Week - 4: NumPy and Pandas - I

Introduction to NumPy:-

In this module, you will learn about the basics of NumPy, which is the fundamental package for scientific computing in Python. NumPy consists of a robust data structure called multidimensional arrays. Pandas is another powerful Python library that provides a fast and easy-to-use data analysis platform.

You will understand the advantages of using NumPy. You will also learn how to:

- Create NumPy arrays,
- Convert lists and tuples to NumPy arrays,
- Inspect the structure and content of arrays, and
- Subset, slice, index and iterate through arrays.

Before we get into the technicalities of a NumPy array, explore its useful functions and understand how it is implemented in Python, it is crucial to understand **why NumPy is an important library for working with data.**

NumPy, an acronym for the term 'Numerical Python', is a library in Python which is used extensively for efficient mathematical computing. This library allows users to store large amounts of data using less memory and perform extensive operations efficiently. It provides optimised and simpler functionalities to perform aforementioned operations using homogenous, one-dimensional and multidimensional arrays (You will learn more about this later.).

Now, before delving deep into the concept of NumPy arrays, it is important to note that Python lists can very well perform all the actions that NumPy arrays perform; it is simply the fact that NumPy arrays are faster and more convenient than lists when it comes to extensive computations, which make them extremely useful, especially when you are working with large amounts of data.

Guidelines for coding console questions

The lectures are interspersed with coding consoles to help you practise writing Python code. You will be given a brief problem statement and some pre-written code. You can write the code in the provided space, verify your answer using test cases and submit it when you are confident about it.

Note that the coding console questions are non-graded. Some instructions for these questions are as follows:

- Ignore the pre-written code on the console. Please do not change it.
- Write your answer only in the space where you are asked to write.
- You may run and verify your codes any number of times.

Basics of NumPy:-

NumPy, which stands for 'Numerical Python', is a library meant for scientific calculations. The basic data structure of NumPy is an array. A NumPy array is a collection of values stored together, similar to a list.

This video mentioned two different advantages that NumPy arrays have over lists. These include:

1. **Ability to operate on individual elements in the array without using loops or list comprehension**
2. Speed of execution

The demonstration in the video above did not cover the aspect of speed, so for now, you can assume that a NumPy array is faster than a list. Later in this session, you will be able to take a look at a detailed demonstration to compare the speed of NumPy arrays.

you have learnt how a NumPy array behaves differently with the '+' operator. You also learnt that a NumPy array can be created using a pre-existing list. There will be more details on the use of operators with arrays in the latter part of the session. For now, let's discuss how to create arrays.

Creating NumPy Arrays

There are ***two ways to create NumPy arrays***, which are mentioned below.

1. **By converting the existing lists or tuples to arrays using np.array**
2. **By initialising fixed-length arrays using the NumPy functions**

In this session, you will learn about both these methods.

The **key advantage of using NumPy arrays over lists is that arrays allow you to operate over the entire data, unlike lists.** However, in terms of structure, NumPy arrays are extremely similar to lists. If you try to run the print() command over a NumPy array, then you will get the following output:

[element_1 element_2 element_3...]

The only difference between a NumPy array and a list is that the elements in the NumPy array are separated by a space instead of a comma. Hence, this is an aesthetic feature that differentiates a list and a NumPy array.

```
In [8]: print(type(arr))
        print(arr) #arr is the numpy 1-D array

        <class 'numpy.ndarray'>
        [74 75 72 72 71]

In [9]: print(type(heights))
        print(heights) #heights is the simple list

        <class 'list'>
        [74, 75, 72, 72, 71]
```

An important point to note here is that the array given above is a one-dimensional array. You will learn about multidimensional arrays in the subsequent segments.

Another feature of NumPy arrays is that they are homogeneous in nature. By homogenous, we mean that all the elements in a NumPy array have to be of the same data type, which could be an integer, float, string, etc.

Operations Over 1-D Arrays:-

In the previous segment, you learnt how to create NumPy arrays using existing lists. Once you have loaded the data into an array, NumPy offers a wide range of operations to perform on the data.

you learnt about the calculation of BMI using NumPy arrays. In the BMI example given above, if you had been working with lists, then you would have needed to map a lambda function (or worse, write a for loop). Whereas, with NumPy, you simply use the relevant operators, as NumPy does all the back-end coding on its own.

