NUMPY

NumPy (Numerical Python) is a fundamental library in the Python ecosystem for numerical and scientific computing. It provides support for working with large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. Here are some key points and elaborated notes on why NumPy is used:

Why NumPy is Used:

- **Efficiency**: NumPy is designed to be highly efficient, making it significantly faster than standard Python lists when working with large datasets. This efficiency is crucial in scientific and numerical computing.
- **Multi-dimensional Arrays**: NumPy introduces the ndarray data structure, which allows you to work with multi-dimensional arrays effortlessly. These arrays are essential for tasks like image processing, linear algebra, and scientific simulations.
- **Element-wise Operations**: NumPy supports element-wise operations, which means you can perform mathematical operations on entire arrays without the need for explicit loops. This simplifies code and enhances performance.
- **Broadcasting**: NumPy enables broadcasting, which is a powerful feature for performing operations on arrays of different shapes. It automatically aligns and extends smaller arrays to match the shape of the larger one, making code concise and readable.
- **Mathematical Functions**: NumPy provides a wide range of mathematical functions for operations like trigonometry, linear algebra, statistics, and more. These functions are optimized for numerical accuracy and performance.
- **Random Number Generation**: It includes functions for generating random numbers, which are essential for simulations and statistical analysis.
- **Integration with Other Libraries**: NumPy seamlessly integrates with other popular libraries in the scientific Python ecosystem, such as SciPy, Matplotlib, and Pandas, allowing you to perform a wide range of tasks.

As we all know that python had only 4 Data Structures TUPLE, LIST, DICTIONARY and SET, but we didn’t hear anything about ARRAY’s.

WHAT IS AN ARRAY IN PYTHON

- An array is a data structure in Python used to store a collection of items, such as numbers, strings, or other data types.
- Arrays allow you to organize and manipulate a group of related data elements under a single name.
- Think of an array as a container that can hold multiple values, and each value is stored at a specific position, known as an index.

NOTE:

Before the widespread use of NumPy in Python, people primarily used Python lists to create and work with arrays or collections of data. While Python lists are versatile and can hold elements of various data types, they lack the performance and functionality optimizations provided by NumPy arrays

So If we have LIST before used to work with array's then why do we need NUMPY Library ?

ASPECT	PYTHON LIST	NUMPY ARRAY
DATA TYPE	HETEROGENEOUS (CAN STORE ELEMENTS OF DIFFERENT DATA TYPES).	HOMOGENEOUS (TYPICALLY STORES ELEMENTS OF THE SAME DATA TYPE).
EFFICIENCY	SLOWER FOR NUMERICAL OPERATIONS ON LARGE DATASETS.	HIGHLY OPTIMIZED FOR NUMERICAL AND SCIENTIFIC COMPUTING, FASTER PERFORMANCE.
SIZE FLEXIBILITY	DYNAMIC SIZING; EASILY RESIZED USING METHODS LIKE APPEND.	FIXED SIZE ONCE CREATED, REQUIRING A NEW ARRAY FOR SIZE CHANGES.
MATHEMATICAL FUNCTIONS	LIMITED BUILT-IN MATHEMATICAL FUNCTIONALITY.	EXTENSIVE MATHEMATICAL FUNCTIONS AND OPERATIONS AVAILABLE.
BROADCASTING	LACKS BROADCASTING CAPABILITIES.	SUPPORTS BROADCASTING, SIMPLIFYING OPERATIONS ON ARRAYS OF DIFFERENT SHAPES.
MEMORY EFFICIENCY	LESS MEMORY-EFFICIENT DUE TO DATA TYPE FLEXIBILITY.	MORE MEMORY-EFFICIENT, BETTER CONTROL OVER DATA TYPES, LESS MEMORY CONSUMPTION.
USE CASES	SUITABLE FOR GENERAL-PURPOSE DATA STORAGE AND MANIPULATION.	PREFERRED FOR NUMERICAL AND SCIENTIFIC COMPUTING, DATA ANALYSIS, MACHINE LEARNING, AND SIMULATIONS.

NUMPY BASICS - A STUDENT'S GUIDE

Introduction: NumPy, short for Numerical Python, is a powerful library in Python that's widely used for numerical and scientific computing. It provides a versatile array data structure and various mathematical functions. In this manual, you'll learn the fundamentals of using NumPy.

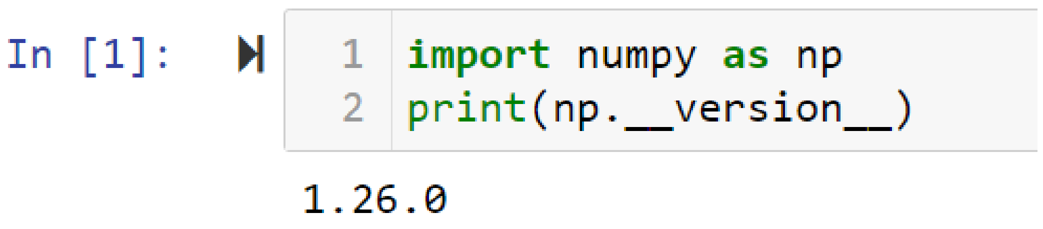
Getting Started:

Installation: This is required if we create a new environment that doesn't have numpy installed, But if we are using anaconda by default environment then it these libraries were pre installed.

Ensure NumPy is installed on your system. If not, install it using a package manager like pip:

```
pip install numpy
```

As we are Using Jupyter Notebook via Anaconda so in the base environment some of the libraries like Numpy ,Pandas, matplotlb etc were preinstalled.

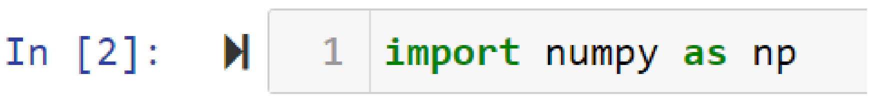


```
In [1]:
import numpy as np
print(np.__version__)
```

```
1.26.0
```

Importing Numpy:

Importing NumPy in Python is straightforward. You can import NumPy using the import statement, and it's common to use the alias np for NumPy to make the code more concise. Here's how to import NumPy in Python:



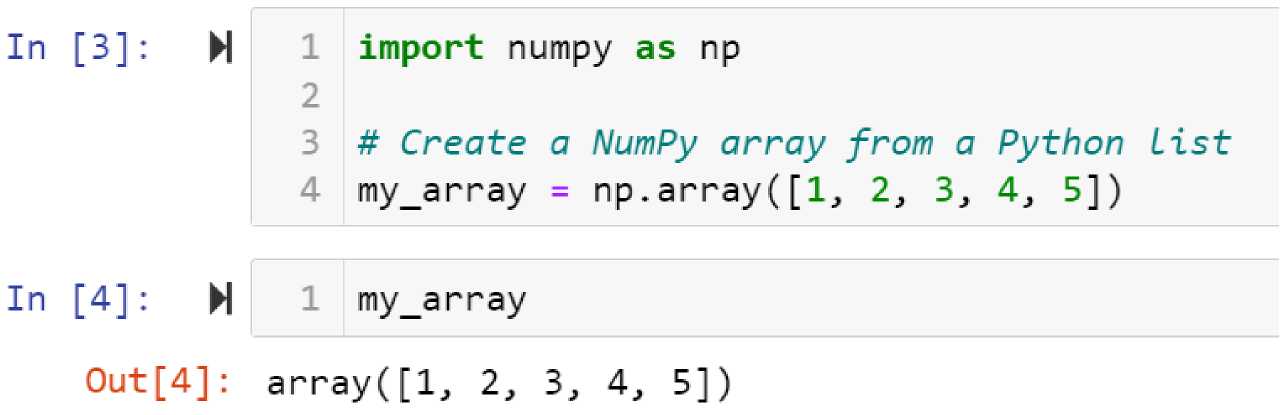
```
In [2]:
import numpy as np
```

NumPy Arrays

In NumPy, the fundamental data structure is the ndarray, short for "N-dimensional array." It is a versatile data structure that allows you to store and manipulate large, homogeneous data sets efficiently. The ndarray is at the core of many numerical computations in Python.

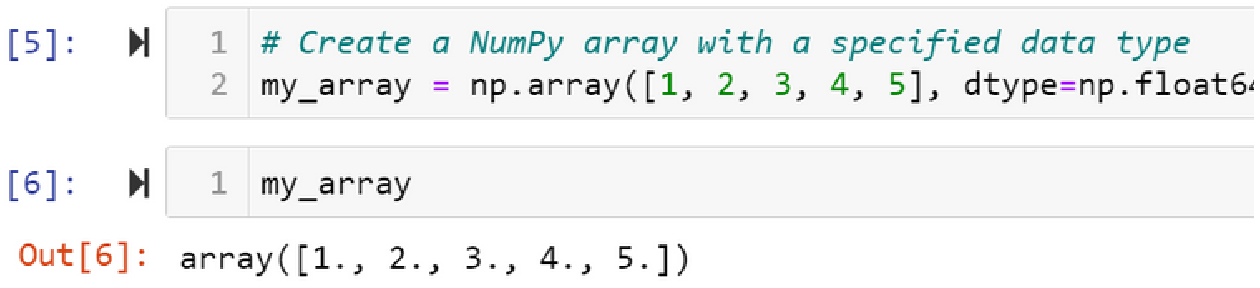
Creating NumPy Arrays:

To create a NumPy array, you can use the np.array() function by passing a Python list or another iterable as an argument. NumPy will automatically infer the data type.



