Numpy

In [1]:
```
import numpy as np
```

1D array

In [2]:
```
a = np.array([10,20,30,50,60,0])
```

Attributes of arrays

dtype

ndim

size

shape

In [3]:
```
a.dtype
```

Out[3]:
```
dtype('int32')
```

In [4]:
```
a.shape
```

Out[4]:
```
(6,)
```

In [5]:
```
a.ndim
```

Out[5]:
```
1
```

In [6]:
```
a.size
```

Out[6]:
```
6
```

In [7]:
```
a[0]
```

Out[7]:
```
10
```

In [8]:
```
a[2]
```

Out[8]:
```
30
```

Multi dimensional array

In [9]:
```
a_2d = np.array([[10,20,30,], [23,55,15]])
```

In [10]:
```
a_2d
```

Out[10]:
```
array([[10, 20, 30],
       [23, 55, 15]])
```

In [11]:
```
a_2d.dtype
```

Out[11]:
```
dtype('int32')
```

In [12]:
```
a_2d.shape
```

Out[12]:
```
(2, 3)
```

In [13]:
```
a_2d.ndim
```

Out[13]:
```
2
```

In [14]:
```
a_2d.size
```

Out[14]:
```
6
```

In [15]:
```
a_2d[0]
```

Out[15]:
```
array([10, 20, 30])
```

Another type of multi dimensional arrays

In [16]:
```
another_type_array = np.array([[[1,2,3,4]]])
```

In [17]:
```
another_type_array
```

Out[17]:
```
array([[[1, 2, 3, 4]]])
```

In [18]:
```
another_type_array.shape
```

Out[18]:
```
(1, 1, 4)
```

In [19]:
```
another_type_array.ndim
```

Out[19]:
```
3
```

First element in first array

In [20]:
```
a_2d[0][0]
```

Out[20]:
```
10
```

Second element in second array

In [21]:
```
a_2d[1][1]
```

Out[21]:
```
55
```

Lets change its data type into float

In [22]:
```
a_2d = np.array([[10,20,30,], [23,55,15]], dtype = float)
```

In [23]:
```
a_2d
```

Out[23]:
```
array([[10., 20., 30.],
       [23., 55., 15.]])
```

Lets create a bigger array

In [24]:
```
a_bigger = np.array([
[
[ 10 , 20 , 30 , 40 ], [ 8 , 8 , 2 , 1 ], [ 1 , 1 , 1 , 2 ]
],
[
[ 9 , 9 , 2 , 39 ], [ 1 , 2 , 3 , 3 ], [ 0 , 0 , 3 , 2 ]
],
[
[ 12 , 33 , 22 , 1 ], [ 22 , 1 , 22 , 2 ], [ 0 , 2 , 3 , 1 ]
]
], dtype = float )
```

In [25]:
```
a_bigger.shape
```

Out[25]:
```
(3, 3, 4)
```

In [26]:
```
a_bigger.ndim
```

Out[26]:
```
3
```

In [27]:
```
a_bigger.size
```

Out[27]:
```
36
```

In [28]:
```
a_bigger[2][2][3]
```

Out[28]:
```
1.0
```

In [29]:
```
a_bigger[2][2]
```

Out[29]:
```
array([0., 2., 3., 1.])
```

Filling arrays

np.full

By using the full function for example, we fill an array of a certain shape with the same number. In this case we create a 3x5x4 matrix, which is filled with sevens.

In [30]:
```
a_full = np.full(( 3 , 5 , 4 ), 7 )
```

In [31]:
```
a_full
```

Out[31]:
```
array([[[7, 7, 7, 7],
        [7, 7, 7, 7],
        [7, 7, 7, 7],
        [7, 7, 7, 7],
        [7, 7, 7, 7]],

       [[7, 7, 7, 7],
        [7, 7, 7, 7],
        [7, 7, 7, 7],
        [7, 7, 7, 7],
        [7, 7, 7, 7]],

       [[7, 7, 7, 7],
        [7, 7, 7, 7],
        [7, 7, 7, 7],
        [7, 7, 7, 7],
        [7, 7, 7, 7]]])
```

In [32]:
```
a_full.ndim
```

Out[32]:
```
3
```

ZEROS AND ONES

For the cases that we want arrays full of zeros or ones, we even have specific functions.