Now that you have learnt how to use operators to perform basic operations on a 1D array, let's understand how to access the elements of an array. For one-dimensional arrays, indexing, slicing, etc. are similar to those in Python lists, which means that indexing starts at 0. The following video will demonstrate the methodologies of indexing and slicing arrays.

As explained in this video, indexing refers to extracting a single element from an array, while slicing refers to extracting a subset of elements from an array. Both indexing and slicing are exactly the same to those in lists. Having a unified method of extracting elements from lists and NumPy arrays helps in keeping the library simpler.

The aforementioned element extraction methods will only help you when you know the location of the element that you want to extract. In the following video, you will learn how to access elements based on a condition.

To summarise, similar to lists, you can subset your data through conditions based on your requirements in NumPy arrays. To do this, you need to use logical operators such as '<' and '>'. NumPy also has a few inbuilt functions such as max(), min() and mean(), which allow you to calculate statistically important data over the data directly. In the following video, Behzad will explore these functions and also summarise the learnings of this segment.

Heights of the players is stored as a regular Python list: height_in. The height is expressed in inches. Can you make a numpy array out of it ?

```
In [12]: # Define list
         heights = [74, 74, 72, 72, 73, 69, 69, 71, 76, 71, 73, 73, 74, 74, 69, 70, 73, 75, 78, 79, 76, 74, 76, 72, 71, 75, 77,

In [14]: print(type(heights))
         <class 'list'>

In [15]: heights_in = np.array(heights)

In [16]: print(heights_in)
         [74 74 72 ... 75 75 73]

In [17]: type(heights_in)
Out[17]: numpy.ndarray

In [18]: heights_in.shape
Out[18]: (1015,)
```

Convert the heights from inches to meters

```
In [22]: heights_m = heights_in * 0.0254
         print(heights_m)
         [1.8796 1.8796 1.8288 ... 1.905  1.905  1.8542]

In [ ]:
```

Example - 4

A list of weights (in lbs) of the players is provided. Convert it to kg and calculate BMI

```
In [13]: weights_lb = [180, 215, 210, 210, 188, 176, 209, 200, 231, 180, 188, 180, 185, 160, 180, 185, 189, 185, 219, 230, 205,

In [23]: weights = np.array(weights_lb)

In [24]: print(weights)
         [180 215 210 ... 205 190 195]

In [25]: print(weights.shape)
         (1015,)

In [26]: weights_kg = weights * 0.453592

In [27]: print(weights_kg)
         [81.64656 97.52228 95.25432 ... 92.98636 86.18248 88.45044]

In [29]: print(heights_m.shape)
         print(weights_kg.shape)
```

Conditional Sub-Setting Arrays

Count the number of pariticipants who are underweight i.e. bmi < 21

```
In [38]: """Conditional Sub-Setting Arrays : is the way with the help of which you can find the sub-set of the original numpy
         array even without looping through and applying if condition or by applying filter method. It is a hybrid between
         applying indexing on numpy array and applying mathematical operation on numpy array."""

Out[38]: 'Conditional Sub-Setting Arrays : is the way with the help of which you can find the sub-set of the original numpy \n
         array even without looping through and applying if condition or by applying filter method. It is a hybrid between \na
         pplying indexing on numpy array and applying mathematical operation on numpy array.'

In [42]: bmi < 21

Out[42]: array([False, False, False, ..., False, False, False])

In [39]: underweight_can = bmi[bmi<21]

In [41]: print(bmi.shape)
         print(underweight_can.shape)
         print(underweight_can)

         (1015,)
         (11,)
         [20.54255679 20.54255679 20.69282047 20.69282047 20.34343189 20.34343189
          20.69282047 20.15883472 19.4984471  20.69282047 20.9205219 ]
```

Multidimensional Arrays:-

Until now, you learnt about one-dimensional arrays, where all the data is stored in a single line or row. In this segment, you will learn what a multidimensional array is.

A **multidimensional array** is an array of arrays. For example, a two-dimensional array would be an array with each element as a one-dimensional array.

1-D array : [1, 2, 3, 4, 5]
2-D array : [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]

Similarly, a three-dimensional array can be thought of as an array with each element as a two-dimensional array. To create multidimensional arrays, you can give a multidimensional list as an input to the np.array function.