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

# Create a NumPy array from a Python list
my_array = np.array([1, 2, 3, 4, 5])

In [4]:
my_array

Out[4]: array([1, 2, 3, 4, 5])
```

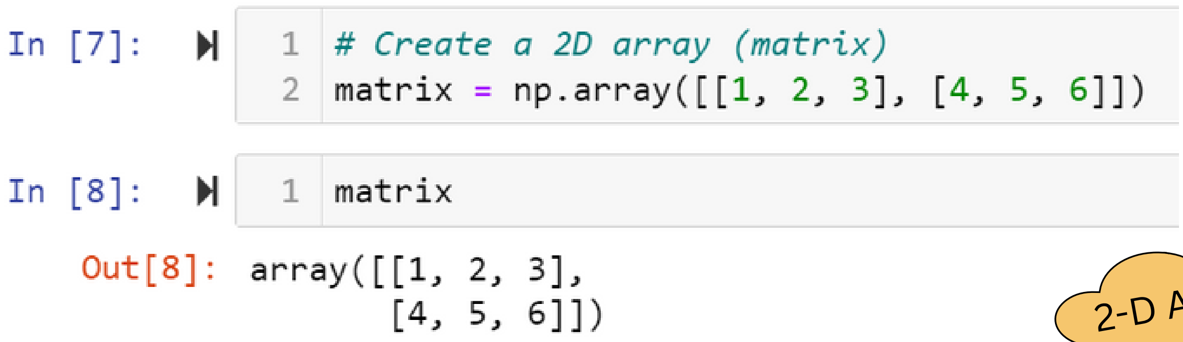
We can also explicitly specify the data type using the dtype argument.



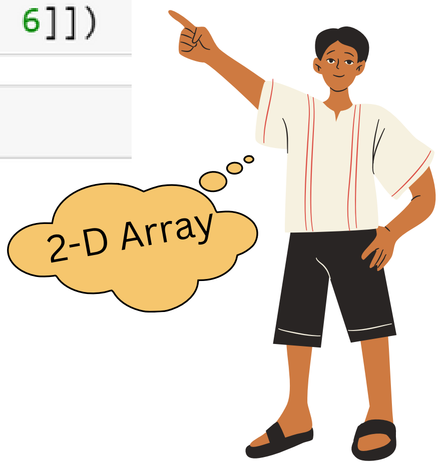
```
[5]:
# Create a NumPy array with a specified data type
my_array = np.array([1, 2, 3, 4, 5], dtype=np.float6

[6]:
my_array

Out[6]: array([1., 2., 3., 4., 5.])
```

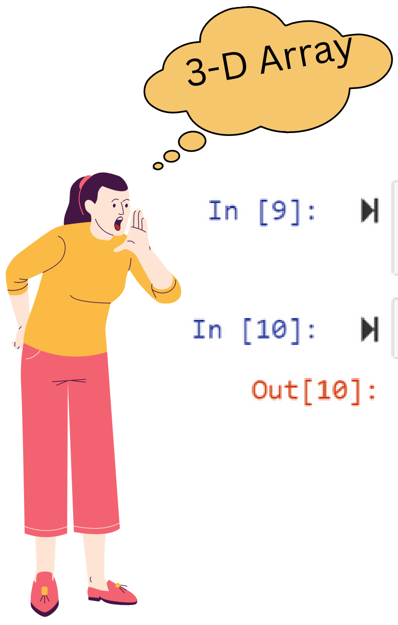
Array Dimensions:

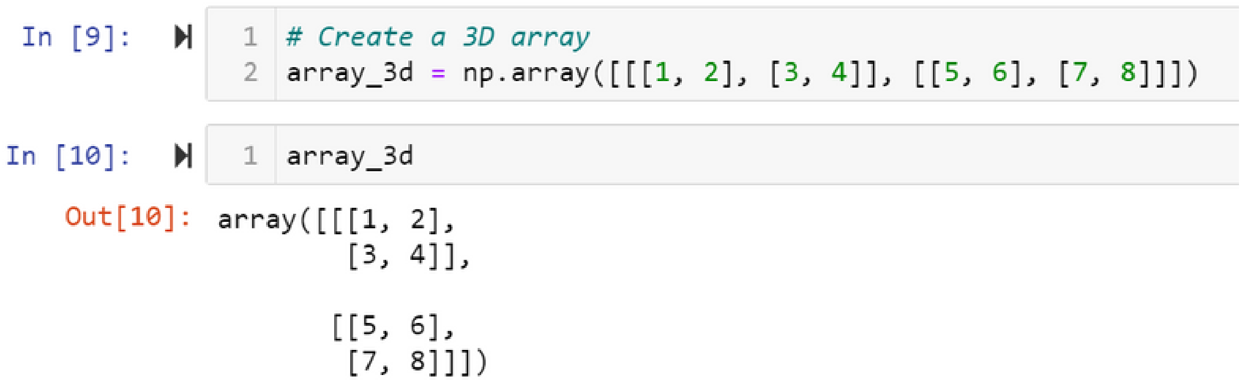
NumPy arrays can have multiple dimensions. You can create arrays with different shapes (1D, 2D, 3D, etc.) by passing nested lists to np.array().



```
In [7]:   # Create a 2D array (matrix)
          matrix = np.array([[1, 2, 3], [4, 5, 6]])

In [8]:   matrix

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





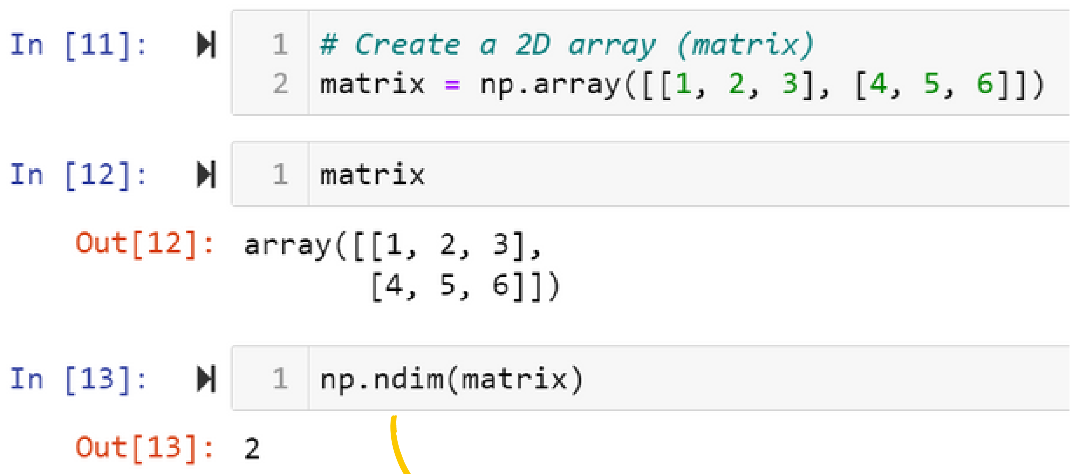


```
In [9]:   # Create a 3D array
          array_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

In [10]:  array_3d

Out[10]: array([[[1, 2],
                 [3, 4]],

                [[5, 6],
                 [7, 8]]])
```

As we've created a lot of Matrix in schools we use to provide the digits under box bracket or square brackets and arrays are almost the same if we increase the box brackets while creating an array it will automatically increases the dimension of an array from 2d to 3d and more.

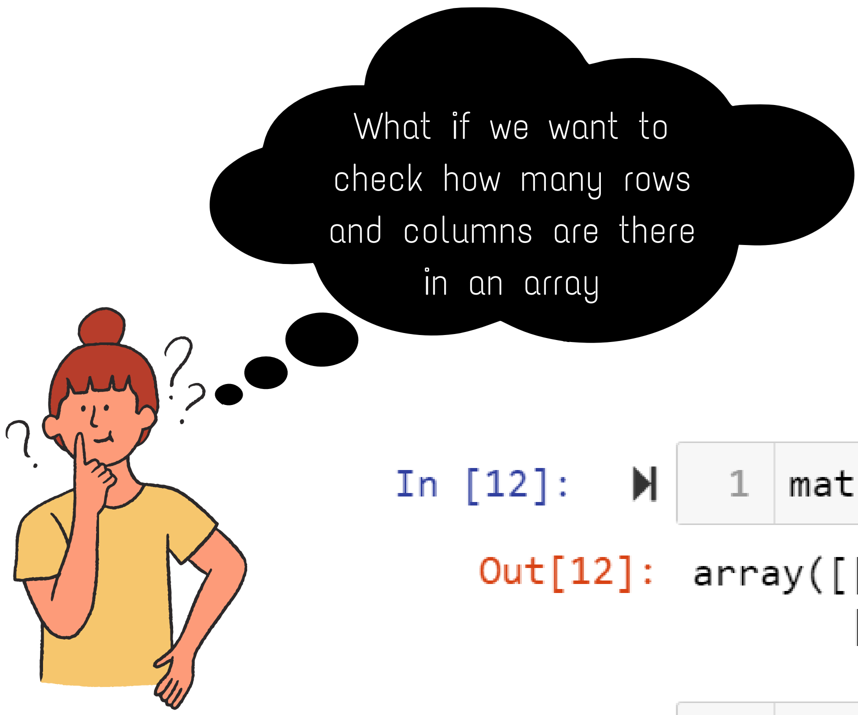
Dimensions and Shape of an Array

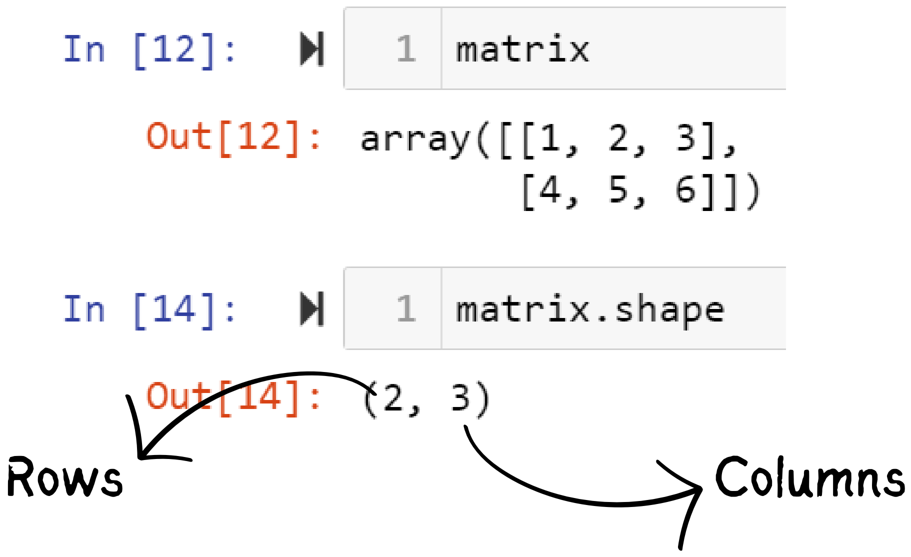
As we have created an array above now we want to check that what is the dimension of that array and how many rows and columns are there in an created array.



```
In [11]:  # Create a 2D array (matrix)
          matrix = np.array([[1, 2, 3], [4, 5, 6]])

In [12]:  matrix

Out[12]:  array([[1, 2, 3],
                 [4, 5, 6]])

In [13]:  np.ndim(matrix)

Out[13]:  2
```

np.ndim method has been used to check the dimensions of an array


What if we want to check how many rows and columns are there in an array



```
In [12]:  matrix

Out[12]:  array([[1, 2, 3],
                 [4, 5, 6]])

In [14]:  matrix.shape

Out[14]:  (2, 3)
```

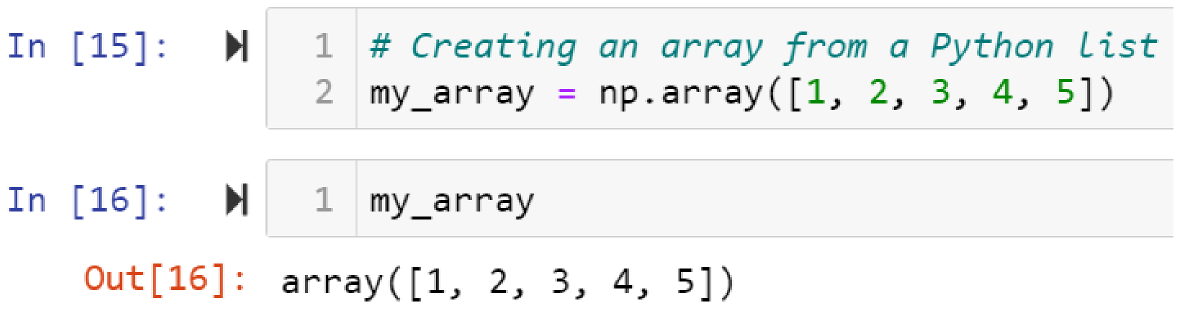
Array Creation:

NumPy provides various functions for creating arrays with different shapes and initial values. Here, We'll explain different ways to create arrays using some common functions.

