----------------------NumPy--------------------------

What is numpy:

- NumPy stands for Numerical Python. Num- Numerical and Py- Python.
- NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.
- NumPy is a Python library used for working with arrays.
- Support large number of data in the form of multi-dimensional array and matrix.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.

Why NumPy ?

- In Python we have lists that serve the purpose of arrays, but they are slow to process.

- NumPy aims to provide an array object that is up to **50x faster than traditional Python lists.**

- The array object in NumPy is called `ndarray`, it provides a lot of supporting functions that make working with `ndarray` very easy.

- NumPy data structures perform better in:

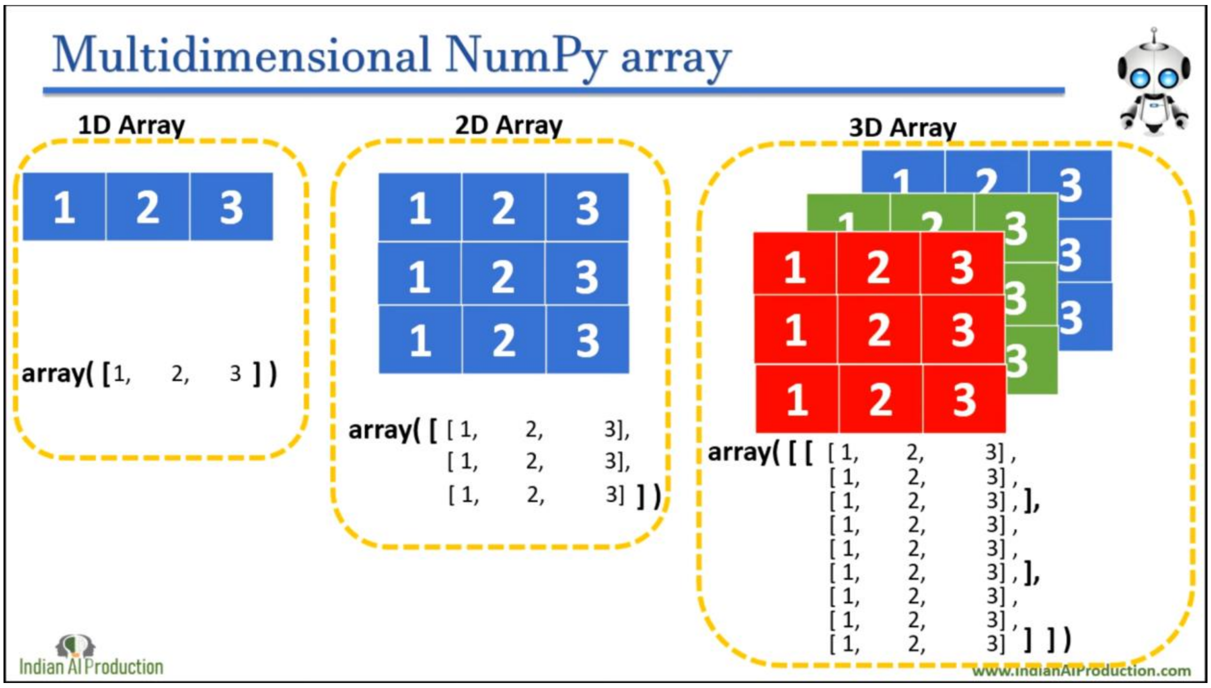
  - Memory – NumPy data structures **take up less space**.
  - Performance – They have a **need for speed** and are faster than lists.
  - Functionality – SciPy and NumPy have optimized functions such as linear algebra operations built in.

NOTE: Every item in an ndarray takes the same size of block in the memory. Each element in ndarray is an object of data-type object also called as **dtype**.

Which Language is NumPy written in?

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Multidimensional NumPy array:


Multidimensional NumPy array
1D Array
2D Array
3D Array
array([1, 2, 3])
array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
array([[[1, 2, 3] , [1, 2, 3] , [1, 2, 3] ,], [1, 2, 3] , [1, 2, 3] , [1, 2, 3] ,], [1, 2, 3] , [1, 2, 3] , [1, 2, 3]]])
Indian AI Production
www.IndianAIProduction.com

NumPy vs Python list:

NumPy arrays are **stored at one continuous place in memory** unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

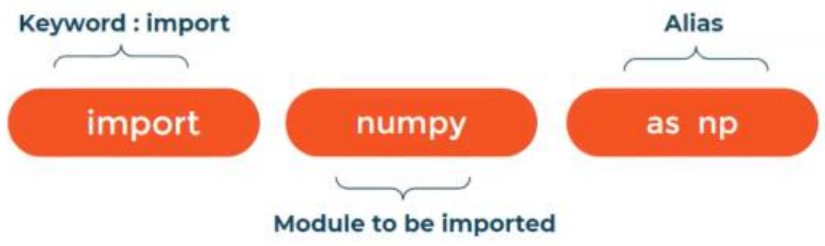
This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

NumPy is written in C language. Because the C language is Fast, NumPy provide:

1) Fast Process
2) Use less memory to store data
3) Convenient (We can easily perform math operation in NumPy as compared to list)

Importing NumPy:

```
import numpy as np
```



How to Create NumPy array:

- **NumPy is used to work with arrays**. The array object in NumPy is called `ndarray`.