-.-Extra thoughts starts-.-.-.-.-.-.-.-.-.-..-

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example

Create a 0-D array with value 42
import numpy as np

arr = np.array(42)

print(arr)

Try it Yourself »

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
These are the most common and basic arrays.

Example

Create a 1-D array containing the values 1,2,3,4,5:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

Try it Yourself »

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.
These are often used to represent matrix or 2nd order tensors.
NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

Example

Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr)
```

Try it Yourself »

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.
These are often used to represent a 3rd order tensor.

Example

Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(arr)
```

Try it Yourself »

Check Number of Dimensions?

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

Example

Check how many dimensions the arrays have:

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

Try it Yourself »

Higher Dimensional Arrays

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using the ndmin argument.

Example

Create an array with 5 dimensions and verify that it has 5 dimensions:

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('number of dimensions :', arr.ndim)
```

Try it Yourself »

In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

-.-Extra thoughts Ends-.-.-.-.-.-.-.-.-..-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-

NumPy arrays have certain features that help in analysing multidimensional arrays. Some features are as follows:

1. shape: It represents the shape of an array as the number of elements in each dimension.
2. ndim: It represents the number of dimensions of an array. For a 2D array, ndim = 2.

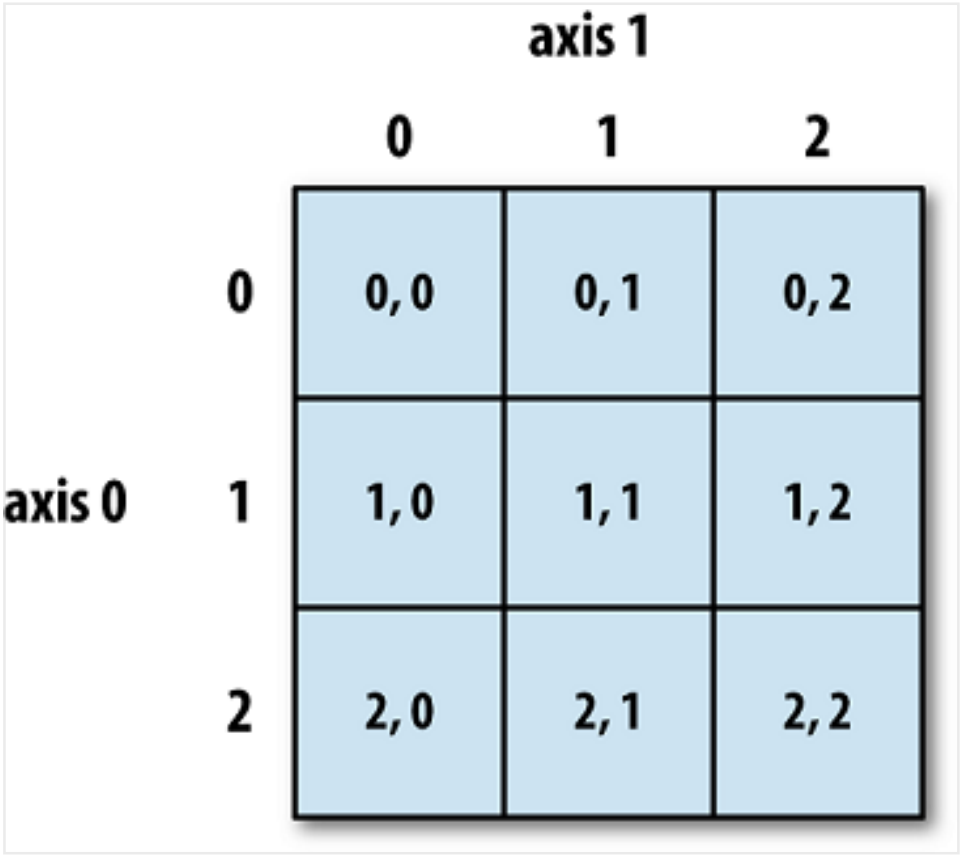
Similar to 1D arrays, nD arrays can also operate on individual

elements without using list comprehensions or loops

you can multiply the different elements in an array with different values. This property of NumPy is called broadcasting. As this is a slightly advanced topic with respect to the scope of this module, it will not be explained in details here. However, If you wish to learn more about broadcasting, then you can visit the following link.

In NumPy, the dimension is called **axis**. In NumPy terminology, for 2-D arrays:

- *axis = 0* - refers to the rows
- *axis = 1* - refers to the columns



Multidimensional Array

Multidimensional arrays are indexed using as many indices as the number of dimensions or axes. For instance, to index a 2-D array, you need two indices: array[x, y]. Each axis has an index starting at 0. The figure provided above shows the axes and their indices for a 2-D array.

The indexing and slicing of nD arrays are similar to those of 1D arrays. The only difference between the two is that you need to give the slicing and indexing instructions for each dimension separately. See example below:

players_converted[:, **0**]

Returns all the rows in the 0th column. Here, ':' is the instruction for all the **rows**, and 0 is the instruction for the 0th **columns**. Similarly, for a 3D array, the slicing command will have three arguments. You can also opt for conditional slicing and indexing of data in nD arrays.

you can apply a condition on the array itself or use a different array with the same dimensions to apply the condition.

you will learn how to initialise fixed-length one-dimensional NumPy arrays using different functions.

Additional Resouces:

Here is a visual representation video on **making arrays and understanding dimensions visually**

Creating NumPy Arrays:-

In the previous segments, you learnt how to convert lists or tuples to arrays using np.array(). There are other ways in which you can create arrays. The following ways are commonly used when you know the size of the array beforehand:-

1. **np.ones()**: It is used to create an array of 1s.
2. **np.zeros()**: It is used to create an array of 0s.
3. **np.random.randint()**: It is used to create a random array of integers within a particular range.
4. **np.random.random()**: It is used to create an array of random numbers.
5. **np.arange()**: It is used to create an array with increments of fixed step size.
6. **np.linspace()**: It is used to create an array of fixed length.

Tip: Use help to see the syntax when required

In [2]:
```
help(np.ones)
```

