NumPy is a very popular python library for large multi-dimensional array and matrix processing, with the help of a large collection of high-level mathematical functions. It is very useful for fundamental scientific computations in Machine Learning. It is particularly useful for linear algebra, Fourier transform, and random number capabilities. High-end libraries like TensorFlow uses NumPy internally for manipulation of Tensors.

In [1]:
```
list = [1,2,34,55,66]
list
```

Out[1]:
```
[1, 2, 34, 55, 66]
```

In [2]:
```
arr = np.array(list)
arr
```

Out[2]:
```
array([ 1,  2, 34, 55, 66])
```

In [3]:
```
arr1 = np.array([[1,2,34,4],[4,6,7,9]])
arr1
```

Out[3]:
```
array([[ 1,  2, 34,  4],
       [ 4,  6,  7,  9]])
```

In [4]:
```
np.zeros((3,4))
```

Out[4]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [5]:
```
np.ones((3,4))
```

Out[5]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [6]:
```
arr2 = np.random.rand(3,4)
arr2
```

Out[6]:
```
array([[0.43480213, 0.9730572 , 0.68308684, 0.85331692],
       [0.12266533, 0.76188104, 0.02328788, 0.94703858],
       [0.55314846, 0.84510114, 0.95887045, 0.43242746]])
```

In [7]:
```
arr3 = np.random.rand(3,4)
arr3
```

Out[7]:
```
array([[0.41355436, 0.94744341, 0.41199124, 0.08539776],
       [0.2291305 , 0.77268602, 0.36014408, 0.42454119],
       [0.34870993, 0.25190213, 0.03502276, 0.58957633]])
```

In [8]:
```
np.arange(3,40,10)
```

Out[8]:
```
array([ 3, 13, 23, 33])
```

In [9]:
```
np.linspace(3,40,10)
```

Out[9]:
```
array([ 3.        ,  7.11111111, 11.22222222, 15.33333333, 19.44444444,
       23.55555556, 27.66666667, 31.77777778, 35.88888889, 40.        ])
```

In [10]:
```
np.full((3,4),5)
```

Out[10]:
```
array([[5, 5, 5, 5],
       [5, 5, 5, 5],
       [5, 5, 5, 5]])
```

In [11]:
```
np.eye(3,4)
```

Out[11]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

Arhtimetatic operations on arrays

In [12]:
```
a = np.array([1,2,3,4,5,6])
print(a+2)
print(a-1)
print(a*2)
print(a/2)
print(a%2)
print(a//2)
print(a**2)
print(a^2)
```

```
[3 4 5 6 7 8]
[0 1 2 3 4 5]
[ 2  4  6  8 10 12]
[0.5 1.  1.5 2.  2.5 3. ]
[1 0 1 0 1 0]
[0 1 1 2 2 3]
[ 1  4  9 16 25 36]
[3 0 1 6 7 4]
```

In [13]:
```
np.eye(3,k=1)
```

Out[13]:
```
array([[0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])
```

In [14]:
```
np.eye(3,k=-2)
```

Out[14]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [1., 0., 0.]])
```

The Shape and Reshaping of NumPy Arrays

Once you have created your ndarray, the next thing you would want to do is check the number of axes, shape, and the size of the ndarray.

In [15]:
```
a1 = np.array([[5,10,15],[20,25,20]])
a1.shape
```

Out[15]:
```
(2, 3)
```

In [16]:
```
a1.ndim
```

Out[16]:
```
2
```

In [17]:
```
a1.size
```

Out[17]:
```
6
```

In [18]:
```
a1.reshape(3,2)
```

Out[18]:
```
array([[ 5, 10],
       [15, 20],
       [25, 20]])
```

Flattening a NumPy array

Sometimes when you have a multidimensional array and want to collapse it to a single-dimensional array, you can either use the flatten() method or the ravel() method:

In [19]:
```
b = a1.flatten()
b
```

Out[19]:
```
array([ 5, 10, 15, 20, 25, 20])
```

In [20]:
```
c = a1.ravel()
c
```

Out[20]:
```
array([ 5, 10, 15, 20, 25, 20])
```

In [21]:
```
b[0] = 1000
print(b)
print(a1)
```

```
[1000   10   15   20   25   20]
[[ 5 10 15]
 [20 25 20]]
```

In [22]:
```
c[0] = 1000
print(c)
print(a1)
```

