```
 # This is formatted as code
```

Experiment No : 01

Aim : Learn basics of Numpy library for storing and efficiently processing any external data into python execution pipeline.

Theory : NumPy (short for Numerical Python) provides an efficient interface to store and operate on dense data buffers. In some ways, *NumPy arrays* are like Python's built-in *list type*, but NumPy arrays provide much more *efficient storage* and *data operations* as the arrays grow larger in size. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python

NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

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. Arrays are very frequently used in data science, where speed and resources are very important.

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.

Working :

```
import numpy
numpy.__version__
```

```
'1.21.6'
```

A Python List Is More Than Just a List

Let's consider now what happens when we use a Python data structure that holds many Python objects. The standard mutable multi-element container in Python is the list. We can create a list of integers as follows:

```
L = list(range(10))
L
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
type(L[0])
```

```
int
```

```
L2 = [str(c) for c in L]
L2
```

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

```
type(L2[0])
```

```
str
```

```
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
```

```
[bool, str, float, int]
```

Fixed-Type Arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in `array` module (available since Python 3.3) can be used to create dense arrays of a uniform type:

```
import array
L = list(range(10))
A = array.array('i', L)
A
```

```
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Creating Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

```
import numpy as np
# integer array:
np.array([1, 4, 2, 5, 3])
```

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

Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

```
np.array([3.14, 4, 2, 3])
```

```
array([3.14, 4.  , 2.  , 3.  ])
```

If we want to explicitly set the data type of the resulting array, we can use the `dtype` keyword:

```
np.array([1, 2, 3, 4], dtype='float32')
```

```
array([1., 2., 3., 4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensional; here's one way of initializing a multidimensional array using a list of lists:

```
# nested lists result in multi-dimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

```
array([[2, 3, 4],
       [4, 5, 6],
       [6, 7, 8]])
```

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

```
# Create a length-10 integer array filled with zeros
np.zeros(10, dtype=int)
```

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

```
# Create a 3x5 floating-point array filled with ones
np.ones((3, 5), dtype=float)
```

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

```
# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
```

```
array([[3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

```
# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
```

```
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
# Create a 3x3 array of uniformly distributed
# random values between 0 and 1
np.random.random((3, 3))
```

```
array([[0.65279032, 0.63505887, 0.99529957],
       [0.58185033, 0.41436859, 0.4746975 ],
       [0.6235101 , 0.33800761, 0.67475232]])
```

```
# Create a 3x3 array of normally distributed random values
# with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
```

```
array([[ 1.0657892 , -0.69993739,  0.14407911],
       [ 0.3985421 ,  0.02686925,  1.05583713],
       [-0.07318342, -0.66572066, -0.04411241]])
```

```
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```

```
array([[7, 2, 9],
       [2, 3, 3],
       [2, 3, 4]])
```

```
# Create a 3x3 identity matrix
np.eye(5)
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

```
# Create an uninitialized array of three integers
# The values will be whatever happens to already exist at that memory location
np.empty(3)
```

```
array([1.5e-323, 2.0e-323, 2.5e-323])
```

NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users

of C, Fortran, and other related languages.

The standard NumPy data types are listed in the following table. Note that when constructing an array, they can be specified using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

Following table shows all datatypes for Numpy Array

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

NumPy Array Attributes

We will learn about important Attributes with NumPy Array objects

Each array object has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array), the `dtype`, the data type of the array :

```
#Consider following sample arrays
np.random.seed(0)  # seed for reproducibility
```

```
x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array

print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
print("dtype:", x3.dtype)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
dtype: int64
```

Other attributes include `itemsize`, which lists the size (in bytes) of each array element, and `nbytes`, which lists the total size (in bytes) of the array:

```
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

```
itemsize: 8 bytes
nbytes: 480 bytes
```

Array Indexing: Accessing Single Elements

Next we learn how to access single element in a NumPy array NumPy follows indexing similar to that of Python in a dimension index starts at 0 till length-1

So x1[0] will mean 0th element and x1[5] means sixth element in array x1.

We can use negative index value to indicate accessing elements from back side of array.

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices as shown in below code cell.

