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__
```

```
'2.0.2'
```

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([[0.86416675, 0.93056529, 0.26524861],
       [0.25207752, 0.29543641, 0.98891162],
       [0.49444042, 0.23405962, 0.37213307]])
```

```
# Create a 3x3 array of normally distributed random values
# with mean 0 and standard deviation 1
```

```
np.random.normal(0, 1, (3, 3))
```

```
array([[-0.94361272, -1.53677146,  0.65766021],
       [-0.57034607,  0.68799449,  1.1151934 ],
       [-0.14835688, -0.02469043, -1.67739515]])
```

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

```
array([[3, 6, 4],
       [6, 6, 6],
       [0, 3, 3]])
```

```
# 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([0. , 0.5, 1. ])
```

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]
```

```
np.int64(1)
```

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

```
np.int64(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:

```
x2_sub_copy = x2_sub.copy()
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
print(x2)
```

```
[[42  5]
 [ 7  6]]
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

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)
```

```
[[9 8 7]
 [6 5 4]]
[]
```

```
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 array
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)
```

```
1.72 s ± 204 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)
```

```
1.08 ms ± 126 µs per loop (mean ± std. dev. of 7 runs, 1000 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])
```

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)

Operator	Equivalent ufunc	Description
`-`	`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
```

```
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)
```

```
np.int64(15)
```

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

```
np.int64(120)
```

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

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

Answer Following Questions :

1)What are UFuncs in numpy?

Ans:-

A NumPy ufunc:

Works element-wise on arrays

Is implemented in C (so it’s very fast)

Supports broadcasting

Can operate on scalars and arrays

Often has an equivalent Python operator

Example:

import numpy as np

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

np.add(a, b)

Output: array([5, 7, 9])

This is the same as:

a + b

Common UFunc Categories

1. Arithmetic UFuncs np.add(a, b) np.subtract(a, b) np.multiply(a, b) np.divide(a, b) np.power(a, b)
2. Trigonometric UFuncs np.sin(a) np.cos(a) np.tan(a)
3. Comparison UFuncs np.greater(a, b) np.equal(a, b)
4. Logical UFuncs np.logical_and(a, b) np.logical_or(a, b)
5. Mathematical UFuncs np.sqrt(a) np.exp(a) np.log(a)

2)Which are various attributes of numy arrays object?

Ans:-

1. ndim

Number of dimensions (axes)

a = np.array([[1, 2], [3, 4]]) a.ndim

➞ 2

2. shape

Size of the array in each dimension

a.shape

➞ (2, 2)

3. size

Total number of elements

a.size

➞ 4

4. dtype

Data type of elements

a.dtype

➞ int64 (or int32, depends on system)

5. itemsize

Memory size (in bytes) of each element

a.itemsize

➞ 8 (for int64)

6. nbytes

Total memory consumed by the array

a.nbytes

➞ 32 bytes

7. T

Transpose of the array

a.T

8. strides

Steps (in bytes) to move to the next element in each dimension a.strides

(Used mostly for advanced memory handling)

9. flags

Information about memory layout

a.flags

Shows whether the array is:

C-contiguous

Writable

Owns its data, etc.

10. real and imag

(For complex arrays)

c = np.array([1+2j, 3+4j]) c.real c.imag

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

Ans:-

Assume a 3D array import numpy as np

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

1. Dimensions (Number of axes)

Use ndim

obj.ndim

➡ 3 Confirms it is a 3-dimensional array