```
Help on function ones in module numpy:

ones(shape, dtype=None, order='C')
    Return a new array of given shape and type, filled with ones.

    Parameters
    ----------
    shape : int or sequence of ints
        Shape of the new array, e.g., ``(2, 3)`` or ``2``.
    dtype : data-type, optional
        The desired data-type for the array, e.g., `numpy.int8`.  Default is
        `numpy.float64`.
    order : {'C', 'F'}, optional, default: C
        Whether to store multi-dimensional data in row-major
        (C-style) or column-major (Fortran-style) order in
        memory.

    Returns
    -------
    out : ndarray
        Array of ones with the given shape, dtype, and order.

    See Also
    --------
    ones_like : Return an array of ones with shape and type of input.
    empty : Return a new uninitialized array.
    zeros : Return a new array setting values to zero.
    full : Return a new array of given shape filled with value.
```

Creating a 1 D array of ones

In [3]:
```
arr = np.ones(5)
arr
```

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

Notice that, by default, numpy creates data type = float64

In [4]:
```
arr.dtype
```

Out[4]:
```
dtype('float64')
```

Can provide dtype explicitly using dtype

In [5]:
```
arr = np.ones(5, dtype=int)
arr
```

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

In [6]:
```
arr.dtype
```

Out[6]:
```
dtype('int64')
```

Creating a 5 x 3 array of ones

In [7]:
```
np.ones((5,3))
```

Out[7]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Creating array of zeros

In [8]:
```
np.zeros(5)
```

Out[8]:
```
array([0., 0., 0., 0., 0.])
```

In [9]:
```
# convert the type into integer.
np.zeros(5, dtype=int)
```

Out[9]:
```
array([0, 0, 0, 0, 0])
```

In [12]:
```
# Create a list of integers range between 1 to 5.
list(range(1,5))
```

Out[12]:
```
[1, 2, 3, 4]
```

In [13]:
```
np.arange(3)
```

Out[13]:
```
array([0, 1, 2])
```

In [14]:
```
np.arange(3.0)
```

Out[14]:
```
array([0., 1., 2.])
```

Notice that 3 is included, 35 is not, as in standard python lists

From 3 to 35 with a step of 2

In [20]:
```
np.arange(3,35,2)
```

Out[20]:
```
array([ 3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33])
```

Array of random numbers