```
# acessing third list's first element
x2[2, 0]
```

```
1
```

```
# accessing second last element from second list
x2[1, -2]
```

```
8
```

```
# modifying value at a perticular index
x2[0, 0] = 12
x2
```

```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (`:`) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x`, use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0`, `stop=` *`size of dimension`*, `step=1`. We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

One-dimensional subarrays

```
x = np.arange(10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
x[:5]  # first five elements
```

```
array([0, 1, 2, 3, 4])
```

```
x[:5]  # elements after index five
```

```
array([0, 1, 2, 3, 4])
```

```
x[4:7]  # sub-array of index 4, 5, 6
```

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

```
x[::2]  # every other element
```

```
array([0, 2, 4, 6, 8])
```

```
x[2::2]  # every other element starting at index 2
```

```
array([2, 4, 6, 8])
```

A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

```
x[::-1]  # all elements, reversed
```

```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

```
x[3::-2]  # reversed every other from index 3
```

```
array([3, 1])
```

Multi-dimensional subarrays

Multi-dimensional slices work in the same way, with multiple slices separated by commas. For example:

```
x2[:2, :3]  # This is sub Array of x2 with first two rows and first three columns
```

```
array([[12,  5,  2],
       [ 7,  6,  8]])
```

```
#Check how we can reverse the multidimension array
x2[::-1, ::-1]
```

```
array([[ 7,  7,  6,  1],
       [ 8,  8,  6,  7],
       [ 4,  2,  5, 12]])
```

Accessing array rows and columns

One commonly needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (:):

```
print(x2[:, 0])  # first column of x2
```

```
[12  7  1]
```

```
print(x2[0, :])  # first row of x2
```

```
[12  5  2  4]
```

Subarrays as no-copy views

One important–and extremely useful–thing to know about array slices is that they return *views* rather than *copies* of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:

```
#Extract 2*2 sub array from x2
x2_sub = x2[:2, :2]
```

```
print(x2_sub)
```

```
[[12  5]
 [ 7  6]]
```

```
x2_sub[0, 0] = 99
#Above statement not oly modifies subarray but also the original array as well
print(x2_sub)
```

```
[[99  5]
 [ 7  6]]
```

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

```
import numpy as np

# Create a 4x6 array
my_array = np.array([[1, 2, 3, 4, 5, 6],
                     [7, 8, 9, 10, 11, 12],
                     [13, 14, 15, 16, 17, 18],
                     [19, 20, 21, 22, 23, 24]])

# Extract a 2x3 subarray from the left bottom corner
subObj = my_array[2:, :3]

print(subObj)
```

```
[[13 14 15]
 [19 20 21]]
```

Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

```
array([1, 2, 3, 3, 2, 1])
```

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


# concatenate along the first axis
np.concatenate([grid, grid])
```

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

```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

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

When joining arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions as shown below:

```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])

# vertically stack the arrays
np.vstack([x, grid])
```

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

```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

Notice that *N* split-points, leads to *N + 1* subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

```
grid = np.arange(16).reshape((4, 4))
grid
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Introducing UFuncs

For many types of operations, NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a *vectorized* operation. This can be accomplished by simply performing an operation on the array, which will then be applied to each element. This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

```
#Consider following loop based implementation to ind reciprocals for each element of an ar
np.random.seed(0)

def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
```

```
        output[i] = 1.0 / values[i]
    return output

big_array = np.random.randint(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
```

```
    2.56 s ± 478 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
#Same operatoin using UFuncs applying '/' over array elements
%timeit (1.0 / big_array)
```

```
    3.16 ms ± 431 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Vectorized operations in NumPy are implemented via *ufuncs*, whose main purpose is to quickly execute repeated operations on values in NumPy arrays. UFuncs are extremely flexible – before we saw an operation between a scalar and an array, but we can also operate between two arrays as well as multidimentional arrays.

Computations using vectorization through ufuncs are nearly always more efficient than their counterpart implemented using Python loops, especially as the arrays grow in size. Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.