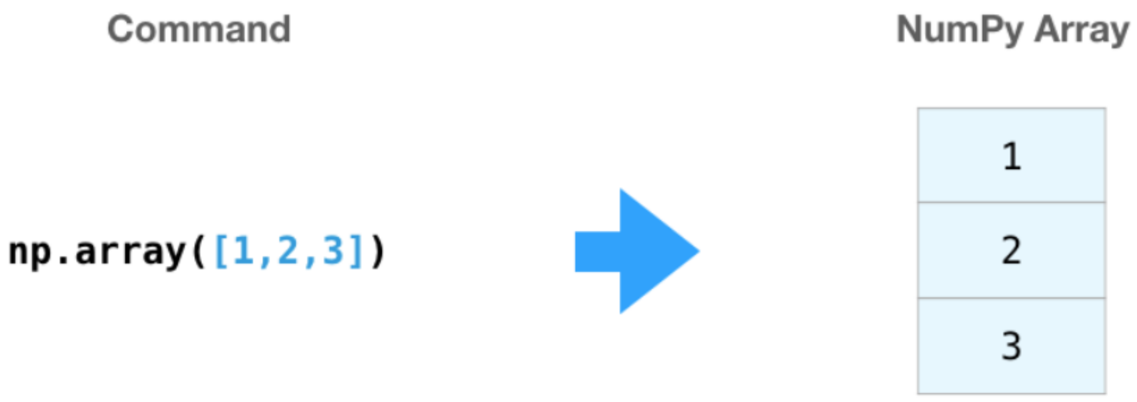
- We can create a NumPy `ndarray` object by using the `array()` function.

1.Create zero dimensional array:

```
import numpy as np
zero_dimensional_array = np.array(42)
print(zero_dimensional_array)
-----------------------------------------------------------------------------
>>> 42
```

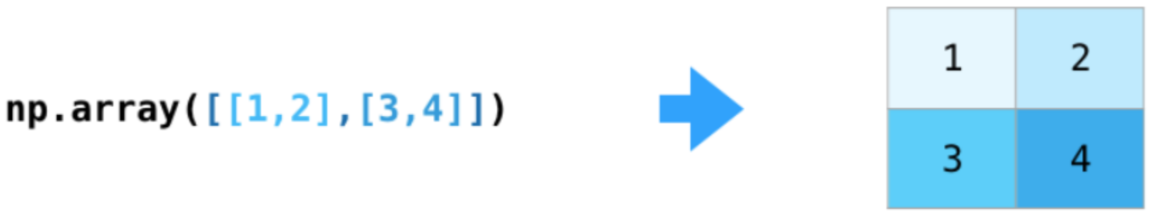
2.Create one dimensional array:

```
import numpy as np

one_dimensional_array = np.array([1,2,3])
print(one_dimensional_array)
-----------------------------------------------------------------------------

>>> [1 2 3]
```



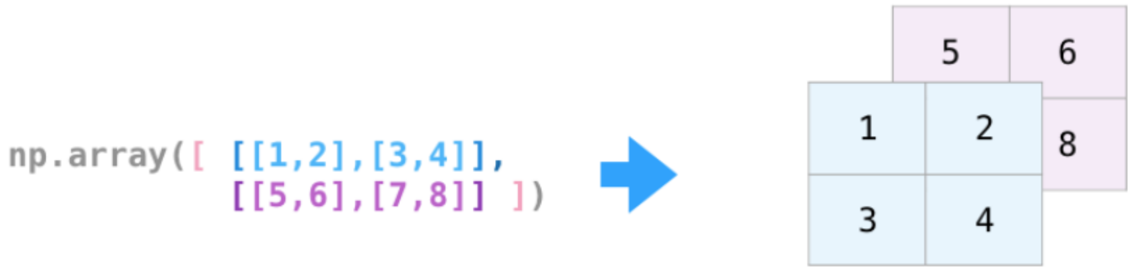
3.Create two-dimensional array:

```
import numpy as np
two_dimensional_array = np.array([[1,2],[3,4]])
print(two_dimensional_array)
-----------------------------------------------------------------------------


>>> [[1 2]
    [3 4]]
```



4.Create three-dimensional array:

```
import numpy as np
three_dimensional_array = np.array([[[1,2],[3,4]],
                                    [[5,6],[7,8]]])
print(three_dimensional_array)
print(three_dimensional_array.shape)
-------------------------------------------------------------------------


 >>> [[[1 2]

     [3 4]]

     [[5 6]
     [7 8]]]

     (2, 2, 2)
```



Check Number of dimensions ?

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

>>> 0
    1
    2
    3
```

5.Uninitialized

```
import numpy as np
ar = np.empty([5,3],dtype = int)
print(ar)
----------------------------------------------------------------------------

>>> [[ 0  0  0]

   [ 0  0  0]

   [ 0  0  0]

   [ 0 0  0]

   [ 0  0  0]]
```

6.Initialized with zeros:

```
import numpy as np
a = np.zeros([3,3])  #Default dtype = float
print(a)
----------------------------------------------------------------------------

>>> [[0.  0.  0.]
   [0.  0.  0.]
   [0.  0.  0.]]
```

7.Initialized with ones:

```
import numpy as np
mx_1s = np.ones(5)
print(mx_1s)
----------------------------------------------------------------------------

>>> [1. 1. 1. 1. 1.]
```

8.To create ones 2d Dimensional Array:

```
import numpy as np
mx_1s = np.ones((3,4))
print(mx_1s)
-----------------------------------------------------------------------------