```
[1000   10   15   20   25   20]
[[1000   10   15]
 [  20   25   20]]
```

In [23]:
```
ar1 = np.array([[1,2,3,4,5],[6,7,8,9,10]])
ar2 = np.array([[11,12,13,14,15],[16,17,18,19,20]])
```

In [24]:
```
ar1 + ar2
```

Out[24]:
```
array([[12, 14, 16, 18, 20],
       [22, 24, 26, 28, 30]])
```

In [25]:
```
ar1 - ar2
```

Out[25]:
```
array([[-10, -10, -10, -10, -10],
       [-10, -10, -10, -10, -10]])
```

In [26]:
```
ar1 * ar2
```

Out[26]:
```
array([[ 11,  24,  39,  56,  75],
       [ 96, 119, 144, 171, 200]])
```

In [27]:
```
ar1/ar2
```

Out[27]:
```
array([[0.09090909, 0.16666667, 0.23076923, 0.28571429, 0.33333333],
       [0.375     , 0.41176471, 0.44444444, 0.47368421, 0.5       ]])
```

Transpose of a NumPy array

In [28]:
```
ar1 = np.transpose(ar1)
ar1
```

Out[28]:
```
array([[ 1,  6],
       [ 2,  7],
       [ 3,  8],
       [ 4,  9],
       [ 5, 10]])
```

dot product(multiplication of matrix)

In [29]:
```
dpro = ar1.dot(ar2)
dpro
```

Out[29]:
```
array([[107, 114, 121, 128, 135],
       [134, 143, 152, 161, 170],
       [161, 172, 183, 194, 205],
       [188, 201, 214, 227, 240],
       [215, 230, 245, 260, 275]])
```

In [30]:
```
ar2.dot(ar1)
```

Out[30]:
```
array([[205, 530],
       [280, 730]])
```

slicing

In [31]:
```
dpro[:,:]
```

Out[31]:
```
array([[107, 114, 121, 128, 135],
       [134, 143, 152, 161, 170],
       [161, 172, 183, 194, 205],
       [188, 201, 214, 227, 240],
       [215, 230, 245, 260, 275]])
```

In [32]:
```
dpro[:2,:]
```

Out[32]:
```
array([[107, 114, 121, 128, 135],
       [134, 143, 152, 161, 170]])
```

In [33]:
```
dpro[:3,:3]
```

Out[33]:
```
array([[107, 114, 121],
       [134, 143, 152],
       [161, 172, 183]])
```

In [34]:
```
d = dpro.flatten()
```

In [35]:
```
d
```

Out[35]:
```
array([107, 114, 121, 128, 135, 134, 143, 152, 161, 170, 161, 172, 183,
       194, 205, 188, 201, 214, 227, 240, 215, 230, 245, 260, 275])
```

In [36]:
```
d[-5:]
```

Out[36]:
```
array([215, 230, 245, 260, 275])
```

In [37]:
```
d[2:10:2]
```

Out[37]:
```
array([121, 135, 143, 161])
```

Sorting

In [38]:
```
dpro
```

Out[38]:
```
array([[107, 114, 121, 128, 135],
       [134, 143, 152, 161, 170],
       [161, 172, 183, 194, 205],
       [188, 201, 214, 227, 240],
       [215, 230, 245, 260, 275]])
```

In [39]:
```
np.sort(dpro,axis=0,kind='mergesort')
```

Out[39]:
```
array([[107, 114, 121, 128, 135],
       [134, 143, 152, 161, 170],
       [161, 172, 183, 194, 205],
       [188, 201, 214, 227, 240],
       [215, 230, 245, 260, 275]])
```

In [40]:
```
d1 = np.array([[1,23,4,20],[1,2,34,5],[45,3,19,2],[10,3,1,27]])
```

In [41]:
```
d1
```

Out[41]:
```
array([[ 1, 23,  4, 20],
       [ 1,  2, 34,  5],
       [45,  3, 19,  2],
       [10,  3,  1, 27]])
```

In [42]:
```
np.sort(d1)
```

Out[42]:
```
array([[ 1,  4, 20, 23],
       [ 1,  2,  5, 34],
       [ 2,  3, 19, 45],
       [ 1,  3, 10, 27]])
```

In [43]:
```
np.sort(d1,axis=0)
```

Out[43]:
```
array([[ 1,  2,  1,  2],
       [ 1,  3,  4,  5],
       [10,  3, 19, 20],
       [45, 23, 34, 27]])
```

In [44]:
```
np.sort(d1,axis=0,kind='mergesort')
```

Out[44]:
```
array([[ 1,  2,  1,  2],
       [ 1,  3,  4,  5],
       [10,  3, 19, 20],
       [45, 23, 34, 27]])
```

In [45]:
```
np.sort(d1,axis=1)
```

Out[45]:
```
array([[ 1,  4, 20, 23],
       [ 1,  2,  5, 34],
       [ 2,  3, 19, 45],
       [ 1,  3, 10, 27]])
```

In [46]:
```
np.sort(d1,axis=-1)
```

Out[46]:
```
array([[ 1,  4, 20, 23],
       [ 1,  2,  5, 34],
       [ 2,  3, 19, 45],
       [ 1,  3, 10, 27]])
```

Expanding and Squeezing a NumPy array

Expanding a NumPy array You can add a new axis to an array using the expand_dims() method by providing the array and the axis along which to expand:

In [52]:
```
a = np.array([1,2,3])
b = np.expand_dims(a,axis=0)
c = np.expand_dims(a,axis=1)
print('Original:','\n','Shape',a.shape,'\n',a)
print('Expand along columns:','\n','Shape',b.shape,'\n',b)
print('Expand along rows:','\n','Shape',c.shape,'\n',c)
```