Exploring NumPy's UFuncs

Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs.

Array arithmetic

NumPy's ufuncs feel very natural to use because they make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

```
x = np.arange(4)
print("x     =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2)  # floor division
#some advanced arithmatic expression   -(x/2+1)^2
-(0.5*x + 1) ** 2
```

```
    x     = [0 1 2 3]
    x + 5 = [5 6 7 8]
    x - 5 = [-5 -4 -3 -2]
    x * 2 = [0 2 4 6]
    x / 2 = [0.  0.5 1.  1.5]
```

```
x // 2 = [0 0 1 1]
array([-1.  , -2.25, -4.  , -6.25])
```

The following table lists the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Specialized ufuncs

NumPy has many more ufuncs available, including hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. A look through the NumPy documentation reveals a lot of interesting functionality.

Another excellent source for more specialized and obscure ufuncs is the submodule `scipy.special`. If you want to compute some obscure mathematical function on your data, chances are it is implemented in `scipy.special`. There are far too many functions to list them all, but the following snippet shows a couple that might come up in a statistics context:

```
#importing package special from scipy package
from scipy import special
```

```
# Gamma functions (generalized factorials) and related functions
x = [1, 5, 10]
print("gamma(x)     =", special.gamma(x))
print("ln|gamma(x)| =", special.gammaln(x))
print("beta(x, 2)   =", special.beta(x, 2))
```

```
gamma(x)     = [1.0000e+00 2.4000e+01 3.6288e+05]
ln|gamma(x)| = [ 0.          3.17805383 12.80182748]
beta(x, 2)   = [0.5        0.03333333 0.00909091]
```

Many other special functions like error functions, beta integral can also be evaluated.

Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we'd like to *reduce* an array with a particular operation, we can use the

`reduce` method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

```
x = np.arange(1, 6)
np.add.reduce(x)
```

```
15
```

```
np.multiply.reduce(x)
```

```
120
```

```
#note the difference in output with accumulate
np.add.accumulate(x)
```

```
array([ 1,  3,  6, 10, 15])
```

Q1. What are UFuncs un numpy?

Answer

UFuncs (Universal Functions) in NumPy are functions that operate on ndarrays in an element-wise manner, which means that the operation is applied to each element in the ndarrayindividually. UFuncs are designed to be fast, and can be used to perform a wide variety of mathematical operations on ndarrays, such as arithmetic, trigonometric, and logarithmicfunctions, among others. One of the key advantages of UFuncs is that they allow for vectorization of operations, whichmeans that the operations can be applied to entire arrays instead of having to loop over eachelement in the array individually. This can lead to significant performance improvements, particularly when working with large arrays. Some examples of UFuncs in NumPy include: 1.Arithmetic operations (addition, subtraction, multiplication, division, etc.) 2.Trigonometric functions (sine, cosine, tangent, etc.) 3.Exponential and logarithmic functions (exponential, logarithm, etc.) 4.Comparison operators (less than, greater than, equal to, etc.) UFuncs in NumPy are typically implemented in C and compiled with efficient low-level code, making them very fast and efficient. They can be accessed using the standard NumPy syntax, such as np.add(), np.sin(), np.exp(), etc

Q2. Which are various attributes of numy arrays object?

Answer

In NumPy, ndarray is the main object used to represent multi-dimensional arrays. It has manyattributes that can be used to get information about the array. Some of the commonly usedattributes of ndarray object are: 1.ndarray.shape: returns the dimensions of the array as a tuple. For a 1D array of lengthn, shape would be (n,). For a 2D array with n rows and m columns, shape would be (n, m), andsoon. 2.ndarray.ndim: returns the number of dimensions of the array. 3.ndarray.size: returns the total number of elements in the array. 4.ndarray.dtype: returns the data type of the array elements. 5.ndarray.itemsize: returns the size of each element in bytes. 6.ndarray.nbytes: returns the total number of bytes used by the array data. 7.ndarray.T: returns the transpose of the array. ndarray.real: returns the real part of the array (if it has complex elements). 8.ndarray.imag: returns the imaginary part of the array (if it has complex elements). 9.ndarray.flat: returns an iterator over the array elements in a one-dimensional form. 10.ndarray.flags: returns information about the memory layout and other attributes of thearray. These are just a few examples of the many attributes available in NumPy ndarray objects. Byaccessing these attributes, we can obtain various information about the array, such as itsshape, size, data type, and more.