As we've created array's before manually now let's try how we can create array with some synthetic data via NUMPY library.

np.array()

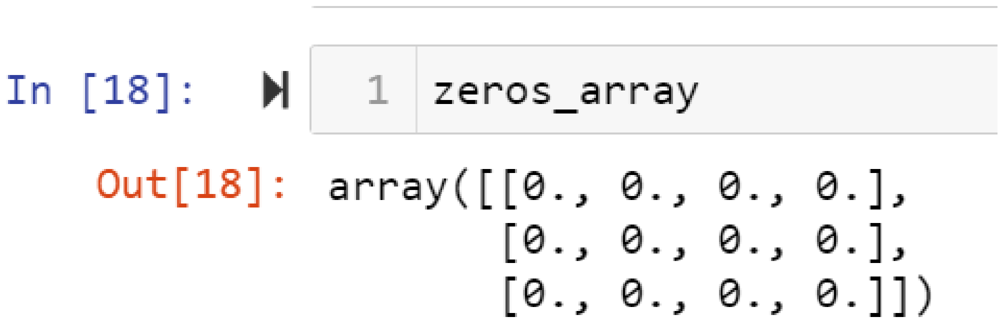
We can create a NumPy array from a Python list or iterable using np.array().



```
In [15]:  # Creating an array from a Python list
          my_array = np.array([1, 2, 3, 4, 5])

In [16]:  my_array

Out[16]:  array([1, 2, 3, 4, 5])
```

np.zeros()

Creates an array filled with zeros. We can specify the shape by passing a tuple with the desired dimensions.



```
In [18]:  zeros_array

Out[18]:  array([[0., 0., 0., 0.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.]])
```

np.ones()

Creates an array filled with ones, similar to np.zeros(). You specify the shape with a tuple.

```
In [19]:  # Create a 2x3 array of ones
          ones_array = np.ones((2, 3))

In [20]:  ones_array

Out[20]:  array([[1., 1., 1.],
                 [1., 1., 1.]])
```

np.full()

Creates an array filled with used custom value, similar to zeros and ones. We specify the shape with a tuple and outside the shape tuple we need to mention the value that we want an array of.

```
In [21]:  # Create a 3X2 array of user custom Value
          custom_array = np.full((3,2),10)

In [22]:  custom_array

Out[22]:  array([[10, 10],
                 [10, 10],
                 [10, 10]])
```

These are the 4 most used and simple methods to create arrays.

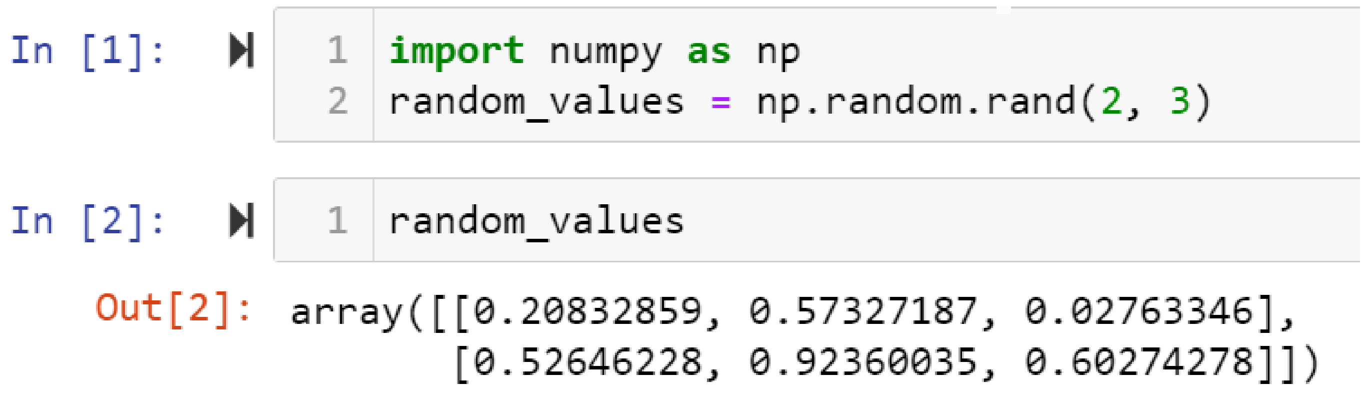
Above methods are used to create arrays but there is a limit attached while we create these array's that in zeros there should be only zero in ones there should be only ones and in full there should be a custom number. So we use Random methods to create array's that has random data in them.

Array Creation with Numpy Random Method:

It includes a submodule called numpy.random that provides various methods for generating random numbers and random arrays. These random numbers are generated using various probability distributions, and they are often used in simulations, statistics, and machine learning.

np.random.rand

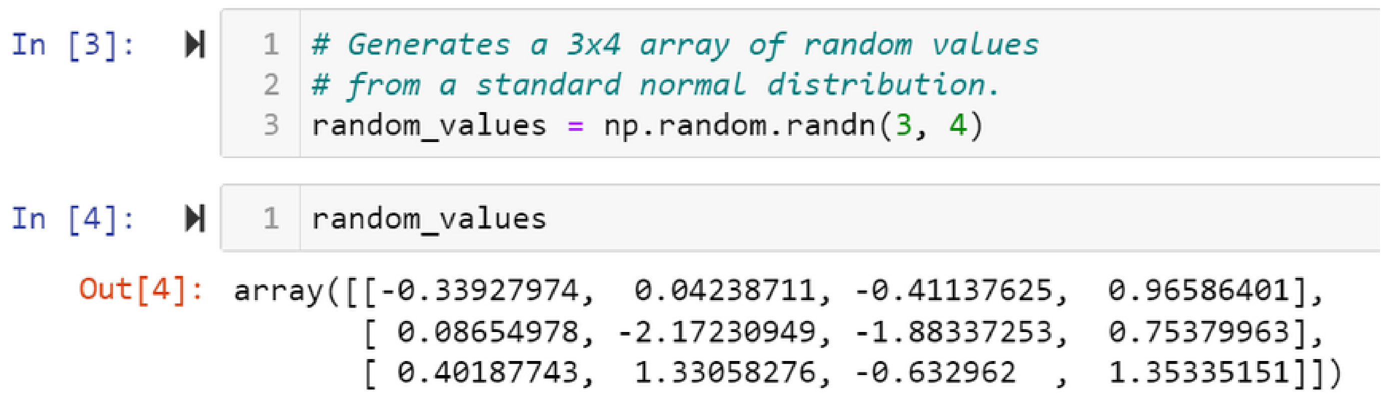
np.random.rand: This function generates random values from a uniform distribution over the range [0, 1]. You can pass the shape of the output array as arguments.



```
In [1]:  import numpy as np
         random_values = np.random.rand(2, 3)

In [2]:  random_values

Out[2]: array([[0.20832859, 0.57327187, 0.02763346],
               [0.52646228, 0.92360035, 0.60274278]])
```

np.random.randn

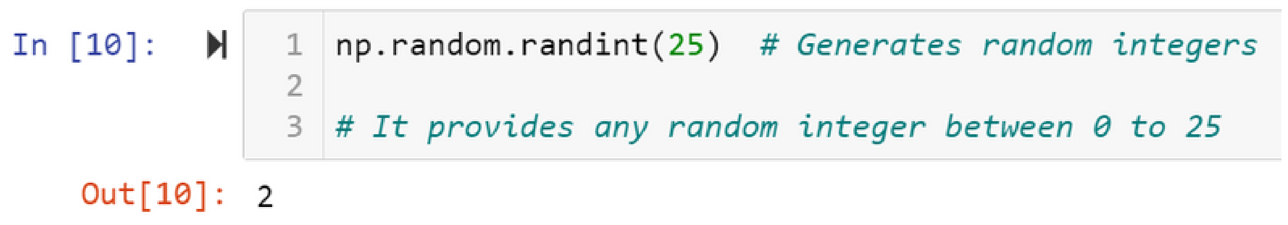
np.random.randn: This function generates random values from a standard normal distribution (mean 0, standard deviation 1). You can pass the shape of the output array as arguments.



```
In [3]:  # Generates a 3x4 array of random values
         # from a standard normal distribution.
         random_values = np.random.randn(3, 4)

In [4]:  random_values

Out[4]: array([[-0.33927974,  0.04238711, -0.41137625,  0.96586401],
               [ 0.08654978, -2.17230949, -1.88337253,  0.75379963],
               [ 0.40187743,  1.33058276, -0.632962  ,  1.35335151]])
```

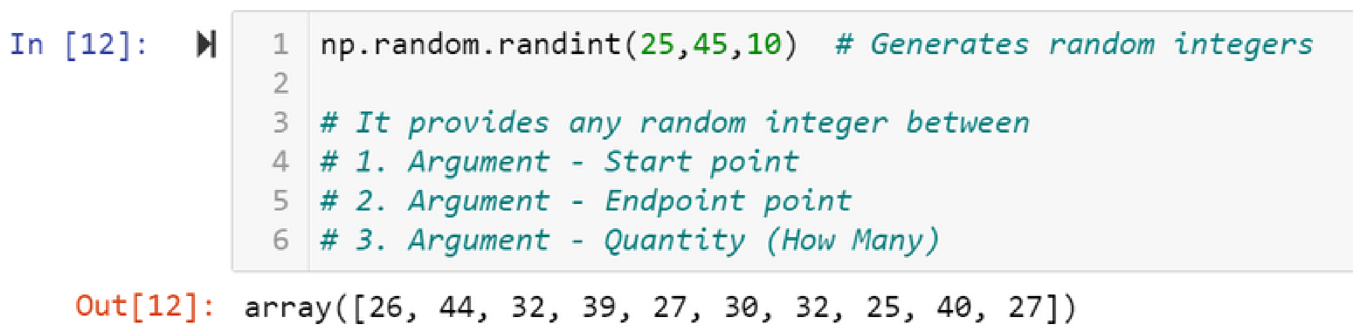
np.random.randint

np.random.randint: This function generates random integers within a specified range. You can specify the lower and upper bounds and the shape of the output array.



```
In [10]:  np.random.randint(25)   # Generates random integers

          # It provides any random integer between 0 to 25

Out[10]: 2
```