>>> [[1. 1. 1. 1.]
     [1. 1. 1. 1.]
     [1. 1. 1. 1.]]
```

9.Array from specified numbers:

```
import numpy as np
a = np.full([3,3],30)
print(a)
-----------------------------------------------------------------------------

>>> [[30 30 30]
    [30 30 30]
    [30 30 30]]
```

10.To create an identity matrix:

```
import numpy as np
a = np.identity(3)
print(a)
-----------------------------------------------------------------------------

>>> [[1. 0. 0.]
    [0. 1. 0.]
    [0. 0. 1.]]
```

11.Creating ndarray with existing data:

List to Ndarray:

```
import numpy as np
list_1 = [1,2,3,4]
ar = np.asarray(list_1)   # You can use array() also
print(ar)
-----------------------------------------------------------------------------

>>> [1 2 3 4]
```

Tuple to Ndarray:

```
import numpy as np
tuple_1 = (1,2,3,4)
```

```
ar = np.asarray(tuple_1)
print(ar)
-----------------------------------------------------------------------------

>>> [1 2 3 4]
```

12.Array from numerical ranges

```
import numpy as np
array_with_range = np.arange(5)
print(array_with_range)
-----------------------------------------------------------------------------


>>> [0 1 2 3 4]



another_array_with_range = np.arange(10,20,2)
print(another_array_with_range)
-----------------------------------------------------------------------------

>>> [10 12 14 16 18]
```

To change the type:

```
import numpy as np
mx_1s = np.ones((3,4), dtype= int)
print(mx_1s)
>>> [[1 1 1 1]
     [1 1 1 1]
     [1 1 1 1]]
-----------------------------------------------------------------------------
import numpy as np
mx_1s = np.ones((3,4), dtype= bool)
print(mx_1s)

>>> [[ True  True  True  True]
     [ True  True  True  True]
     [ True  True  True  True]]
```

NOTE:If you are not sure about how many rows and columns to give just put -1 .

Access Array Elements:

- Array indexing is the same as accessing an array element.
- You can access an array element by referring to its index number.
- The indexes in **NumPy arrays start with 0**, meaning that the first element has index 0, and the second has index 1 etc.

Access 1-D Arrays:

Get the first element from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
---------------------------------------------------------------------------

>>> 1
```

Get the second element from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[1])
---------------------------------------------------------------------------

>>> 2
```

Get the third and fourth element from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[2] + arr[3])
---------------------------------------------------------------------------

>>> 7
```

Access 2-D Arrays:

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Access the 2nd element on 1st dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st dim: ', arr[0, 1])
---------------------------------------------------------------------------
```

```
>>> 2nd element on 1st dim:  2
```

Access the 5th element on 1st dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('5th element on 2nd dim: ', arr[1, 4])
---------------------------------------------------------------------------
>>> 5th element on 2nd dim:  10
```

Access 3-D Arrays:

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

Access the third element of the second array of the first array:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
---------------------------------------------------------------------------

>>> 6
```

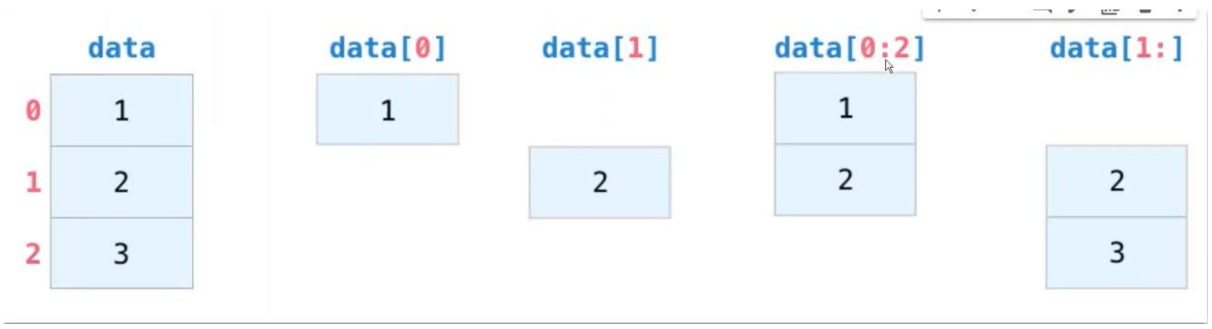
Negative Indexing:

Use negative indexing to access an array from the end.

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
---------------------------------------------------------------------------

>>>Last element from 2nd dim:  10
```

Slicing:

- Slicing in python means taking elements from one given index to another given index.
- We pass slice instead of index like this: `[start:end]`.
- We can also define the step, like this: `[start:end:step]`.
- If we don't pass start its considered 0.
- If we don't pass end its considered length of array in that dimension.
- If we don't pass step its considered 1.



```
import numpy as np

mx = np.arange(1,101).reshape(10,10)
print(mx)
---------------------------------------------------------------------------

>>> [[  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  33  34  35  36  37  38  39  40]
     [ 41  42  43  44  45  46  47  48  49  50]
     [ 51  52  53  54  55  56  57  58  59  60]
     [ 61  62  63  64  65  66  67  68  69  70]
     [ 71  72  73  74  75  76  77  78  79  80]
     [ 81  82  83  84  85  86  87  88  89  90]
     [ 91  92  93  94  95  96  97  98  99 100]]
```

We will take this above NumPy array for below Operations:

