Introduction: ***NumPy*** is an important package of Python that is not just used extensively in Data Science but also in other domains like astronomy, engineering, finance, stock market, online retail, insurance, multimedia, sports, biology and healthcare etc. In all of these domains, there is a requirement to ***store and process high dimensional arrays efficiently***.

We have already reviewed lists, tuples, sets and dictionaries which apparently can also meet the requirement to store and process arrays. Then why do we need ***NumPy***?

In the real life situations our performance requirements are very high and the data types mentioned above are very inefficient to meet that. Imagine that you are modelling a Deep Learning model and its takes 100 days to get trained with Python lists. Will you not want to train the model in just 1 day?

Lists are designed to store heterogeneous data in a single array so there is an overhead of type checking. They do not exploit the available hardware mechanisms to accelerate the performance (e.g. DRAM access and vectorised processing)

NumPy on the other hand is designed to bring performance and functionality improvement for numerical computing. It is becoming a de-facto package for numerical computing and base for other Python packages.

NumPy efficiently store n-dimensional ***(nd)*** arrays leveraging the DRAM burst access technology, avoids type checking overheads and efficiently broadcasts the operations across different dimensions of an array (e.g. perform an operation to all the elements just by specifying once). It provides implementation of many functions across linear algebra, statistics etc. and not just that; these functions provide a very high performance in terms of processing time.

Performance Efficiency in NumPy

Let us say we have a very large array of size 10^8 elements and in each location the index of the location itself is stored. We want to perform an operation to store the square of the number present in that location. This is first implemented using lists and its performance is measured.

```
%%time
N = 100000000
myList = list (range(N))
for i in range(N):
    myList[i] = myList[i]*myList[i]
```

```
Wall time: 51.1 s
```

It takes around 51 seconds. Alternatively, the same functionality can be achieved using ***map ()*** function with ***lambda*** operator. Please note that lambda operator help to write a throw-away, in-line and anonymous function that just serves the purpose at that place where it is required.

```
%%time
myList = list (range(N))
myList = map (lambda x: x*x, myList)
```

```
Wall time: 7.38 s
```

We can observe a significant improvement and it takes just 7.38 seconds. Now the same functionality is performed using ***NumPy***. The numpy package is imported as an alias ***np*** and its functions are accessed using this alias. This import needs to be done only once in the beginning of the code where NumPy package functions are going to be utilized.

```
import numpy as np
```

```
%%time
myList = np.arange(N)
myList = myList * myList
```

```
Wall time: 1.26 s
```

We can further observe a significant improvement and it takes just 1.26 seconds.

The NumPy function ***arange ()*** is used above which returns evenly spaced values within a given interval of type ***numpy.ndarray.*** Also observe that how the square is done for the entire n-dimensional array without using the indices.

```
print (type(myList))
```

```
<class 'numpy.ndarray'>
```

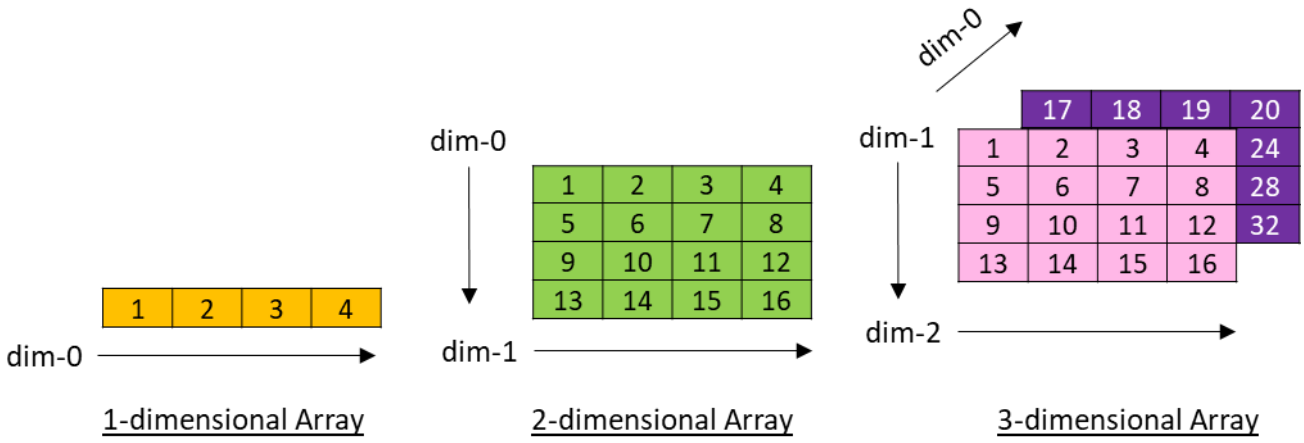
sum () is another useful function that adds the elements of an array.