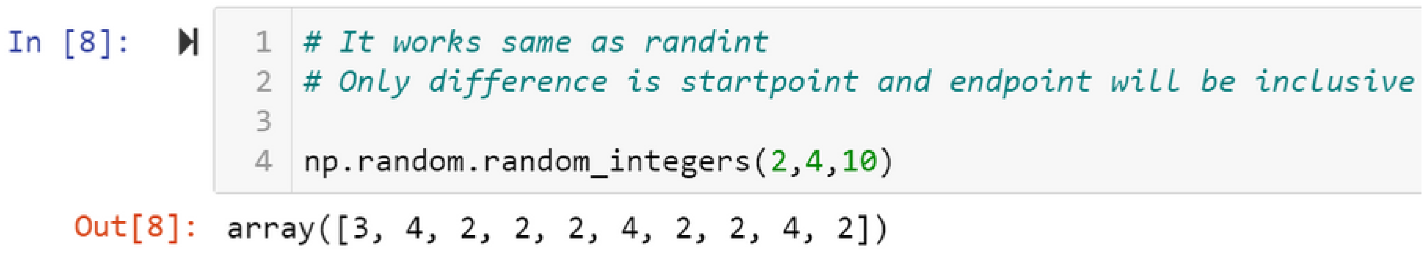
randint take exactly 3 arguments (Start Point, End Point, Increment)



```
In [12]:  np.random.randint(25,45,10)   # Generates random integers

          # It provides any random integer between
          # 1. Argument - Start point
          # 2. Argument - Endpoint point
          # 3. Argument - Quantity (How Many)

Out[12]: array([26, 44, 32, 39, 27, 30, 32, 25, 40, 27])
```

np.random.random_itegers

The numpy.random.random_integers function was deprecated in favor of numpy.random.randint to make the behavior of generating random integers more consistent with common practice and to align with the behavior of other random number generation functions in NumPy.



```
In [8]:  # It works same as randint
         # Only difference is startpoint and endpoint will be inclusive

         np.random.random_integers(2,4,10)
Out[8]:  array([3, 4, 2, 2, 2, 4, 2, 2, 4, 2])
```

numpy.random.randint

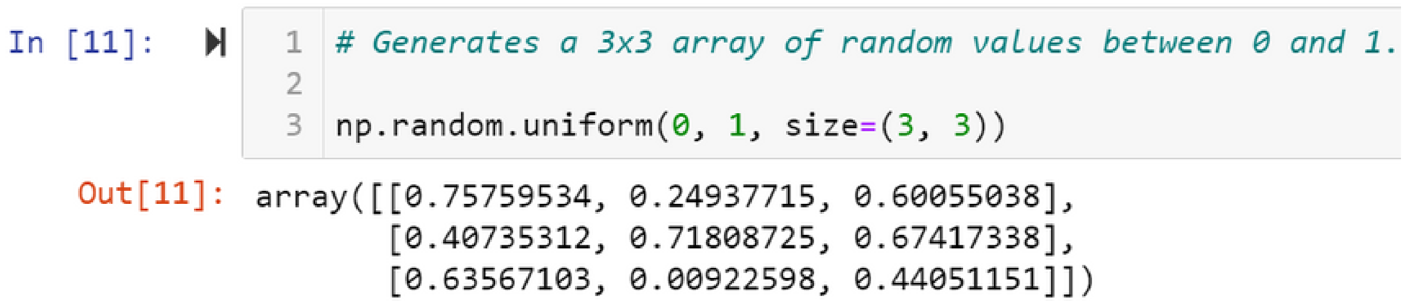
- Generates random integers within a specified range [low, high).
- The low argument is inclusive, meaning it is possible to generate values equal to low.
- The high argument is exclusive, meaning it is not possible to generate values equal to high.
- You can specify the shape of the output array.
- This function is widely used for generating random integers.

numpy.random.random_integers

- Also generated random integers within a specified range [low, high].
- Both the low and high arguments were inclusive, meaning values at both extremes of the range were possible.
- You could specify the shape of the output array, similar to numpy.random.randint.

np.random.uniform

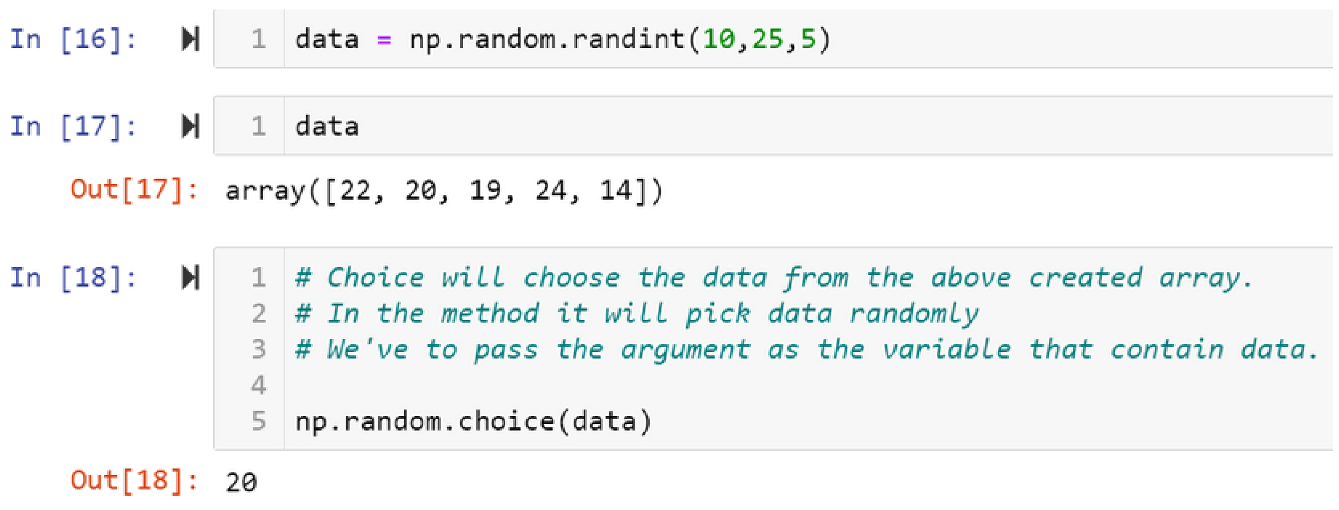
numpy.random.uniform: This function generates random numbers from a uniform distribution within a specified range. You can set the lower and upper bounds, and the shape of the output array.



```
In [11]:  # Generates a 3x3 array of random values between 0 and 1.

          np.random.uniform(0, 1, size=(3, 3))
Out[11]:  array([[0.75759534, 0.24937715, 0.60055038],
                 [0.40735312, 0.71808725, 0.67417338],
                 [0.63567103, 0.00922598, 0.44051151]])
```

np.random.choice

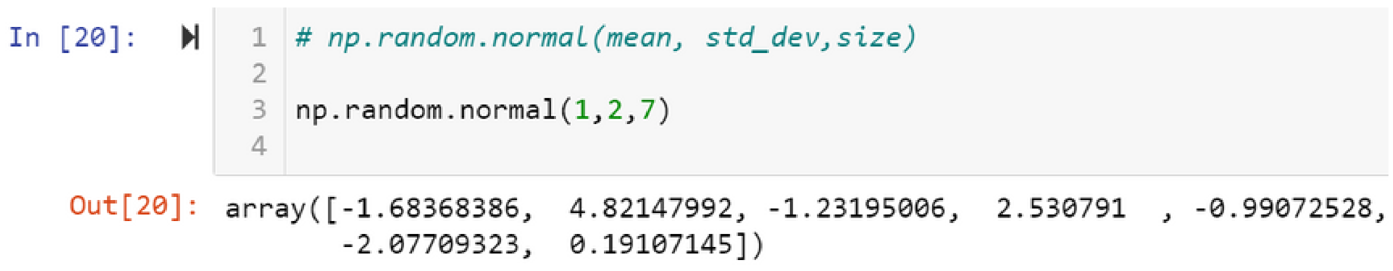
In Numpy choice method works when we have already created an array and choice will pick any data from the created array randomly.



```
In [16]:  data = np.random.randint(10,25,5)

In [17]:  data

Out[17]: array([22, 20, 19, 24, 14])

In [18]:  # Choice will choose the data from the above created array.
          # In the method it will pick data randomly
          # We've to pass the argument as the variable that contain data.

          np.random.choice(data)

Out[18]: 20
```

np.random.choice

In NumPy, you can use the numpy.random.normal method to generate random numbers from a normal (Gaussian) distribution. The normal distribution is characterized by its mean (μ) and standard deviation (σ). Here's how to use numpy.random.normal:

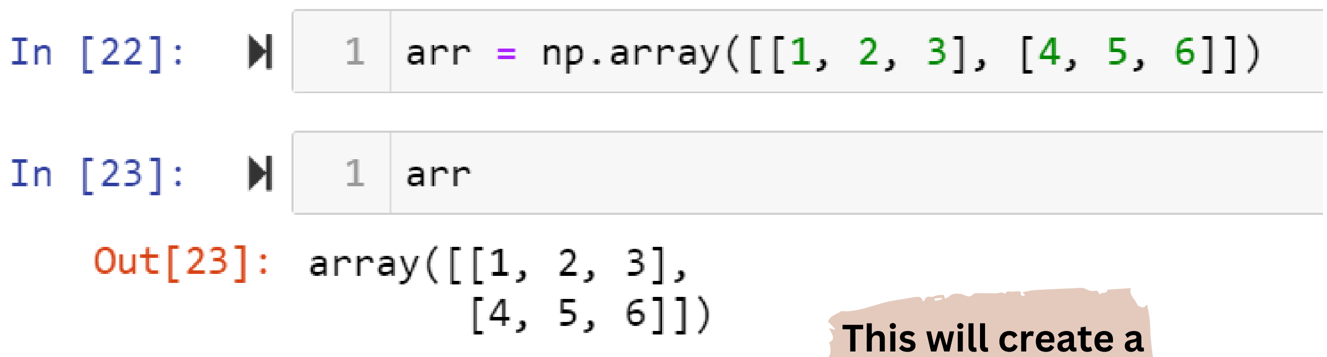


```
In [20]:  # np.random.normal(mean, std_dev,size)

          np.random.normal(1,2,7)


Out[20]: array([-1.68368386,  4.82147992, -1.23195006,  2.530791  , -0.99072528,
                -2.07709323,  0.19107145])
```

Array Attributes

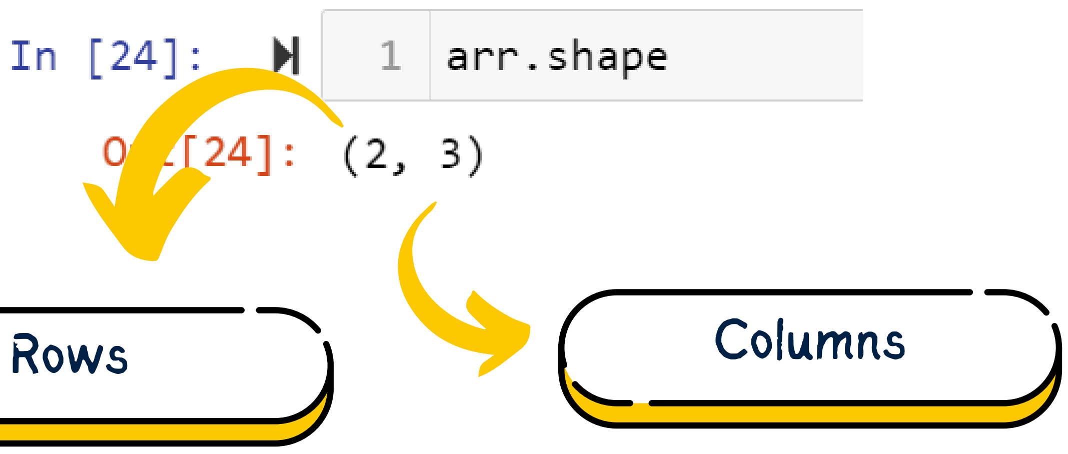
The four array attributes you mentioned are fundamental for understanding and working with NumPy arrays

shape: This attribute returns a tuple representing the dimensions of the array. For a 2D array, it will be a tuple with two elements - the number of rows and the number of columns. For higher-dimensional arrays, it will have more elements.



```
In [22]:  arr = np.array([[1, 2, 3], [4, 5, 6]])

In [23]:  arr

Out[23]: array([[1, 2, 3],
                [4, 5, 6]])
```