Access the Element:

```
sl_1 = mx[2 , 6]
print(sl_1)
------------------------------------------------------------------------------

>>> 27
```

Access the whole Row:

```
sl_2 = mx[2]
print(sl_2)
------------------------------------------------------------------------------

>>> [21 22 23 24 25 26 27 28 29 30]
```

Access the column (But in row format):

```
sl_3 = mx[:, 0]
print(sl_3)
------------------------------------------------------------------------------

>>> [ 1 11 21 31 41 51 61 71 81 91]
```

Access the columns:

```
sl_4 = mx[:, 0:2]
print(sl_4)
------------------------------------------------------------------------------

>>>  [[ 1  2]
     [11 12]
     [21 22]
     [31 32]
     [41 42]
     [51 52]
     [61 62]
     [71 72]
     [81 82]
     [91 92]]
```

Slice some part of an array:

```
sl_5 = mx[1:4, 1:4]
print(sl_5)
------------------------------------------------------------------------------

>>> [[12 13 14]
    [22 23 24]
    [32 33 34]]
```

Access the whole matrix:

```
# There are three ways to access the print the whole matrx
sl_6 = mx[:]
sl_7 = mx[::]
sl_8 = mx[:,:]
```

Concatenate:

Create an two array to concatenate them:

```
import numpy as np

con_1 = np.arange(1,17).reshape(4,4)
con_2 = np.arange(17,33).reshape(4,4)

print(con_1)
print(con_2)
--------------------------------------------------------------------------


>>> [[ 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]]
```

Concatenate Column vice:

```
import numpy as np

con_1 = np.arange(1,17).reshape(4,4)
con_2 = np.arange(17,33).reshape(4,4)


concatenate_1 = np.concatenate((con_1,con_2))
print(concatenate_1)
--------------------------------------------------------------------------


>>> [[ 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]]


concatenate_2 = np.vstack((con_1,con_2)) # v for vertical
print(concatenate_2)
---------------------------------------------------------------------------

>>> [[ 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]]
```

Concatenate Row vice:

```
concatenate_3 = np.concatenate((con_1,con_2), axis=1)
print(concatenate_3)
---------------------------------------------------------------------------

>>> [[ 1   2   3   4  17  18  19  20]
     [ 5   6   7   8  21  22  23  24]
     [ 9  10  11  12  25  26  27  28]
     [13  14  15  16  29  30  31  32]]

concatenate_4 = np.hstack((con_1,con_2)) # h for horizontal
print(concatenate_4)
>>> [[ 1   2   3   4  17  18  19  20]
     [ 5   6   7   8  21  22  23  24]
     [ 9  10  11  12  25  26  27  28]
     [13  14  15  16  29  30  31  32]]
```

NumPy Data Types:

NumPy has some extra data types, and refer to data types with one character, like `i` for integers, `u` for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- `i` - integer
- `b` - boolean
- `u` - unsigned integer
- `f` - float
- `c` - complex float
- `m` - timedelta
- `M` - datetime
- `O` - object
- `S` - string
- `U` - unicode string
- `V` - fixed chunk of memory for other type (void)

Checking the Data Type of an Array:

The NumPy array object has a property called `dtype` that returns the data type of the array:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr.dtype)
---------------------------------------------------------------------------

>>>int32
```

Creating Arrays with a Defined Data Type:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)
---------------------------------------------------------------------------

>>> [b'1' b'2' b'3' b'4']
    |S1
```

For `i`, `u`, `f`, `S` and `U` we can define size as well.

Converting Data Type on Existing Arrays:

The best way to change the data type of an existing array, is to make a copy of the array with the `astype()` method.

The `astype()` function creates a copy of the array, and allows you to specify the data type as a parameter.

The data type can be specified using a string, like `'f'` for float, `'i'` for integer etc. or you can use the data type directly like `float` for float and `int` for integer.

```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
new_arr = arr.astype('i')
print(new_arr)
print(new_arr.dtype)
```

```
>>> [1 2 3]

    int32
```

NumPy Array Copy vs View:

The Difference Between Copy and View:

The main difference between a copy and a view of an array is that **the copy is a new array, and the view is just a view of the original array.**

The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

COPY:

Make a copy, change the original array, and display both arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42
print(arr)
print(x)
---------------------------------------------------------------------------

>>> [42  2  3  4  5]
    [1 2 3 4 5]
```

VIEW:

Make a view, change the original array, and display both arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42
print(arr)
print(x)
---------------------------------------------------------------------------

>>> [42  2  3  4  5]
    [42  2  3  4  5]
```

Make Changes in the VIEW:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
```

```
x = arr.view()
x[0] = 31
print(arr)
print(x)
------------------------------------------------------------------------------

