```

Out[97]:

```
array([[1.        , 1.41421356],
       [1.73205081, 2.        ]])
```

In [98]:

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

Out[98]:

```
array([[19, 22],
       [43, 50]])
```

In [99]:

```
x@y              #returns the result of matrix multiplication
```

Out[99]:

```
array([[19, 22],
       [43, 50]])
```

In [100]:

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

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
```

```
[4 6]
[3 7]
```

In [101]:

```
np.log(x)
```

Out[101]:

```
array([[0.        , 0.69314718],
       [1.09861229, 1.38629436]])
```

In [ ]:

```

```

Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

In [102]:

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

Out[102]:

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

In [103]:

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

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

In [104]:

```
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 [105]:

```
# 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 [106]:

```
# 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.]]
```

argmin and argmax

In [107]:

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

Out[107]:

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

In [108]:

```
x.min()
```

Out[108]:

```
1
```

In [109]:

```
x.max()
```

Out[109]:

```
4
```

In [110]:

```
x.argmax()     #returns index of maximum value
```

Out[110]:

```
1
```

In [111]:

```
x.argmin()    #returns index of minimum value
```

Out[111]:

```
0
```

Statistical Functions

In [112]:

```
x = np.array([2,3,5,7])
x
```

Out[112]:

```
array([2, 3, 5, 7])
```

In [113]:

```
np.mean(x)
```

Out[113]:

```
4.25
```

In [114]:

```
np.median(x)
```

Out[114]:

Out[114]:

```
4.0
```

In [115]:

```
np.var(x)
```

Out[115]:

```
3.6875
```

In [116]:

```
np.std(x)
```

Out[116]:

```
1.920286436967152
```

In [117]:

```
marks = np.array([30,31,32,40,90,95,97,98,99,100])
```

In [118]:

```
np.percentile(marks, 40)
```

Out[118]:

```
70.0
```

Flattening

In [119]:

```
x = np.array([[3,4,5], [1,4,7]])
x
```

Out[119]:

```
array([[3, 4, 5],
       [1, 4, 7]])
```

In [120]:

```
x.ravel()   # return a contiguous flattened array
```

Out[120]:

```
array([3, 4, 5, 1, 4, 7])
```

In [121]:

```
x.T    #transpose
```

Out[121]:

```
array([[3, 1],
       [4, 4],
       [5, 7]])
```

In [122]:

```
x.T.ravel()
```

Out[122]:

```
array([3, 1, 4, 4, 5, 7])
```

Sorting Data

In [123]:

```
a = np.array([[3,5,6], [6,8,9]])
```

In [124]:

```
b = np.sort(a, axis = 1)
```

In [125]:

```
b
```

Out[125]:

```
array([[3, 5, 6],
       [6, 8, 9]])
```

In [126]:

```
a.sort(axis=1)
a
```

Out[126]:

```
array([[3, 5, 6],
       [6, 8, 9]])
```

In [127]:

```
# sorting with fancy indexing

a = np.array([4,3,5,1])
j = np.argsort(a)
j
```

Out[127]:

```
array([3, 1, 0, 2], dtype=int64)
```

In [128]:

```
a[j]
```

Out[128]:

```
array([1, 3, 4, 5])
```

Reading Image File

In [130]:

```
from PIL import Image
from IPython.display import display
```

In [131]:

```
im = Image.open("cover.jpeg")
```

In [132]:

```
display(im)
```

In [133]:

```
array = np.array(im)
array
```

Out[133]:

```
array([[[ 47,  53,  67],
        [ 47,  53,  67],
        [ 47,  53,  67],
        ...,
        [224, 216, 213],
        [225, 217, 214],
        [227, 219, 216]],

       [[ 47,  53,  67],
        [ 47,  53,  67],
        [ 47,  53,  67],
        ...,
        [224, 216, 213],
        [225, 217, 214],
        [227, 219, 216]],

       [[ 47,  53,  67],
        [ 47,  53,  67],
        [ 47,  53,  67],
        ...,
        [224, 216, 213],
        [225, 217, 214],
        [227, 219, 216]],

       ...,

       [[ 47,  53,  67],
        [ 47,  53,  67],
        [ 47,  53,  67],
        ...,
        [148, 102,  76],
        [148, 102,  76],
        [151, 105,  79]],

       [[ 47,  53,  67],
        [ 47,  53,  67],
        [ 47,  53,  67],
        ...,
        [141,  95,  71],
        [136,  90,  66],
        [148, 102,  78]],

       [[ 47,  53,  67],
        [ 47,  53,  67],
        [ 47,  53,  67],
        ...,
        [138,  94,  69],
        [134,  90,  65],
        [147, 103,  78]]], dtype=uint8)
```

In [134]:

```
array.shape
```

Out[134]:

```
(426, 1280, 3)
```

In [ ]:

```

```