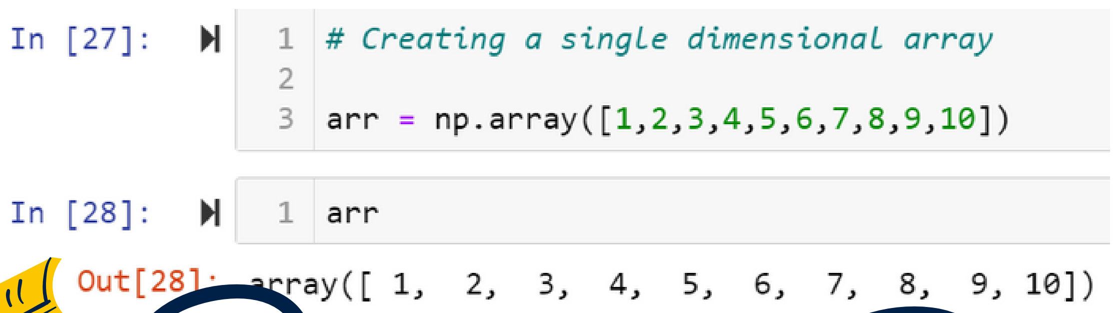
This will create a 2-D array. If we check the sahpe below :



```
In [24]:  arr.shape

Out[24]: (2, 3)
```

There are other methods as well to create array's with multiple rows and multiple columns with the reshape argument.

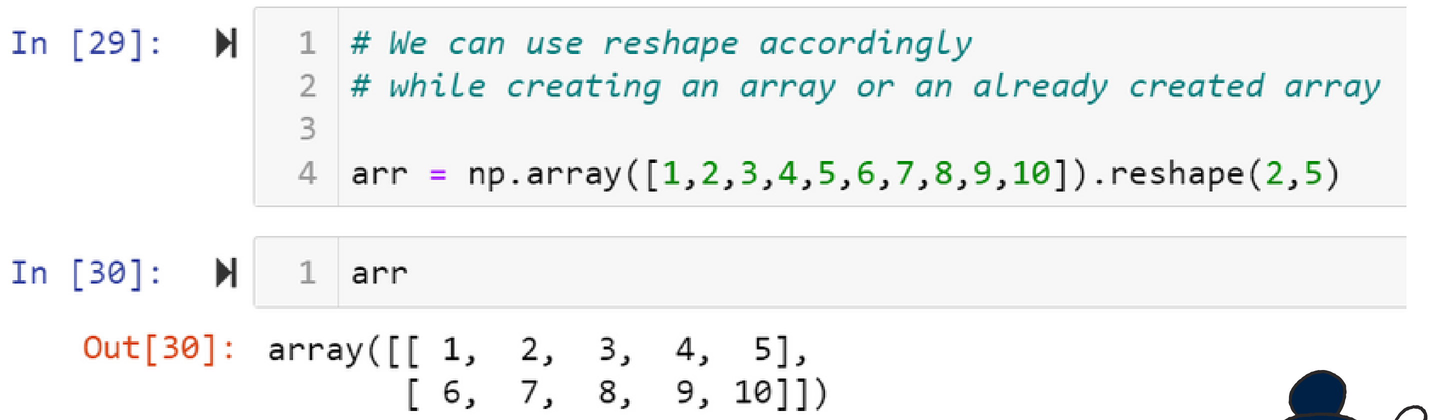
reshape: If we create any array that has single dimension or we use any random methods that create array's that has single dimension we can use reshape method to create the dimensions accordingly. In the reshape method we pass two argument first arguments is meant to be the rows and second one is for columns.



```
In [27]:  # Creating a single dimensional array

          arr = np.array([1,2,3,4,5,6,7,8,9,10])

In [28]:  arr

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

As we have created a single dimension array now we use reshape to make it multi dimesion

We've used reshape method to reshape an array accordingly while creating an array or already created array.

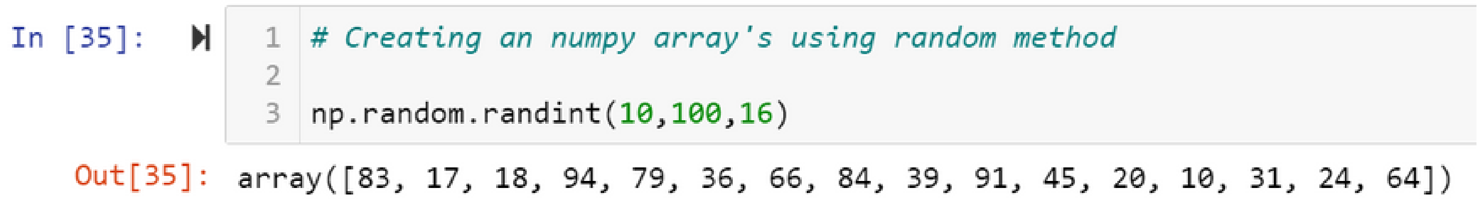


```
In [29]:  # We can use reshape accordingly
          # while creating an array or an already created array

          arr = np.array([1,2,3,4,5,6,7,8,9,10]).reshape(2,5)

In [30]:  arr

Out[30]: array([[ 1,  2,  3,  4,  5],
                [ 6,  7,  8,  9, 10]])
```

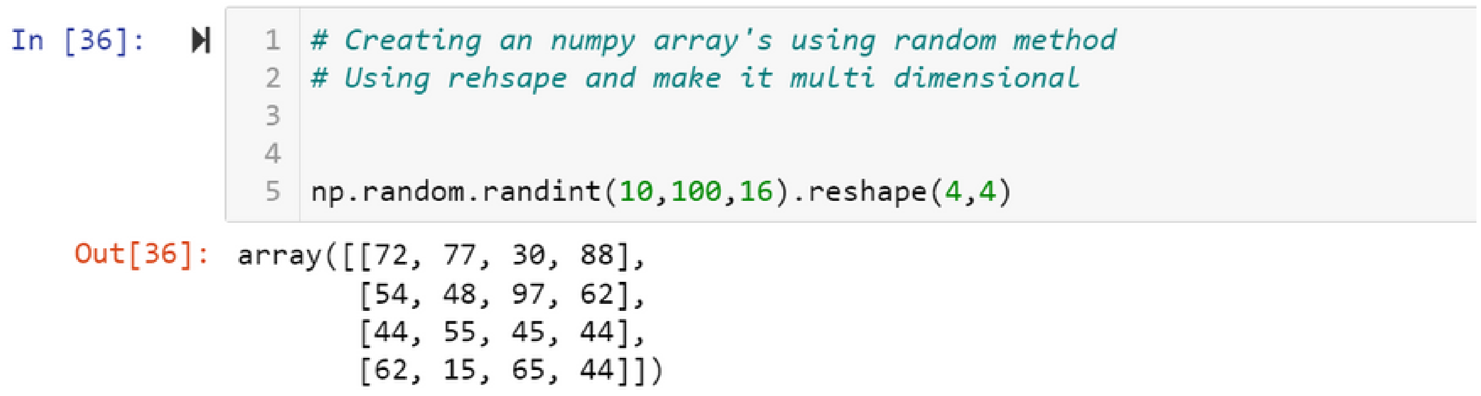
But how should we use the reshape method what to put in the argument ?

As we are using an Method Reshape, first we've to check how many elements are available in an array as above created array we've 10 elements, and in reshape we've provided 2 and 5, if we do 2*5 = 10, will be the answer. Or we can also go with 5*2 = 10. Its just it will change the rows and columns accordingly.

We can also use reshape method with random function while creating an array.

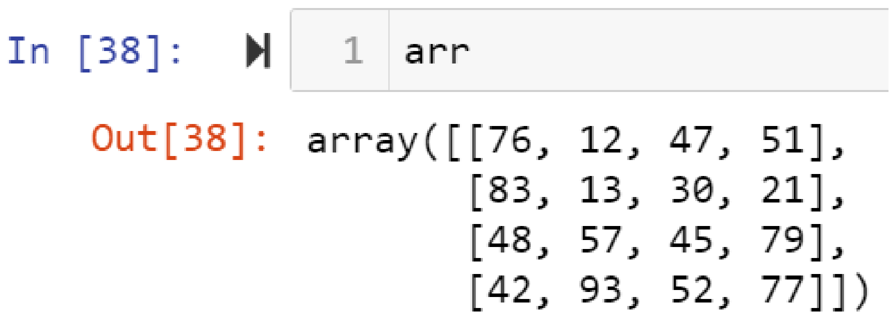


```
In [35]:  # Creating an numpy array's using random method

          np.random.randint(10,100,16)

Out[35]: array([83, 17, 18, 94, 79, 36, 66, 84, 39, 91, 45, 20, 10, 31, 24, 64])
```

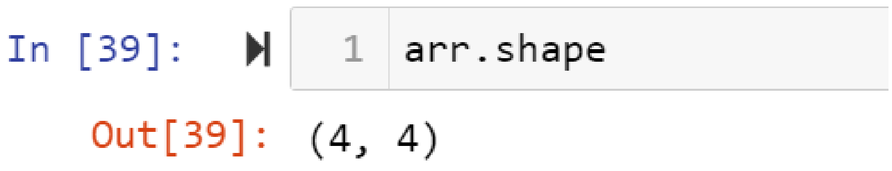


```
In [36]:  # Creating an numpy array's using random method
          # Using rehsape and make it multi dimensional


          np.random.randint(10,100,16).reshape(4,4)

Out[36]: array([[72, 77, 30, 88],
                [54, 48, 97, 62],
                [44, 55, 45, 44],
                [62, 15, 65, 44]])
```

Ok !!! Don’t get confused that the numbers have been changed while creating the multi dimensional array the reson we all know its a RANDOM METHOD it will always change the data whenever we run that command.