In [33]:
```
a_zeros = np.zeros((3,3))
a_zeros
```

Out[33]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [34]:
```
b_ones = np.ones((2,3,4))
b_ones
```

Out[34]:
```
array([[[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 [35]:
```
b_ones.ndim
```

Out[35]:
```
3
```

EMPTY AND RANDOM

Other options would be to create an empty array or one that is filled with random numbers. For this, we use the respective functions once again.

In [36]:
```
e = np.empty((4,4))
e
```

Out[36]:
```
array([[4.67296746e-307, 1.69121096e-306, 1.11260755e-306,
        2.11395597e-307],
       [1.42418987e-306, 1.37961641e-306, 1.60220528e-306,
        1.24611266e-306],
       [9.34598925e-307, 1.24612081e-306, 1.11260755e-306,
        1.60220393e-306],
       [1.51320640e-306, 9.34609790e-307, 1.86921279e-306,
        1.24610723e-306]])
```

In [37]:
```
b_random = np.random.random((2,2))
b_random
```

Out[37]:
```
array([[0.0999359 , 0.27888808],
       [0.71268151, 0.03505873]])
```

In [38]:
```
b_random.ndim
```

Out[38]:
```
2
```

In [39]:
```
b_random.size
```

Out[39]:
```
4
```

RANGES

Instead of just filling arrays with the same values, we can fill create sequences of values by specifying the boundaries. For this, we can use two different functions, namely arange and linspace .

Arange

Arguements are : start- stop -step

In [40]:
```
a_arange = np.arange(10 ,50, 5)
```

In [41]:
```
a_arange
```

Out[41]:
```
array([10, 15, 20, 25, 30, 35, 40, 45])
```

linspace

By using linspace we also create a list from a minimum value to a maximum value. But instead of specifying the step-size, we specify the amount of values that we want to have in our list. They will all be spread evenly and have the same distance to their neighbors.

start-stop-number of values

In [42]:
```
b_linspace = np.linspace(0, 50 , 2)
b_linspace
```

Out[42]:
```
array([ 0., 50.])
```

In [43]:
```
b_linspace.ndim
```

Out[43]:
```
1
```

In [44]:
```
b_linspace.size
```

Out[44]:
```
2
```

In [45]:
```
b_linspace1 = np.linspace(0,50,30)
```

In [46]:
```
b_linspace1
```

Out[46]:
```
array([ 0.        ,  1.72413793,  3.44827586,  5.17241379,  6.89655172,
        8.62068966, 10.34482759, 12.06896552, 13.79310345, 15.51724138,
       17.24137931, 18.96551724, 20.68965517, 22.4137931 , 24.13793103,
       25.86206897, 27.5862069 , 29.31034483, 31.03448276, 32.75862069,
       34.48275862, 36.20689655, 37.93103448, 39.65517241, 41.37931034,
       43.10344828, 44.82758621, 46.55172414, 48.27586207, 50.        ])
```

In [47]:
```
b_linspace1.size
```

Out[47]:
```
30
```

In [48]:
```
b_linspace2 = np.linspace(0,50,25, dtype=int)
b_linspace2
```

Out[48]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 25, 27, 29, 31, 33,
       35, 37, 39, 41, 43, 45, 47, 50])
```

Math functions

np.exp(a)

Takes e to the power of each value

np.sin(a)

Returns the sine of each value

np.cos(a)

Returns the cosine of each value

np.tan(a)

Returns the tangent of each value

np.log(a)

Returns the logarithm of each value

np.sqrt(a)

Returns the square root of each value

AGGREGATE FUNCTIONS

a.sum()

Returns the sum of all values in the array

a.min()

Returns the lowest value of the array

a.max()

Returns the highest value of the array

a.mean()

Returns the arithmetic mean of all values in the array

np.median(a)

Returns the median value of the array

np.std(a)

Returns the standard deviation of the values in the array

Create some arrays and apply all the functions

Eg:

In [49]:
```
np.median(b_linspace2)
```

Out[49]:
```
25.0
```

In [50]:
```
np.sum(b_linspace2)
```

Out[50]:
```
614
```

In [51]:
```
np.sqrt(b_linspace2)
```

Out[51]:
```
array([0.        , 1.41421356, 2.        , 2.44948974, 2.82842712,
       3.16227766, 3.46410162, 3.74165739, 4.        , 4.24264069,
       4.47213595, 4.69041576, 5.        , 5.19615242, 5.38516481,
       5.56776436, 5.74456265, 5.91607978, 6.08276253, 6.244998  ,
       6.40312424, 6.55743852, 6.70820393, 6.8556546 , 7.07106781])
```

In [52]:
```
np.std(b_linspace2)
```

Out[52]:
```
14.924020905908701
```

In [53]:
```
np.cos(b_linspace2)
```

Out[53]:
```
array([ 1.        , -0.41614684, -0.65364362,  0.96017029, -0.14550003,
       -0.83907153,  0.84385396,  0.13673722, -0.95765948,  0.66031671,
        0.40808206, -0.99996083,  0.99120281, -0.29213881, -0.74805753,
        0.91474236, -0.01327675, -0.90369221,  0.76541405,  0.26664293,
       -0.98733928,  0.5551133 ,  0.52532199, -0.99233547,  0.96496603])
```