```
In [21]: np.random.randint(2, size=10)
Out[21]: array([0, 1, 0, 1, 1, 1, 0, 0, 0, 0])

In [24]: np.random.randint(3,5, size=10)
Out[24]: array([3, 3, 3, 3, 4, 3, 3, 3, 3, 4])
```

2D Array of random numbers

```
In [25]: np.random.random([3,4])
Out[25]: array([[0.37947795, 0.50446351, 0.76204337, 0.23268129],
                [0.49530063, 0.37298231, 0.17830691, 0.9400508 ],
                [0.18746889, 0.99395211, 0.03729134, 0.16021317]])
```

Sometimes, you know the length of the array, not the step size

Array of length 20 between 1 and 10

```
In [27]: np.linspace(1,10,20)
Out[27]: array([ 1.        ,  1.47368421,  1.94736842,  2.42105263,  2.89473684,
                 3.36842105,  3.84210526,  4.31578947,  4.78947368,  5.26315789,
                 5.73684211,  6.21052632,  6.68421053,  7.15789474,  7.63157895,
                 8.10526316,  8.57894737,  9.05263158,  9.52631579, 10.        ])
```

Many other functions can also be used to create arrays. A few methods that will not be covered in this segment are mentioned below. Please read the official NumPy documentation to understand the usage of these methods.

1. np.full(): It is used to create a constant array of any number 'n'.
2. np.tile(): It is used to create a new array by repeating an existing array for a particular number of times.
3. np.eye(): It is used to create an identity matrix of any dimension

In the next segment, you will learn about the mathematical methods available for performing mathematical transformations on NumPy arrays

Mathematical Operations on NumPy I:-

The objective of this segment is to discuss the mathematical capabilities of NumPy as a library that is meant for scientific calculations.

1. Manipulate arrays
    1. Reshape arrays

    2. Stack arrays
2. Perform operations on arrays
    1. Perform basic mathematical operations
        1. Power
        2. Absolute
        3. Trigonometric
        4. Exponential and logarithmic
3. Apply built-in functions
4. Apply your own functions
5. Apply basic linear algebra operations

In the first half of this segment, you will learn about concepts such as reshaping and stacking arrays. These concepts are important and might come in handy. In the latter half of this segment, demonstrations will show all the mathematical topics mentioned above. The notebook below will be useful in this and the next segment.

A point to note here is that when the shapes of NumPy arrays are not the same, the arrays cannot be operated on. Because of this restriction on the operability, the flexibility of NumPy reduces significantly. Fortunately, the developers of NumPy noticed this issue and developed a few functions that can modify the shape of a NumPy array. They will be covered one by one in the following video.

The commands demonstrated in this video can be used to change the dimensions of a given NumPy array. These commands are as follows:

1. **hstack:** It puts two arrays with the same number of rows together. By using this command, the number of rows stays the same, while the number of columns increases.
2. **vstack:** It puts two arrays on top of each other. This command works only when the number of columns in both the arrays is the same. This command can only change the number rows of an array.
3. **reshape:** It can change the shape of an array as long as the number of elements in the array before and after the reshape operation is the same.

you learnt about a few commands such as power and absolute that are available in the NumPy library. In the next segment, let's continue exploring the mathematical capabilities of the NumPy library.

Additional Resources:

Here is a video tutorial on **Reshaping & Indexing NumPy Arrays**

Mathematical Operations on NumPy II:-

The objective of this segment is to cover the mathematical capabilities of NumPy. Note that these mathematical functions might not be of direct use for you. In actual practice as a data scientist, you might not use these functions, but the advanced functions that you will use will be built using these functions. So, it would help if you remember that the NumPy library has all of these capabilities.

Let's watch the following video and understand the **trigonometric** capabilities of NumPy.

The next set of mathematical capabilities
is **exponential** and **logarithmic** functions.

You learnt about the mathematical functions that can be directly calculated. Another important feature offered by NumPy is **empty arrays**, where you can initialise an empty array and later use it to store the output of your operations.

Once you have created an array, you may also want to run aggregation operations on the data stored in it. An aggregation function helps you summarise the numerical data.

Example - 8 (Aggregates)

```
In [67]: x = np.arange(1,6)
         x
Out[67]: array([1, 2, 3, 4, 5])