```
In [38]:  arr

Out[38]: array([[76, 12, 47, 51],
                [83, 13, 30, 21],
                [48, 57, 45, 79],
                [42, 93, 52, 77]])
```



```
In [39]:  arr.shape

Out[39]: (4, 4)
```

With this shape method we can get the shape of the data how many rows and columns are there !!!

Indexing and Slicing With Array's

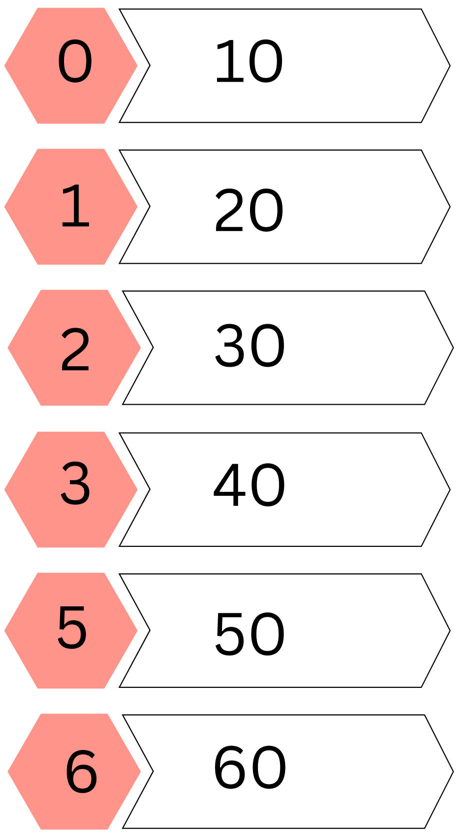
As we‘ve done indexing with python data structures, indexing in single dimensional array‘s are the same. We use indicing when we try to extract data from a Multi Dimensional Data.

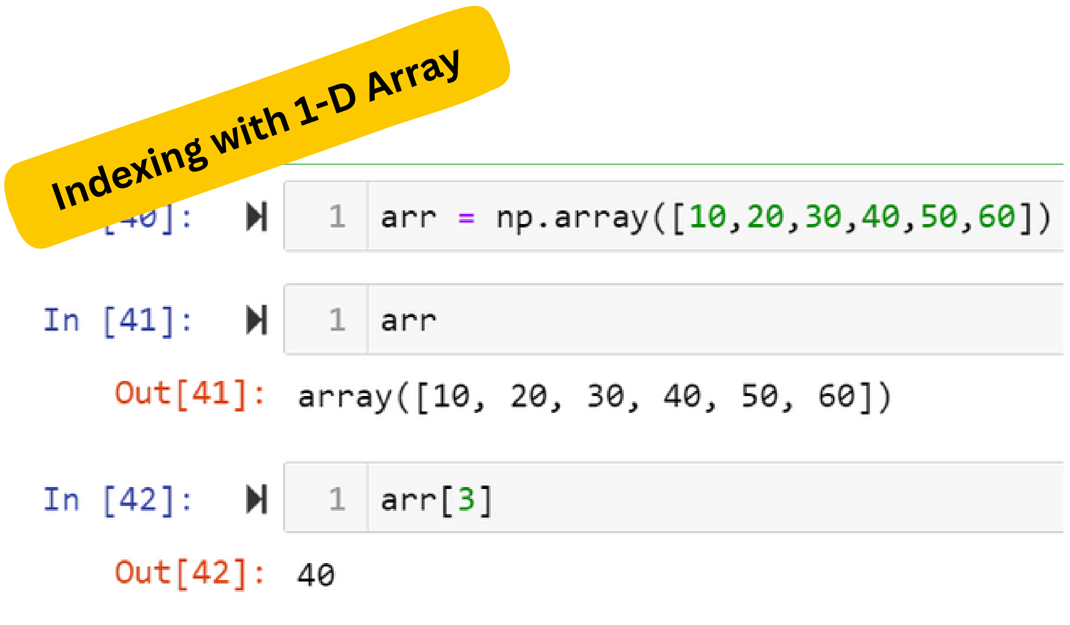
As we know how the normal indexing works

If we have a Data in a List Data Structure, as we all know python indexing starts from 0 and then goes on indexing starts from 0 and then goes on.

data = [10,20,30,40,50,60]

As we do indexing in Python the same pattern will be followed while we do indexing with the array‘s.



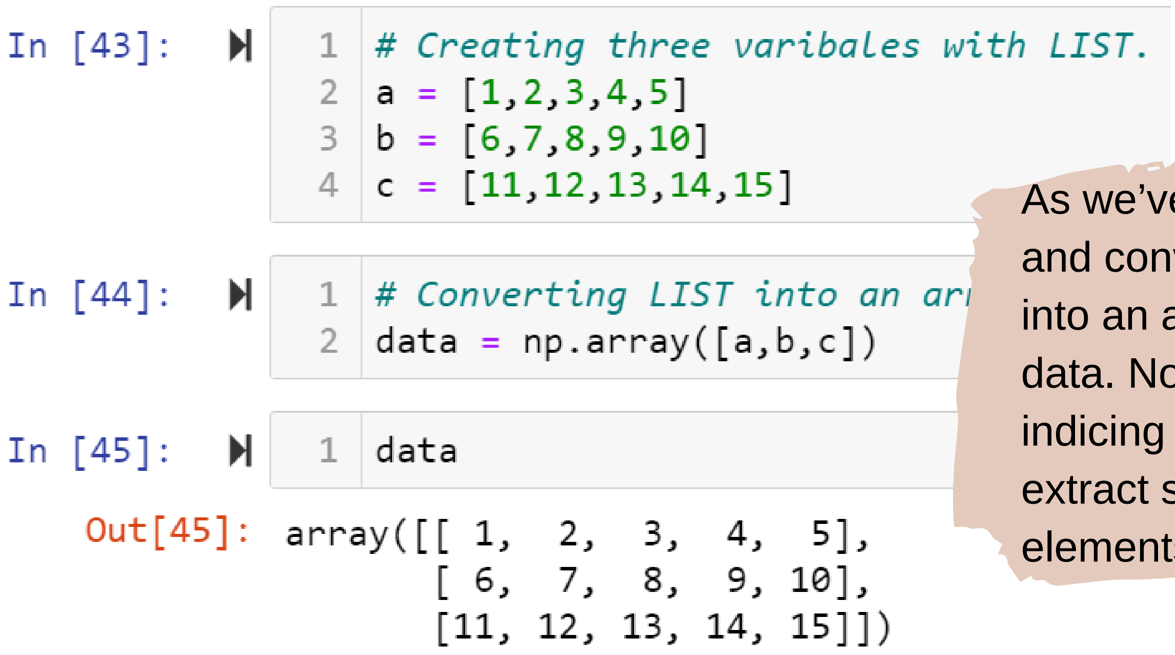


```
In [40]:    arr = np.array([10,20,30,40,50,60])

In [41]:    arr

Out[41]: array([10, 20, 30, 40, 50, 60])

In [42]:    arr[3]

Out[42]: 40
```

Indicing with Multi Dimensional Data

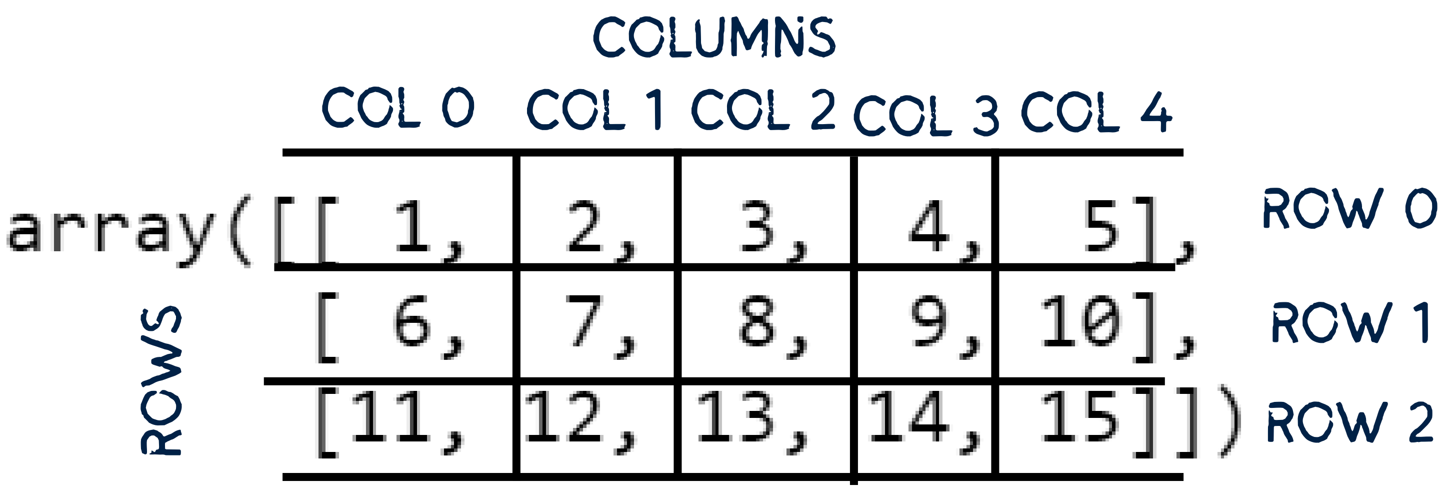
When we do indicing that doesn't work work same as indexing works.

As we all know when we are talking about multiple dimensions there should be multiple rows and columns available. So we need to mention rows and columns when doing indicong.

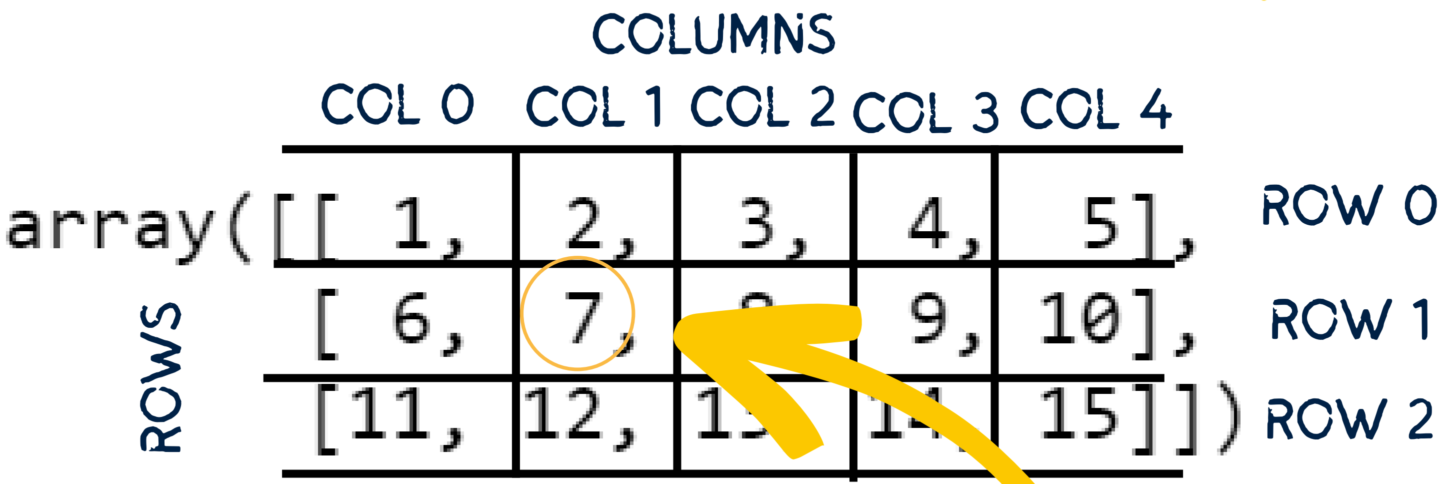