```
Original:
 Shape (3,)
 [1 2 3]
Expand along columns:
 Shape (1, 3)
 [[1 2 3]]
Expand along rows:
 Shape (3, 1)
 [[1]
 [2]
 [3]]
```

Array merging

In [53]:
```
dpro
```

Out[53]:
```
array([[107, 114, 121, 128, 135],
       [134, 143, 152, 161, 170],
       [161, 172, 183, 194, 205],
       [188, 201, 214, 227, 240],
       [215, 230, 245, 260, 275]])
```

In [58]:
```
dpro1 = dpro[:4,:4]
dpro1
```

Out[58]:
```
array([[107, 114, 121, 128],
       [134, 143, 152, 161],
       [161, 172, 183, 194],
       [188, 201, 214, 227]])
```

In [59]:
```
d1
```

Out[59]:
```
array([[ 1, 23,  4, 20],
       [ 1,  2, 34,  5],
       [45,  3, 19,  2],
       [10,  3,  1, 27]])
```

In [60]:
```
np.vstack((dpro1,d1))
```

Out[60]:
```
array([[107, 114, 121, 128],
       [134, 143, 152, 161],
       [161, 172, 183, 194],
       [188, 201, 214, 227],
       [  1,  23,   4,  20],
       [  1,   2,  34,   5],
       [ 45,   3,  19,   2],
       [ 10,   3,   1,  27]])
```

In [63]:
```
np.concatenate((dpro1,d1),axis=0)
```

Out[63]:
```
array([[107, 114, 121, 128],
       [134, 143, 152, 161],
       [161, 172, 183, 194],
       [188, 201, 214, 227],
       [  1,  23,   4,  20],
       [  1,   2,  34,   5],
       [ 45,   3,  19,   2],
       [ 10,   3,   1,  27]])
```

In [61]:
```
np.hstack((dpro1,d1))
```

Out[61]:
```
array([[107, 114, 121, 128,   1,  23,   4,  20],
       [134, 143, 152, 161,   1,   2,  34,   5],
       [161, 172, 183, 194,  45,   3,  19,   2],
       [188, 201, 214, 227,  10,   3,   1,  27]])
```

In [64]:
```
np.concatenate((dpro1,d1),axis=1)
```

Out[64]:
```
array([[107, 114, 121, 128,   1,  23,   4,  20],
       [134, 143, 152, 161,   1,   2,  34,   5],
       [161, 172, 183, 194,  45,   3,  19,   2],
       [188, 201, 214, 227,  10,   3,   1,  27]])
```

In [66]:
```
dpro
```

Out[66]:
```
array([[107, 114, 121, 128, 135],
       [134, 143, 152, 161, 170],
       [161, 172, 183, 194, 205],
       [188, 201, 214, 227, 240],
       [215, 230, 245, 260, 275]])
```

In [69]:
```
np.hsplit(dpro1,2)
```

Out[69]:
```
[array([[107, 114],
        [134, 143],
        [161, 172],
        [188, 201]]),
 array([[121, 128],
        [152, 161],
        [183, 194],
        [214, 227]])]
```

In [70]:
```
np.vsplit(dpro1,2)
```

Out[70]:
```
[array([[107, 114, 121, 128],
        [134, 143, 152, 161]]),
 array([[161, 172, 183, 194],
        [188, 201, 214, 227]])]
```

Squeezing a NumPy array

On the other hand, if you instead want to reduce the axis of the array, use the squeeze() method. It removes the axis that has a single entry. This means if you have created a 2 x 2 x 1 matrix, squeeze() will remove the third dimension from the matrix:

In [71]:
```
#squeeze
a = np.array([[[1,2,3],
[4,5,6]]])
b = np.squeeze(a, axis=0)
print('Original','\n','Shape',a.shape,'\n',a)
print('Squeeze array:','\n','Shape',b.shape,'\n',b)
```