2. Shape (Size along each dimension)

Use shape

obj.shape

➡ (2, 2, 2) Means:

2 blocks

each block has 2 rows

each row has 2 columns

3. Total Size (Number of elements)

Use size

obj.size

➡ 8 Total elements = 2 × 2 × 2

4. Data Type (Type of elements)

Use dtype

obj.dtype

➡ int64 (or system-dependent)

4)Consider array object named Obj with dimentions 3*4*2 and you want to convert it to 4*6 shape exlpain 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.

Ans:-

Original array Obj has shape (3, 4, 2)

Total elements = 3 × 4 × 2 = 24

New shape (4, 6) also needs 4 × 6 = 24 elements

Since the total number of elements stays the same, NumPy allows reshaping.

We use the reshape() method.

Code cell (NumPy example) import numpy as np

Create a 3×4×2 array with random integers between 1 and 50 Obj = np.random.randint(1, 51, size=(3, 4, 2))

print("Original array:") print(Obj) print("Original shape:", Obj.shape)

Reshape to 4×6 Obj_reshaped = Obj.reshape(4, 6)

print("\nReshaped array:") print(Obj_reshaped) print("New shape:", Obj_reshaped.shape)

Explanation of the code

np.random.randint(1, 51, size=(3, 4, 2)) → Creates a 3D array with values from 1 to 50

Obj.shape → Confirms the original shape (3, 4, 2)

Obj.reshape(4, 6) → Rearranges the same 24 elements into a 2D array of shape (4, 6)

5)Consider above 4*6 array, sample 2*3 sub array from left bottom of this array store the result in varible named subObjCpy.

Ans:-

Left columns → first 3 columns → :3

Bottom rows → last 2 rows → -2:

Use .copy() to store it as a separate array (not just a view)

Code cell import numpy as np

Example 4×6 array (from previous reshape) Obj_reshaped = np.array([[10, 12, 15, 18, 20, 22], [25, 27, 30, 32, 35, 37], [40, 42, 45, 47, 48, 49], [5, 7, 9, 11, 13, 14]])

print("Original 4×6 array:") print(Obj_reshaped)

Extract 2×3 sub-array from left bottom subObjCpy = Obj_reshaped[-2:, :3].copy()

print("\n2×3 sub-array from left bottom:") print(subObjCpy) print("Shape of subObjCpy:", subObjCpy.shape)

Obj_reshaped[-2:, :3]

-2: → selects last 2 rows

:3 → selects first 3 columns

.copy() → creates an independent array

6)Make changes in 1 perticular value in the subObjCpy. Check values of Obj subObjCpy to check impact of this change of element in original and copied object. What do you conclude?

Ans:-

Code cell (experiment) import numpy as np

Original 4×6 array Obj = np.array([[10, 12, 15, 18, 20, 22], [25, 27, 30, 32, 35, 37], [40, 42, 45, 47, 48, 49], [5, 7, 9, 11, 13, 14]]) Create copied sub-array from left bottom subObjCpy = Obj[-2:, :3].copy()

print("Original Obj:") print(Obj)

print("\nCopied subObjCpy before change:") print(subObjCpy)

Modify one element in subObjCpy subObjCpy[0, 0] = 999

print("\nCopied subObjCpy after change:") print(subObjCpy)

print("\nOriginal Obj after change in subObjCpy:") print(Obj)

7)If you were to filter sub array view from above Obj object and want to apply modification in original and/or sub obj change respective element in other how will you achieve it?

Ans:-

Step 1: Create a sub-array view import numpy as np

Original 4×6 array Obj = np.array([[10, 12, 15, 18, 20, 22], [25, 27, 30, 32, 35, 37], [40, 42, 45, 47, 48, 49], [5, 7, 9, 11, 13, 14]])

Create a sub-array view (no .copy()) subObjView = Obj[-2:, :3] # Last 2 rows, first 3 columns

print("Original Obj:") print(Obj)

print("\nSub-array view:") print(subObjView)

Step 2: Modify an element in the sub-array Change an element in the sub-array view subObjView[0, 0] = 999

print("\nSub-array view after modification:") print(subObjView)

print("\nOriginal Obj after modifying the view:") print(Obj)

Step 3: Modify an element in the original array Change an element in the original array Obj[-1, 2] = 555

print("\nOriginal Obj after second modification:") print(Obj)

print("\nSub-array view reflects changes in original:") print(subObjView)