In [69]: sum(x)
Out[69]: 15

In [68]: np.add.reduce(x)
Out[68]: 15

In [70]: np.add.accumulate(x)
Out[70]: array([ 1,  3,  6, 10, 15])

In [72]: np.multiply.accumulate(x)
Out[72]: array([  1,   2,   6,  24, 120])

In [ ]:
```

Using the reduce() and accumulate() functions, you can easily summarise the data available in arrays. The reduce() function results in a single value, whereas the accumulate() function helps you apply your aggregation sequentially on

each element of an array. These functions require a base function to aggregate the data, for example, add() in the case given above.

The last mathematical capability that we will discuss is the linear algebra module in the NumPy library. Linear algebra is a significantly used module in machine learning. You might not be expected to write code using the NumPy library, but the functions being used will depend on the functions demonstrated below.

you learnt about numerous functions, such as rank and the inverse of a matrix. Although NumPy can calculate the results of these functions, the operations need to be valid. For example, if it is not possible for a matrix to be inverted, then the NumPy inverse operations will also throw an error.

You can also find the exponents of a matrix from the linalg module of NumPy. Behzad will demonstrate the same in the video here.

With this, we have come to the end of the demonstration of mathematical capabilities of the NumPy library.
In the next segment, we will have a detailed demonstration of computational speeds of NumPy arrays versus the lists.

Computation Times in NumPy vs Python Lists:-

You will often work with extremely large datasets; thus, it is important for you to understand how much computation time (and memory) you can save using NumPy as compared with the use of standard Python lists.
To compare both these data structures, it is recommended that you code along with us and experiment with the different kinds of data to see the difference in real time.

Now, let's compare the computation times of arrays and lists through a simple task of calculating the element-wise product of numbers.

Compare Computation Times in NumPy and Standard Python Lists

Now that we know how to use numpy, let us see code and witness the key advantages of numpy i.e. convenience and speed of computation.

In the data science landscape, you'll often work with extremely large datasets, and thus it is important point for you to understand how much computation time (and memory) you can save using numpy, compared to standard python lists.

Let's compare the computation times of arrays and lists for a simple task of calculating the element-wise product of numbers.

In [3]:
```
import time
import numpy as np
```

In [4]:
```
## Comparing time taken for computation
list_1 = [i for i in range(1000000)]
list_2 = [j**2 for j in range(1000000)]

t0 = time.time()
product_list = list(map(lambda x, y: x*y, list_1, list_2))
t1 = time.time()
list_time = t1 - t0
print (t1-t0)

# numpy array
array_1 = np.array(list_1)
array_2 = np.array(list_2)

t0 = time.time()
product_numpy = array_1 * array_2
t1 = time.time()
numpy_time = t1 - t0
print (t1-t0)

print("The ratio of time taken is {}".format(list_time//numpy_time))
```

```
0.06237626075744629
0.0011398792266845703
The ratio of time taken is 54.0
```

There is a huge difference in the time taken to perform the same operation using lists vs the Numpy arrays. Let's try to find the ratio of the speed of NumPy array as compared to lists.

NumPy is an order of magnitude faster than lists. This is with arrays of sizes in millions, but you may work on much larger arrays with sizes in billions. Then, the difference may be even larger.
Some reasons for such difference in speed are as follows:

- NumPy is written in C, which is basically being executed behind the scenes.
- NumPy arrays are more compact than lists, i.e., they take less storage space than lists.

The following discussions demonstrate the differences in the speeds of NumPy and standard Python lists.

- Why are NumPy arrays so fast?
- Why NumPy instead of Python lists?

Summary:-

In this session, you learnt about the most important package for scientific computing in Python: NumPy. The various operations that you learnt about include:

- Arrays, which are the basic data structure in the NumPy library
- Creating NumPy arrays from a list or a tuple
- Creating randomly large arrays which can be done using the arange command
- Analysing the shape and dimension of an array

using array.shape, array.ndim and so on
- Indexing, slicing and subsetting an array, which is very similar to indexing in lists
- Working on multidimensional arrays
- Manipulating arrays using reshape(), hstack() and vstack()

Additional Reading:

If you want to learn more about this topic that what is covered in this module, you can optionally use the additional resources provided below.

- NumPy in detail