Q3. If you have 3 dimentional array in numpy object Obj how to identify its size, type and dimentions ?

Answer

To identify the size, data type, and dimensions of a 3-dimensional NumPy array, you can usethe following attributes and methods: shape: This attribute returns a tuple representing the dimensions of the array. For a 3- dimensional array, the shape will be of the form (n, m, p), where n is the size of the first dimension, m is the size of the second dimension, and p is the size of the third dimension. For example, to get the shape of the array Obj, you can use Obj.shape. dtype: This attribute returns the data type of the array. For example, to get the data type of thearray Obj, you can use Obj.dtype. size: This method returns the total number of elements in the array. For example, to get thesize of the array Obj, you can use Obj.size. Here is an example of how you can use these attributes and methods:

```
import numpy as np
# create a 3-dimensional array
Obj = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
# get the shape of the array
print("Shape:", Obj.shape)
# get the data type of the array
print("Data type:", Obj.dtype)
# get the total number of elements in the array
print("Size:", Obj.size)
```

```
Shape: (2, 2, 2)
Data type: int64
```

```
Size: 8
```

Q4. Consider Obj object has dimentions 342 and you want to convert it to 46 *shape, explain how will you do it ? Show this using a code cell, taking an example of 3*4*2 np array of random int between 1 and 50.

Answer

To convert an array of shape (342,) to shape (46,), we need to reshape the array whileensuring that all the elements in the original array are preserved. We can do this using thereshape method of numpy. Here's an example code that demonstrates how to reshape a random integer numpy array of shape (34, 2) with values between 1 and 50 to a new shape of (46,):

```
import numpy as np
# Create a random integer numpy array of shape (34, 2) with values between 1 and 50
arr = np.random.randint(1, 50, size=(34, 2))
print("Original array of shape:", arr.shape)
print(arr)
# Reshape the array to shape (46,) while preserving all elements
new_shape = (46,)
if np.prod(new_shape) == np.prod(arr.shape):
  new_arr = arr.reshape(new_shape)
  print("Reshaped array of shape:", new_arr.shape)
  print(new_arr)
else:
  print("Cannot reshape array to new shape", new_shape, "while preserving all elements")
```

```
Original array of shape: (34, 2)
[[26 27]
 [ 6 27]
 [23 19]
 [36 29]
 [18  7]
 [ 8 28]
 [49 14]
 [ 3 40]
 [10 30]
 [12 29]
 [45 30]
 [14 41]
 [17  1]
 [14 20]
 [45 36]
 [30 12]
 [28 29]
 [22  6]
 [44 39]
 [33 44]
 [18 25]
 [ 7 42]
 [27 26]
```

```
 [24 12]
 [46  1]
 [42 44]
 [ 2 48]
 [15 20]
 [44  2]
 [47  2]
 [29 25]
 [42 45]
 [38 14]
 [48 45]]
Cannot reshape array to new shape (46,) while preserving all elements
```

Q5. Consider above 46 *array, sample* 23 sub array from left bottom of this array store the result in varible named subOb

```
import numpy as np
# Create a 4x6 array
my_array = np.array([[1, 2, 3, 4, 5, 6],
[7, 8, 9, 10, 11, 12],
[13, 14, 15, 16, 17, 18],
[19, 20, 21, 22, 23, 24]])
# Extract a 2x3 subarray from the left bottom corner
subObj = my_array[2:, :3]
print(subObj)
```

```
[[13 14 15]
 [19 20 21]]
```

Conclusion : Thus we have learned basics of Numpy library for storing and efficiently processing any external data into python execution pipeline.