>>> [31  2  3  4  5]
    [31  2  3  4  5]
```

NumPy Array Shape:

The shape of an array is the number of elements in each dimension.

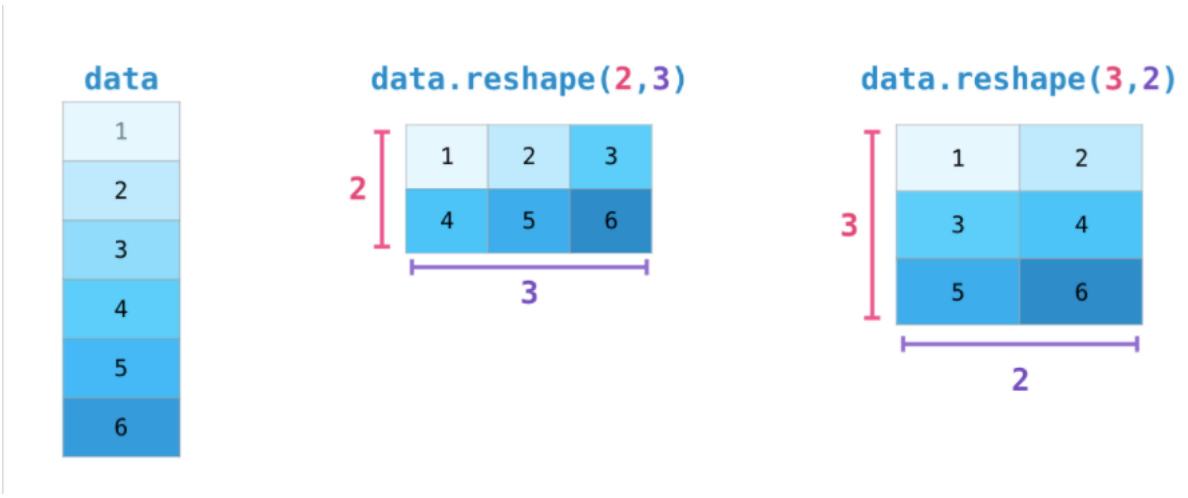
NumPy arrays have an attribute called `shape` that returns a tuple with each index having the number of corresponding elements.

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
---------------------------------------------------------------------------

>>> (2, 4)
```

The example above returns `(2, 4)`, which means that **the array has 2 dimensions, and each dimension has 4 elements.**

NumPy Array Reshaping:

Reshaping means gives a new shape to an array without changing its data.



The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D:

Reshaping Converting the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
---------------------------------------------------------------------------


>>> [[ 1  2  3]
    [ 4  5  6]
    [ 7  8  9]
    [10 11 12]]
```

Reshape From 1-D to 3-D:

Convert the following 1-D array with 12 elements into a 3-D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)
print(newarr)
----------------------------------------------------------------------------


>>> [[[ 1  2]
     [ 3  4]
     [ 5  6]]

    [[ 7  8]
     [ 9 10]
     [11 12]]]
```

NOTE: We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

Unknown Dimension:

You are allowed to have one "unknown" dimension.

Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.

Pass `-1` as the value, and NumPy will calculate this number for you.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])newarr = arr.reshape(2, 2, -1)
print(newarr)
----------------------------------------------------------------------------


>>> [[[1 2]
      [3 4]]

     [[5 6]
      [7 8]]]
```

Note: We can not pass `-1` to more than one dimension.

Flattening the arrays:

Flattening array means converting a multidimensional array into a 1D array.

We can use `reshape(-1)` to do this.

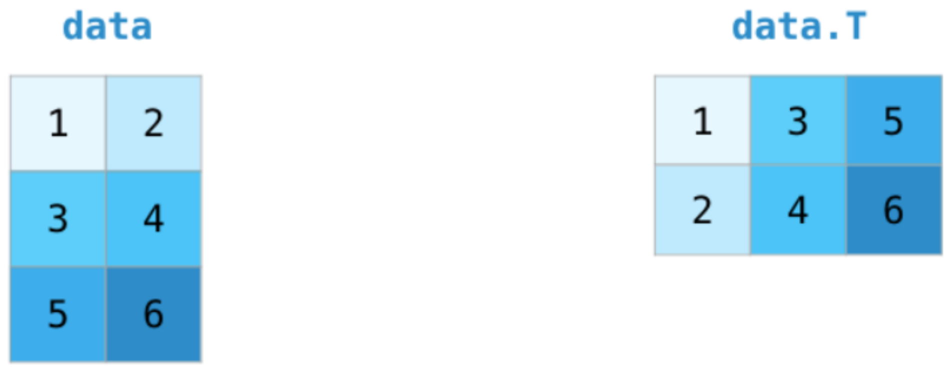
Convert the array into a 1D array.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
newarr = arr.reshape(-1)
print(newarr)

-----------------------------------------------------------------------------


>>> [1 2 3 4 5 6]
```

NOTE: There are a lot of functions for changing the shapes of arrays in numpy flatten, ravel and also for rearranging the elements rot90, flip, fliplr, flipud etc. These fall under Intermediate to advanced section of numpy

NumPy transpose Array:

Transpose is defined as an operator which flips a matrix over its diagonal; that is, it switches the row and column indices of the matrix A by producing another matrix, often denoted by At



```
# Create a 2 D Array
import numpy as np
simple_array = np.array(np.random.randint(10,45,(8,2)))
print(f"The defined array is \n{simple_array}")
print(f" \n The shape of the array is {simple_array.shape}")
----------------------------------------------------------------------------


>>> The defined array is
    [[35 21]
     [40 28]
     [35 37]
     [24 37]
     [15 11]
     [33 19]
     [42 28]
     [16 29]]

     The shape of the array is (8, 2)
```

Let's transpose the above array.

```
# Storing the transpose in another matrix object
import numpy as np
simple_array = np.array(np.random.randint(10,45,(8,2)))
transpose_array =np.transpose(simple_array)
print(f" The transpose of the array is \n{transpose_array}")
print(f" \n The shape of the array is {transpose_array.shape}")
----------------------------------------------------------------------------


>>> The transpose of the array is
  [[24 24 13 41 35 38 17 25]
   [30 42 20 33 33 25 34 11]]

   The shape of the array is (2, 8)
```

NumPy Joining Array:

Joining means putting contents of two or more arrays in a single array.