```
In [43]:  # Creating three varibales with LIST.
          a = [1,2,3,4,5]
          b = [6,7,8,9,10]
          c = [11,12,13,14,15]

In [44]:  # Converting LIST into an ar
          data = np.array([a,b,c])

In [45]:  data

Out[45]: array([[ 1,  2,  3,  4,  5],
                [ 6,  7,  8,  9, 10],
                [11, 12, 13, 14, 15]])
```

As we've created a List and converted that list into an array named as data. Now we do indicing from data and extract specific elements



```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
```



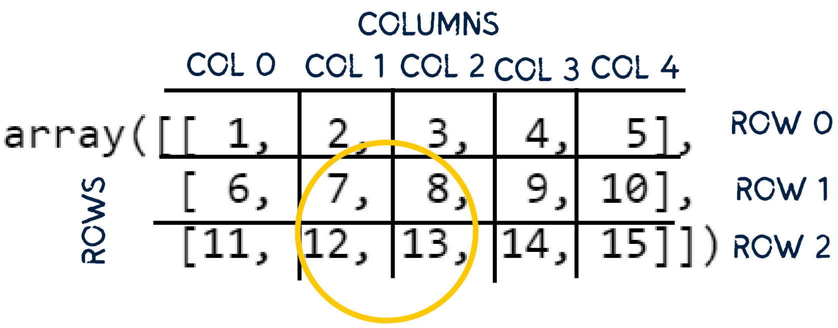
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
```

If we try to extract data then we need to isolate the data elements with array indicings.

We can do indexing where we need to extract multiple data from an array.

```
In [47]:
#data [rows,columns]
data[1,1]

Out[47]: 7
```

Now we want to extract 7.8.12 and 13 from an array.



```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
```

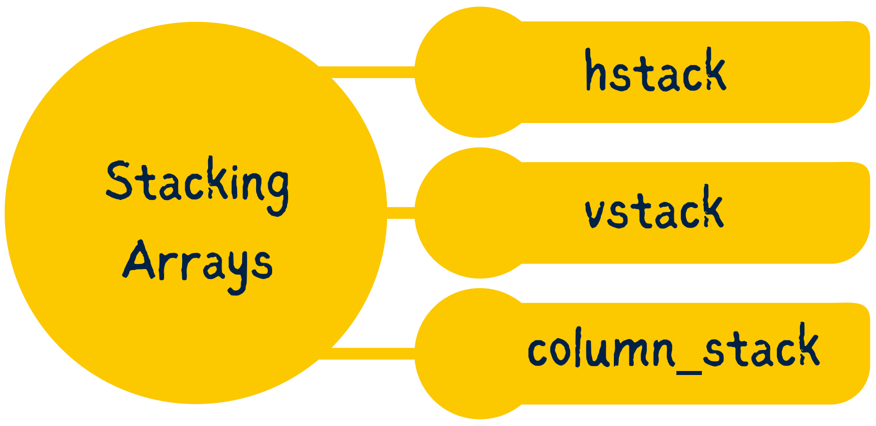
```
In [48]:
#data [rows,columns]
data[1:3,1:3]

Out[48]: array([[ 7,  8],
                [12, 13]])
```

Data Manipulation with Array's

Data Manipulation in Arrays means combining Data that is an Array format and extracting and filtering the data accordingly.

Stacking Arrays

Stacking arrays refers to the process of combining multiple arrays together to form a new array. Depending on the axis along which you stack the arrays, you can achieve different results. Here, I'll show you how to stack arrays in Python using NumPy, a popular library for numerical operations.



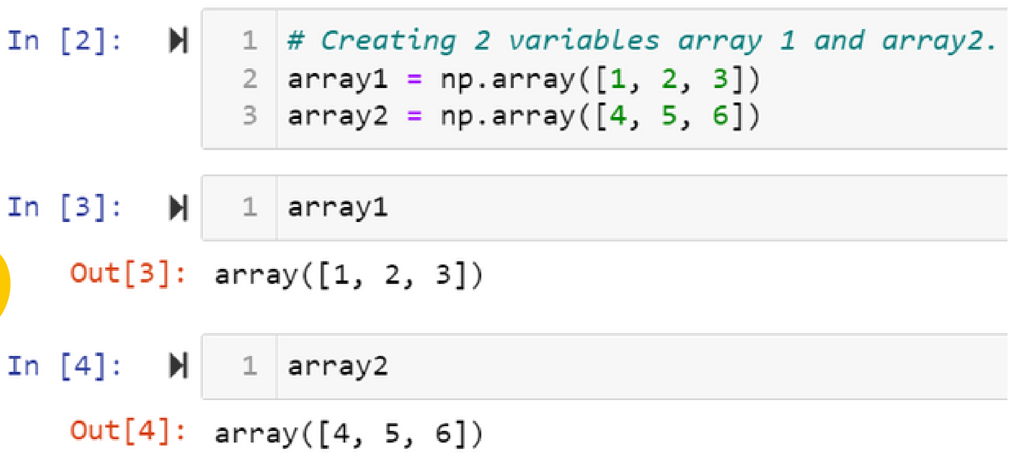
Why do we use Stacking ?

- Combining Data: Stacking allows you to combine multiple arrays or datasets into a single array. This is useful when you have different sources of data that you want to merge for analysis or processing.

- Creating New Features: When working with machine learning or data analysis, you might want to create new features or attributes from existing ones. Stacking arrays can help you create composite features by combining or concatenating multiple arrays

hstack

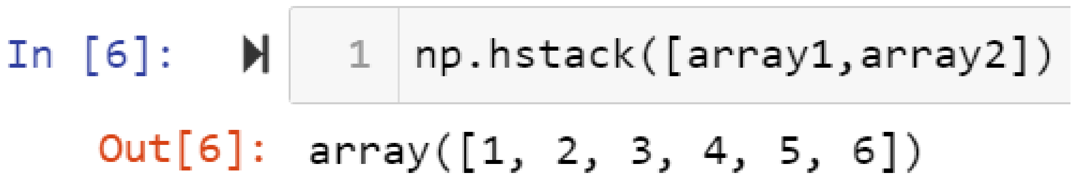
Horizontal stacking, often referred to as hstack, is the process of combining multiple arrays horizontally along the columns. This means that the arrays are stacked side by side, effectively extending the number of columns in the resulting array.



```
In [2]:  # Creating 2 variables array 1 and array2.
         array1 = np.array([1, 2, 3])
         array2 = np.array([4, 5, 6])

In [3]:  array1

Out[3]:  array([1, 2, 3])

In [4]:  array2

Out[4]:  array([4, 5, 6])
```

Array1 and Array2 has been created

We are gonna use hstack method to combine both the variables that contains array array1 and array2.



```
In [6]:  np.hstack([array1,array2])

Out[6]:  array([1, 2, 3, 4, 5, 6])
```

We have used hstack and the data has been stacked horizontally. It woks like the append method from python

vstack

np.vstack is a function provided by the NumPy library in Python for vertical stacking of arrays. Vertical stacking, or "vstack," refers to stacking arrays on top of each other along the rows. It combines arrays by adding rows, effectively increasing the number of rows in the resulting array.

```
In [10]:  array1 = np.array([1, 2, 3])
          array2 = np.array([4, 5, 6])

In [11]:  vertical_stack = np.vstack((array1, array2))

In [12]:  vertical_stack

Out[12]: array([[1, 2, 3],
                [4, 5, 6]])
```

After using the vstack method we can see there is a new row has been created and the data becomes 2 dimensional.

column_stack

Column stacking, often referred to as "column_stack," is the process of stacking arrays side by side along columns to create a new array. This operation is useful when you want to combine multiple 1D arrays or lists into a 2D array, where each original array becomes a column in the resulting 2D array.

```
In [16]:  # we use the same array1 and array2

          np.column_stack([array1,array2])

Out[16]: array([[1, 4],
                [2, 5],
                [3, 6]])
```

vsplit

The np.vsplit() function in NumPy is used to split a multi-dimensional array (e.g., a 2D array) into multiple subarrays along the vertical axis, which means it splits the array into smaller arrays or "chunks" along the rows. This can be particularly useful when dealing with tabular data or any data where the rows represent individual records or observations.

np.vsplit(array, indices_or_sections)

array: The array you want to split.
indices_or_sections: It can be an integer or a list of indices that specify where to split the array along the rows.

```
In [20]:
# Create a sample 2D array

arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9],
                [10, 11, 12]])
```

```
In [21]:
# Split the array into two equal parts along the rows
subarrays = np.vsplit(arr, 2)
```

```
In [22]:
subarrays
```

```
Out[22]: [array([[1, 2, 3],
                [4, 5, 6]]),
          array([[ 7,  8,  9],
                [10, 11, 12]])]
```