```
Original
 Shape (1, 2, 3)
 [[[1 2 3]
  [4 5 6]]]
Squeeze array:
 Shape (2, 3)
 [[1 2 3]
 [4 5 6]]
```

Broadcasting in NumPy arrays – A class apart!

Broadcasting is one of the best features of ndarrays. It lets you perform arithmetics operations between ndarrays of different sizes or between an ndarray and a simple number!

Broadcasting essentially stretches the smaller ndarray so that it matches the shape of the larger ndarray:

In [72]:
```
a = np.arange(10,20,2)
b = np.array([[2],[2]])
print('Adding two different size arrays :','\n',a+b)
print('Multiplying an ndarray and a number :',a*2)
```

```
Adding two different size arrays :
 [[12 14 16 18 20]
 [12 14 16 18 20]]
Multiplying an ndarray and a number : [20 24 28 32 36]
```

NumPy Ufuncs – The secret of its success!

Python is a dynamically typed language. This means the data type of a variable does not need to be known at the time of the assignment. Python will automatically determine it at run-time. While this means a cleaner and easier code to write, it also makes Python

sluggish.

This problem manifests itself when Python has to do many operations repeatedly, like the addition of two arrays. This is so because each time an operation needs to be performed, Python has to check the data type of the element. This problem is overcome by NumPy using the ufuncs function.

The way NumPy makes this work faster is by using vectorization. Vectorization performs the same operation on ndarray in an element-by-element fashion in a compiled code. So the data types of the elements do not need to be determined every time, thereby performing faster operations.

ufuncs are Universal functions in NumPy that are simply mathematical functions. They perform fast element-wise functions. They are called automatically when you are performing simple arithmetic operations on NumPy arrays because they act as wrappers for NumPy ufuncs.

For example, when adding two NumPy arrays using '+', the NumPy ufunc add() is automatically called behind the scene and quietly does its magic:

In [73]:
```
a = [1,2,3,4,5]
b = [6,7,8,9,10]
%timeit a+b
```

```
169 ns ± 26.7 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops ea
ch)
```

In [75]:
```
d1
```

Out[75]:
```
array([[ 1, 23,  4, 20],
       [ 1,  2, 34,  5],
       [45,  3, 19,  2],
       [10,  3,  1, 27]])
```

In [76]:
```
np.max(d1)
```

Out[76]:
```
45
```

In [77]:
```
np.max(d1,axis=1)
```

Out[77]:
```
array([23, 34, 45, 27])
```

In [78]:
```
np.max(d1,axis=0)
```

Out[78]:
```
array([45, 23, 34, 27])
```

In [79]:
```
np.min(d1,axis=1)
```

Out[79]:
```
array([1, 1, 2, 1])
```

In [80]:
```
np.min(d1,axis=0)
```

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

In [81]:
```
np.min(d1)
```

Out[81]:
```
1
```

NumPy arrays and Images

NumPy arrays find wide use in storing and manipulating image data. But what is image data really?

Images are made up of pixels that are stored in the form of an array. Each pixel has a value ranging between 0 to 255 – 0 indicating a black pixel and 255 indicating a white pixel. A colored image consists of three 2-D arrays, one for each of the color channels: Red, Green, and Blue, placed back-to-back thus making a 3-D array. Each value in the array constitutes a pixel value. So, the size of the array depends on the number of pixels along each dimension.

Have a look at the image below:

In [85]:
```
ax = d1.ravel()
ax
```

Out[85]:
```
array([ 1, 23,  4, 20,  1,  2, 34,  5, 45,  3, 19,  2, 10,  3,  1, 27])
```

In [87]:
```
np.sort(ax)
```

Out[87]:
```
array([ 1,  1,  1,  2,  2,  3,  3,  4,  5, 10, 19, 20, 23, 27, 34, 45])
```

In [88]:
```
ax >5
```

Out[88]:
```
array([False, False, False, False, False, False, False, False, False,
        True,  True,  True,  True,  True,  True,  True])
```

In [89]:
```
ax[ax>5]
```

Out[89]:
```
array([10, 19, 20, 23, 27, 34, 45])
```

In [90]:
```
len(ax[ax>5])
```

Out[90]:
```
7
```

In [91]:
```
print(ax[ax%3==0])
print(len(ax[ax%3==0]))
```

```
[ 3  3 27 45]
4
```