In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.

We pass a sequence of arrays that we want to join to the `concatenate()` function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Join three arrays.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr3 = np.array([7, 8, 9])
arr = np.concatenate((arr1, arr2, arr3))
print(arr)
----------------------------------------------------------------------------

>>> [1 2 3 4 5 6 4 5 6]
```

Join two 2-D arrays along rows (axis = 1):

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
arr = np.concatenate((arr1, arr2), axis=1)
print(arr)
----------------------------------------------------------------------------

>>> [[1 2 5 6]
     [3 4 7 8]]
```

Joining Arrays Using Stack Functions:

Stacking is same as concatenation, the only difference is that stacking is done along a new axis.

We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.

We pass a sequence of arrays that we want to join to the `stack()` method along with the axis. If axis is not explicitly passed it is taken as 0.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
```

```
arr = np.stack((arr1, arr2), axis=1)
print(arr)
-------------------------------------------------------------------------


>>> [[1 4]
     [2 5]
     [3 6]]
```

NumPy Splitting Array:

Splitting is reverse operation of Joining.

Joining merges multiple arrays into one and Splitting breaks one array into multiple.

We use `array_split()` for splitting arrays, we pass it the array we want to split and the number of splits.

Splitting 1-D Arrays:

Split the array in 3 parts:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr)
----------------------------------------------------------------------------

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

If the array has less elements than required, it will adjust from the end accordingly.

Split the array in 4 parts:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 4)
print(newarr)
----------------------------------------------------------------------------

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

Access the splitted arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr[0])
print(newarr[1])
print(newarr[2])
----------------------------------------------------------------------------

>>> [1 2]
    [3 4]
    [5 6]
```

Splitting 2-D Arrays:

Use the same syntax when splitting 2-D arrays.

Use the `array_split()` method, pass in the array you want to split and the number of splits you want to do.

Split the 2-D array into three 2-D arrays.

```
import numpy as np
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
newarr = np.array_split(arr, 3)
print(newarr)
-----------------------------------------------------------------------------


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

NumPy Searching Arrays:

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the `where()` method.

Find the indexes where the value is 4:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)

--------------------------------------------------------------------------
>>> (array([3, 5, 6], dtype=int64),)
```

Which means that the value 4 is present at index 3, 5, and 6.

Find the indexes where the values are even:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 0)
print(x)

--------------------------------------------------------------------------
>>> (array([1, 3, 5, 7], dtype=int64),)
```

NumPy Sorting Arrays:

Sorting means putting elements in an ordered sequence.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

```
import numpy as np
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
--------------------------------------------------------------------------

>>> [0 1 2 3]
```

Note: This method returns a copy of the array, leaving the original array unchanged.

You can also sort arrays of strings, or any other data type:

```
import numpy as np
arr = np.array(['banana', 'cherry', 'apple'])
print(np.sort(arr))
--------------------------------------------------------------------------

>>> ['apple' 'banana' 'cherry']
```

Random Numbers in Numpy:

What is a Random Number?

Random number does NOT mean a different number every time. Random means something that can not be predicted logically.

Pseudo Random and True Random.

Computers work on programs, and programs are definitive set of instructions. So it means there must be some algorithm to generate a random number as well.

If there is a program to generate random number it can be predicted, thus it is not truly random.

Random numbers generated through a generation algorithm are called *pseudo random*.

Can we make truly random numbers?

Yes. In order to generate a truly random number on our computers we need to get the random data from some outside source. This outside source is generally our keystrokes, mouse movements, data on network etc.

We do not need truly random numbers, unless its related to security (e.g. encryption keys) or the basis of application is the randomness (e.g. Digital roulette wheels).

In this tutorial we will be using pseudo random numbers.

Generate Random Number

NumPy offers the `random` module to work with random numbers.

```
# Generate a random integer from 0 to 100:
from numpy import random
x = random.randint(100)
print(x)
--------------------------------------------------------------------------

>>> 77
```

Whenever you run this program , you get the different output between 1 to 100.

Generate Random Float

The random module's `rand()` method returns a random float between 0 and 1.

```
#Generate a random float from 0 to 1:
from numpy import random
x = random.rand()
print(x)
--------------------------------------------------------------------------

>>> 0.435340232345325
```

Generate Random Array

In NumPy we work with arrays, and you can use the two methods from the above examples to make random arrays.

Integers:

The `randint()` method takes a `size` parameter where you can specify the shape of an array.

```
# Generate a 1-D array containing 5 random integers from 0 to 100:
from numpy import random
x=random.randint(100, size=(5))
print(x)
--------------------------------------------------------------------------

>>> [45 66 17 78 94]
```

Generate a 2-D array with 3 rows, each row containing 5 random integers from 0 to 100:

```
from numpy import random
x = random.randint(100, size=(3, 5))
print(x)
--------------------------------------------------------------------------

>>> [[58 81 12 88 25]
     [43 19 54 62 93]
     [41 98 13 90  1]]
```

Floats:

The `rand()` method also allows you to specify the shape of the array.