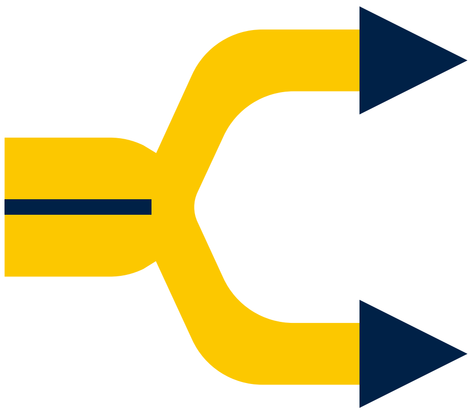
splitted array

splitted array

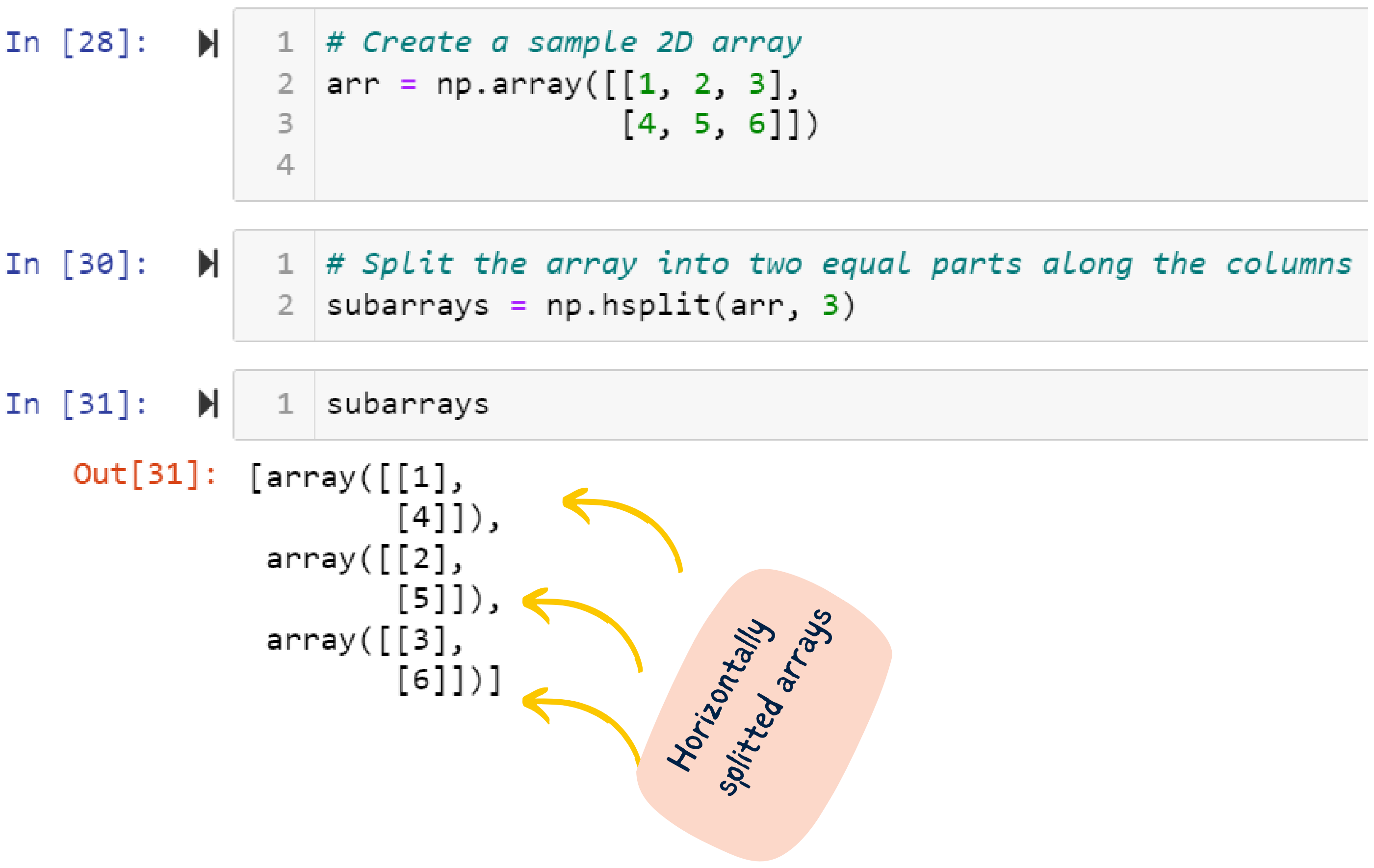
hsplit

The np.hsplit() function in NumPy is used to split a multi-dimensional array (e.g., a 2D array) into multiple subarrays along the horizontal axis, which means it splits the array into smaller arrays or "chunks" along the columns. This can be particularly useful when dealing with tabular data or any data where the columns represent different features or attributes.

np.hsplit(array, indices_or_sections)



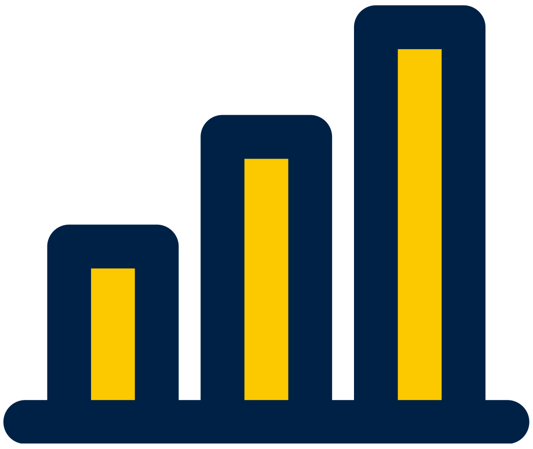
array: The array you want to split.
indices_or_sections: It can be an integer or a list of indices that specify where to split the array along the columns.



```
In [28]:  # Create a sample 2D array
          arr = np.array([[1, 2, 3],
                          [4, 5, 6]])

In [30]:  # Split the array into two equal parts along the columns
          subarrays = np.hsplit(arr, 3)

In [31]:  subarrays

Out[31]: [array([[1],
                 [4]]),
          array([[2],
                 [5]]),
          array([[3],
                 [6]])]
```

Statistics with Numpy Array's

NumPy provides a wide range of functions for statistical analysis and calculations. Here are some common statistical operations you can perform with NumPy



Mean **Median**

Mode

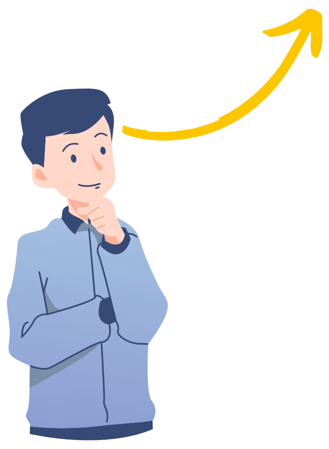
Variance **std**

Suppose we've an array that has more than 20 Elements in it and we want to find out the descriptive values from that array like mean, median and mode etc. Its really manual task to find them all. So for that we can use numpy methods

Let's Dive into the statistical methods with Numpy.

So, first let's create an array. We use randint method with reshape to create an array.

```
In [4]:  data = np.random.randint(25,75,20).reshape(5,4)

In [5]:  data

Out[5]: array([[57, 27, 37, 69],
               [61, 39, 46, 45],
               [41, 31, 74, 40],
               [63, 46, 57, 37],
               [37, 49, 44, 50]])
```



Now if we want to find out the description of statistics of that array manually we know it will take a lot of time.

Mean

```
In [6]:  # Getting Mean
         np.mean(data)

Out[6]: 47.5
```

Median

```
In [7]:  # Getting Median
         np.median(data)

Out[7]: 45.5
```

Variance

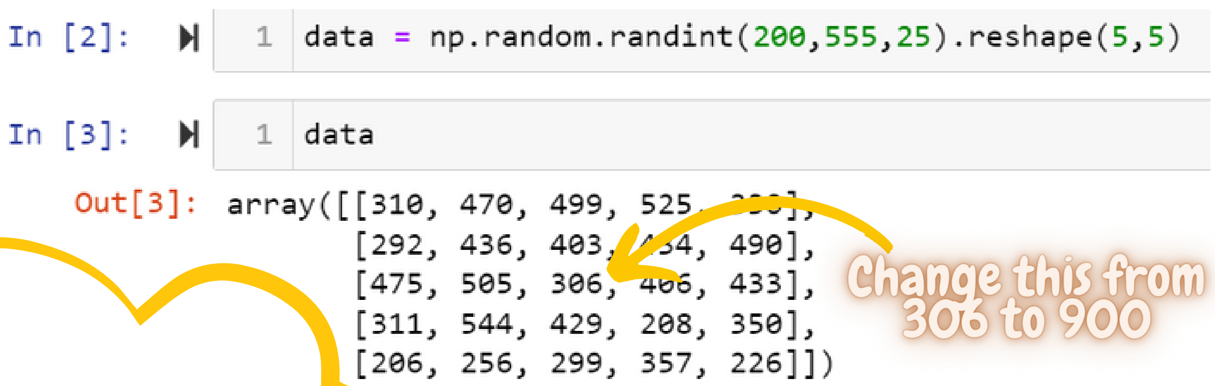
```
In [8]:  # Getting Variance
         np.var(data)

Out[8]: 149.65
```

std

```
In [9]:  # Getting Standard Deviation
         np.std(data)

Out[9]: 12.23315167894194
```

Specific Array Replacement with Numpy

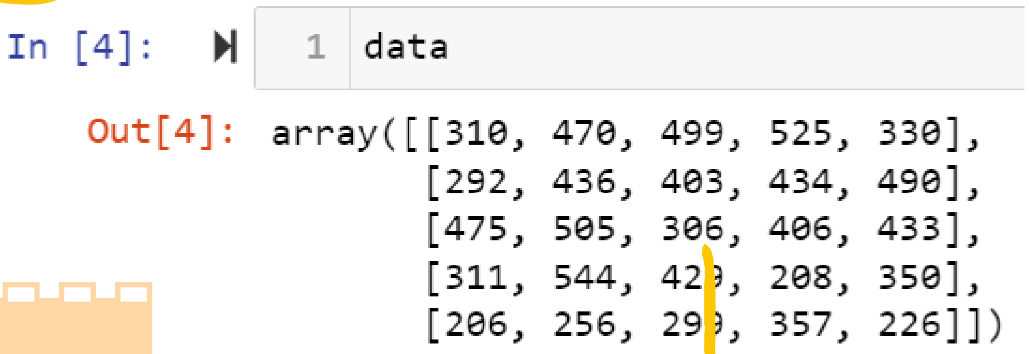
Here we've createdan array with random randint method that contain values from 200 to 555 and the array has 25 values in it and then we use rehape method to create 5 rows and 5 columns.



```
In [2]:  data = np.random.randint(200,555,25).reshape(5,5)

In [3]:  data

Out[3]: array([[310, 470, 499, 525, 330],
               [292, 436, 403, 434, 490],
               [475, 505, 306, 406, 433],
               [311, 544, 429, 208, 350],
               [206, 256, 299, 357, 226]])
```

Change this from 306 to 900

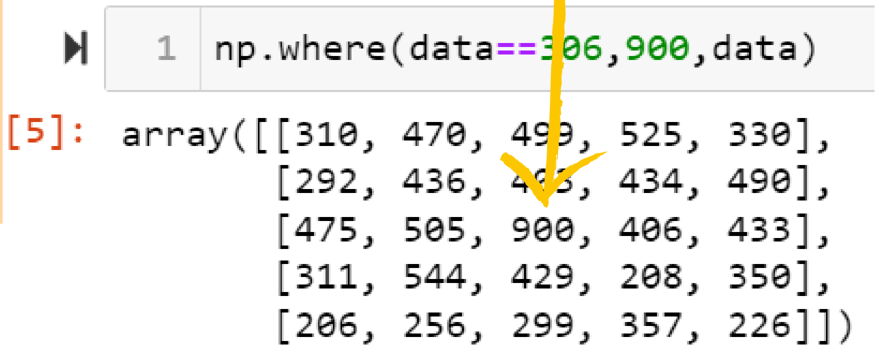
Now from data we need to change a specific value that is 306 to 900 by using where method.



```
In [4]:  data

Out[4]: array([[310, 470, 499, 525, 330],
               [292, 436, 403, 434, 490],
               [475, 505, 306, 406, 433],
               [311, 544, 429, 208, 350],
               [206, 256, 299, 357, 226]])
```

Where method basically takes 3 arguments:

1. Condition (data==306)
2. Replacement (900)
3. Reflection (data)



```
         np.where(data==306,900,data)

[5]: array([[310, 470, 499, 525, 330],
            [292, 436, 403, 434, 490],
            [475, 505, 900, 406, 433],
            [311, 544, 429, 208, 350],
            [206, 256, 299, 357, 226]])
```