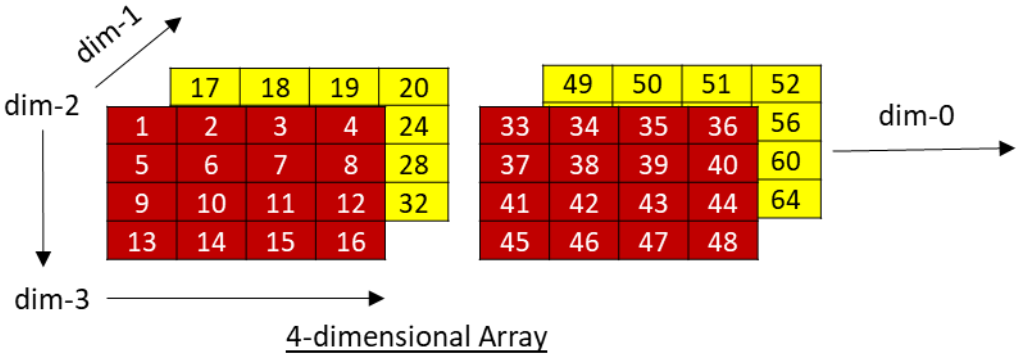
```
n = 10
myList = np.arange(n)
sum = np.sum(myList)
print (sum)
```

```
45
```

n-dimensional Arrays

Before going to the details of ***NumPy***, it is important to understand and visualize n-dimensional array and also how its elements are accessed. Few figures below explain the concept. We are already familiar with one and two dimensional arrays. Let us try to visualize higher dimensional arrays. Only four dimensional arrays are shown below but the same concept can be extended further for higher dimensions.





From the figure above, observe that the counting of the dimension starts from higher dimension to lower dimension. For example:

- In the 4-dimensional array: the 4th dimension is numbered as dim-0, the 3rd dimension is numbered as dim-1 and so on.
- In the 3-dimensional array: the 3rd dimension is numbered as dim-0, the 2nd dimension is numbered as dim-1 and so on.

This mechanism helps to index the elements. For example, if all of these n-dimensional arrays are A1, A2, A3 and A4 respectively, the elements can be accessed as following:

- Element in A1 as A1[dim-0]
- Element in A2 as A2[dim-0, dim-1]
- Element in A3 as A3[dim-0, dim-1, dim-2]
- Element in A4 as A3[dim-0, dim-1, dim-2, dim-3]

Examples:

- The element locations of value 3 in different arrays are: A1 [2], A2 [0, 2], A3 [0, 0, 2], and A4 [0, 0, 0, 2]
- The element location of value 46 is A4 [1, 0, 3, 1].
- For A3, if write A3 [:, 0, 1:3], it means all elements of dim-0, only the 0^{th} element of dim-1 and from 1 to 2 (excluding 3) elements of dim-2. That means the values 2, 3, 18 and 19 are selected.

Exercise:

i. Find out the element locations of element values: 11 in A3, 24 in A3, 11 in A4 and 55 in A4.
ii. Which elements will be selected with A3 [0, 1:3, 2:4]?

Creating n-dimensional Arrays in NumPy

- In addition to the ***arange ()*** function, the NumPy arrays can be created using ***array ()*** function also, which takes a list as an argument. The details of ***arange ()*** will be reviewed in the subsequent worksheets.

```
aList = [0, 1, 2, 3, 4, 5, 6]
arr1d = np.array(aList)
print(arr1d, type (arr1d))
```

```
[0 1 2 3 4 5 6] <class 'numpy.ndarray'>
```

- The following code shows the different attributes associated with the 1-dimesnional array that is created above:

```
arr1d.dtype
```

```
dtype('int32')
```

← *Size of each element. Here is it 32 bits that is 4 bytes.*

```
arr1d.ndim
```

```
1
```

← *This is only 1-dimensional array.*

```
arr1d.shape
```

```
(7,)
```

← *One dimension of 7 elements.*

```
arr1d.size
```

```
7
```

← *There are total 7 elements.*

```
arr1d.itemsize
```

```
4
```

← *Each item is of 4 bytes*

- Similarly a 2-dimensional array can be created with a list and its attributes can be verified.

```
arr2d = np.array([
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
    [13, 14, 15, 16]
])
print("dtype = ", arr2d.dtype)
print("ndim = ", arr2d.ndim)
print("shape = ", arr2d.shape)
print("size = ", arr2d.size)
print("itemsize = ", arr2d.itemsize)
```

```
dtype =  int32
ndim =  2
shape =  (4, 4)
size =  16
itemsize =  4
```

- Similarly a 3-dimensional array can be created with a list and its attributes can be verified.

```
arr3d = np.array([
    [
        [1, 2, 3, 4],
        [5, 6, 7, 8],
        [9, 10, 11, 12],
        [13, 14, 15, 16]
    ],
    [
        [17, 18, 19, 20],
        [21, 22, 23, 24],
        [25, 26, 27, 28],
        [29, 30, 31, 32]
    ]
])
print("dtype = ", arr3d.dtype)
print("ndim = ", arr3d.ndim)
print("shape = ", arr3d.shape)
print("size = ", arr3d.size)
print("itemsize = ", arr3d.itemsize)
```

```
dtype =  int32
ndim =  3
shape =  (2, 4, 4)
size =  32
itemsize =  4
```

Before reading further, ensure that you have practiced the code yourself that is shown so far in this worksheet!

- If the numeric data type any array element is changed to float, the entire array is modified and for each element the datatype is changed. For example, in the code below, the array is created as [0, 1.1, 2, 3, 4, 5, 6]. Now the ***dtype*** for each element is changed to float64.

```
aList = [0, 1.1, 2, 3, 4, 5, 6]
arr1d = np.array(aList)
print(arr1d, type (arr1d))
```

```
[0.  1.1 2.  3.  4.  5.  6. ] <class 'numpy.ndarray'>
```

```
arr1d.dtype
```

```
dtype('float64')
```

- Arrays of all 1's and all 0's can be created using ***ones ()*** and ***zeros ()*** functions.

```
np.zeros ((2, 2))
```

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

```
np.ones ((2, 2, 2))
```

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

       [[1., 1.],
        [1., 1.]]])
```