```
#Generate a 1-D array containing 5 random floats:
from numpy import random
x = random.rand(5)
print(x)
```

```
----------------------------------------------------------------------------

>>> [0.08501005 0.45535222 0.86468357 0.25701995 0.66583101]
```

Generate a 2-D array with 3 rows, each row containing 5 random numbers:

```
from numpy import random
x = random.rand(3, 5)
print(x)
----------------------------------------------------------------------------

>>> [[0.55862062 0.66614591 0.56739866 0.96187613 0.1520532 ]
     [0.32030027 0.7348131  0.88085748 0.99225728 0.14550019]
     [0.69413463 0.27811927 0.46690703 0.87962331 0.13970273]]
```

Generate Random Number From Array

The `choice()` method allows you to generate a random value based on an array of values.

The `choice()` method takes an array as a parameter and randomly returns one of the values.

```
#Return one of the values in an array:
from numpy import random
x = random.choice([3, 5, 7, 9])
print(x)
----------------------------------------------------------------------------

>>> 3
```

The `choice()` method also allows you to return an *array* of values.

Add a `size` parameter to specify the shape of the array.

```
#Generate a 2-D array that consists of the values in the array parameter
(3, 5, 7, and 9):
from numpy import random
x = random.choice([3, 5, 7, 9], size=(3, 5))
print(x)
----------------------------------------------------------------------------

>>> [[7 3 9 5 9]
     [9 5 5 3 3]
     [5 7 3 9 5]]
```

Simple Arithmetic:

Firstly create the arrays for performing arithmetic operations.

```
import numpy as np

arr_1 = np.arange(1,10).reshape(3,3)
arr_2 = np.arange(1,10).reshape(3,3)
print(arr_1)
print(arr_2)
---------------------------------------------------------------------------

>>> [[1 2 3]
     [4 5 6]
     [7 8 9]]
     [[1 2 3]
     [4 5 6]
     [7 8 9]]
```

We will use above arrays for arithmetic operations

Addition:

```
#Method 1:
add_1 = arr_1 + arr_2
print(add_1)
---------------------------------------------------------------------------

>>> [[ 2  4  6]
     [ 8 10 12]
     [14 16 18]]



#Method 2:
add_2 = np.add(arr_1,arr_2)
print(add_2)
---------------------------------------------------------------------------

>>> [[ 2  4  6]
     [ 8 10 12]
     [14 16 18]]
```

Subtraction:

```
#Method 1:
sub_1 = arr_1 - arr_2
print(sub_1)
---------------------------------------------------------------------------

>>> [[0 0 0]
     [0 0 0]
     [0 0 0]]
```

```
# Method 2:
sub_2 = np.subtract(arr_1,arr_2)
Print(sub_2)
----------------------------------------------------------------------------

>>> [[0 0 0]
    [0 0 0]
    [0 0 0]]
```

Division:

```
#Method 1:
div_1 = arr_1 / arr_2
print(div_1)
----------------------------------------------------------------------------

>>> [[1. 1. 1.]
    [1. 1. 1.]
    [1. 1. 1.]]


#Method 2:
div_2 = np.divide(arr_1,arr_2)
print(div_2)
----------------------------------------------------------------------------

>>> [[1. 1. 1.]
    [1. 1. 1.]
    [1. 1. 1.]]
```

Multiplication:

```
#Method - 1
multi_1 = arr_1 * arr_2
print(multi_1)
----------------------------------------------------------------------------

>>> [[ 1  4  9]
    [16 25 36]
    [49 64 81]]

#Method - 2
multi_2 = np.multiply(arr_1, arr_2)
print(multi_2)
----------------------------------------------------------------------------

>>> [[ 1  4  9]
    [16 25 36]
    [49 64 81]]
```

Product:

```
#Method - 1
prod = arr_1 @ arr_2
print(prod)
----------------------------------------------------------------------------
```

```
>>> [[ 30  36  42]
     [ 66  81  96]
     [102 126 150]]

#Method - 2
prod_2 = arr_1.dot(arr_2)
print(prod_2)
----------------------------------------------------------------------------

>>> [[ 30  36  42]
     [ 66  81  96]
     [102 126 150]]
```

Power:

The `power()` function rises the values from the first array to the power of the values of the second array, and return the results in a new array.

Raise the values in arr1 to the power of values in arr2:

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 5, 6, 8, 2, 33])
newarr = np.power(arr1, arr2)
print(newarr)
----------------------------------------------------------------------------

>>> [       1000     3200000   729000000 -520093696        2500           0]
```

The example above will return [1000   3200000 729000000 -520093696 2500   0] which is the result of 10*10*10, 20*20*20*20*20, 30*30*30*30*30*30 etc.

Remainder:

Both the `mod()` and the `remainder()` functions return the remainder of the values in the first array corresponding to the values in the second array, and return the results in a new array.

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 7, 9, 8, 2, 33])
newarr = np.mod(arr1, arr2)
print(newarr)
----------------------------------------------------------------------------

>>> [ 1  6  3  0  0 27]
```

You get the same result when using the `remainder()` function.

Quotient and Mod:

The divmod() function return both the quotient and the the mod. The return value is two arrays, the first array contains the quotient and second array contains the mod.

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 7, 9, 8, 2, 33])
newarr = np.divmod(arr1, arr2)
print(newarr)

---------------------------------------------------------------------------


>>> (array([ 3,  2,  3,  5, 25,  1], dtype=int32), array([ 1,  6,  3,  0,
0, 27], dtype=int32))
```

Rounding Decimals:

Rounding Decimals:

There are primarily five ways of rounding off decimals in NumPy:

- truncation
- fix
- rounding
- floor
- ceil

Truncation:

Remove the decimals, and return the float number closest to zero. Use the `trunc()` and `fix()` functions. Sample example, using fix():

Truncate elements of following array:

```
import numpy as np
arr = np.trunc([-3.1666, 3.6667])
print(arr)
----------------------------------------------------------------------------

>>> [-3.  3.]
```

Sample example, using fix():

```
import numpy as np
arr = np.fix([-3.1666, 3.6667])
print(arr)
----------------------------------------------------------------------------

>>> [-3.  3.]
```

Rounding:

The `around()` function increments preceding digit or decimal by 1 if >=5 else do nothing.

E.g. round off to 1 decimal point, 3.16666 is 3.2.

Round off 3.1666 to 2 decimal places.

```
import numpy as np
arr = np.around(3.1666, 2)
print(arr)
```

```
-------------------------------------------------------------------------

>>> 3.17
```

Floor:

The floor() function rounds off decimal to nearest lower integer.

E.g. floor of 3.166 is 3.

```
import numpy as np
arr = np.floor([-3.1666, 3.6667])
print(arr)
-------------------------------------------------------------------------

>>> [-4.  3.]
```

Ceil:

The ceil() function rounds off decimal to nearest upper integer.

E.g. ceil of 3.166 is 4.

Ceil the elements of following array:

```
import numpy as np
arr = np.ceil([-3.1666, 3.6667])
print(arr)
-------------------------------------------------------------------------

>>> [-3.  4.]
```

NumPy Functions:

1) arange()

To create an array according to range.

2) linespace()

To create array with same distance between elements.

3) reshape()

To convert an 1D array into 2D or 3D.

4) ravel()

To convert multidimensional array into 1D.

5) flatten()

Act same as ravel() function, the only difference is you can provide argument to flatten function().

6) transpose() or T

To convert Row into column or vice versa.

7) max() *(array_name.max())*

To find the maximum value in the given array.

8) min() *(array_name.min())*

To find the minimum value in the given array.

9) max(axis = 0/1) *(array_name.max(axis = 0/1))*

To find the maximum value at every Row and Column
0 – Represent Column.
1 – Represent Row

10) min(axis = 0\1) *(array_name.min(axis = 0/1))*

11) argmax() *(array_name.argmax())*

To find the index value of Maximum Number.

12) argmin() *(array_name.argmin())*

To find the index value of minimum Number.

13) sum() *(np.sum(array_name))*

To sum of all the number in the given array.

14) sum(aris =0\1) *(np.sum(array_name, axis = 0\1))*

To sum of all the Column and Row.

15) mean() *(np.mean(array_name))*

To find the mean of the given array.

16) sqrt() *(np.sqrt(array_name))*

To find the square root of every element in the given array.

17) std() *(np.std(array_name))*

To find the standard division of given array.

18) log() *(np.log(array_name))*

To find the log of every element in the given array.

19) log10() *(np.log10(array_name))*

To find the log10 of every element in the given array.

20) shape() *(array_name.shape())*

Return a tuple consisting of array dimensions and can be used to resize the array.

21) ndim() *(array_name.ndim())*

Return the number of array dimensions.

22) size() *(array_name.size())*

Return the total number of elements in the array.

23) dtype() *(array_name.dtype())*

Return the data type of the array.

24) astype() *(array_name.astype(data_type))*

Changes the data type of an array.

25) full() *(np.full(parameterrs))*

Changes Array of fill value with the given shape, dtype and order.

26) transpose() *(np.transpose(array))*

Transpose is an operator which flips a matrix over its diagonal; that is, it switches the row and column indices of the matrix A by producing another matrix, often denoted by A^T

27) reshape() *(np.reshape(array,shape))*

Gives a new shape to an array without changing its data.

28) resize() *(np.resize(array,shape))*

Changes shape and size of array in-place.

29) flatten() *(np.flatten(array))*

Returns flattened 1D arrays.

30) insert() *(np.insert(arr, obj, values, axis))*

To insert items to an array.
arr: Input array.
obj: The index before which insertion is to be made.
values: The array of values to be inserted.

axis: The axis along insert in given array. If not given, arr is flattened.

31) append() *(np.append(arr, values, axis))*

To append items to an array.
arr: Array
values: To be appended to arr. It must be of the same shape as of arr (excluding axis of appending)
axis: The axis along which append operation is to done. If not given, both parameters are flattened.

32) delete() *(np.append(arr, obj, axis))*

To delete items from an array.
Arr: Input array
Obj: Can ba a slice, an integer or array of integers, indicating the subarray to be deleted from the input array.
Returns a copy of arr with the elements specified by obj removed. Note that delete does not occur in-place. If axis is none, out is a flattened array.

33) unique() *(np.unique(arr, return_index, return_inverse, return_counts))*

To get unique items from an array.
arr: Input array
return_index: If true, returns the indices of elements in the input array.
return_counts: If true, returns the number of times the element in unique array appears in the original array.

34) copy() *(array_name.copy())*

Make an copy of an array.

35) sort() *(array_name.sort(array_name, axis=1\0))*

Sorting an array.

36) random() *(np.random.random(shape))*

To get an array with random input elements of desired shape.

37) randit() *(np.random.randit(start,end,size))*

To get an array with random input integral elements of desired shape.

38) type() *(type(array_name))*

This built-in python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.