Name : Shantanu R Chougule **Roll No. : 71** **Batch : B3**

Experiment No. 2

Title: Data Manipulation (Numpy library) Operations Broadcasting Indexing and slicing

Aim: Data Manipulation (Numpy library) Operations Broadcasting Indexing and slicing

Theory:

Numpy

What is NumPy?

NumPy (Numerical Python) is a powerful library in Python that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. It is a fundamental package for scientific computing in Python.

Properties of NumPy

1. **N-dimensional Arrays:** Efficient multi-dimensional array operations.
2. **Mathematical Functions:** Wide range of functions for computation.
3. **Broadcasting:** Flexible arithmetic on differently-shaped arrays.

Applications of NumPy

1. **Data Analysis:** Manipulating and analyzing datasets.
2. **Scientific Computing:** Large-scale computations in various fields.
3. **Machine Learning:** Preprocessing data and building models.

Import NumPy Library

In [4]:
```
import numpy as np
```

Create Array from Python List

NumPy arrays can be created from Python lists using the `np.array()` function. This conversion allows the list to be used in array operations, which are more efficient for numerical computations compared to standard Python lists.

In [5]:
```
#Create array from Python list
arr_1=np.array([2,3,4,5,6,7,8,9])
print(arr_1)
```

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

Create Array of Float Data Type

You can specify the data type of a NumPy array during its creation using the `dtype` parameter. Setting `dtype=float` ensures that all elements in the array are treated as floating-point numbers, facilitating precise numerical operations.

In [6]:
```
#create array of float data type
arr_2=np.array([2.4,3.4,6.7,8.9])
print(arr_2)
```

```
[2.4 3.4 6.7 8.9]
```

Convert Array into Data Type Float

The `astype()` method is used to convert an existing NumPy array to a different data type, such as float. This method returns a new array with the desired data type while leaving the original array unchanged.

In [7]:
```
#convert array into data type float
arr_3=np.array([2,3,4,5,6,7,8,9],dtype='float32')
arr_3.dtype
```

Out[7]:
```
dtype('float32')
```

Create Array by Using For Loop

Arrays can be created by iterating through a range or sequence with a for loop and appending values to a list, which is then converted to a NumPy array. This approach is useful for generating arrays with custom values based on computation.

In [8]:
```
#create array by using for loop
np.array([range(i, i+3) for i in [2,4,5,6]])
```

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

Create Length of 10 Integers Filled with Zero

The `np.zeros()` function creates an array of a specified length filled with zeros. By setting `dtype=int` , you ensure that the array is composed of integer values, suitable for initialization purposes.

In [9]:
```
#create a length of 10 integers filled with zero
np.zeros(10, dtype='int')
```

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

Create 3 by 5 Array Filled with 3.14

Use the `np.full()` function to create a 3x5 array where every element is filled with a specific value, such as 3.14. This function allows for the creation of arrays with a constant value across all elements.

In [10]:
```
#create 3 by 5 array filled with 3.14
np.full((3,5),3.14)
```

Out[10]:
```
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 Array Filled with Linear Sequence Starting from 0 Ending at 20 Stepping with 2

The `np.arange()` function generates an array with a sequence of values. By specifying the start, stop, and step parameters, you can create an array with values ranging from 0 to 20 with a step size of 2.

In [11]:
```
#create an array filled with linear sequence starting from 0 ending at 20 s
np.arange(0,21,2)
```

Out[11]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20])
```

Create Array Evenly Spread Between 0 to 1

`np.linspace()` is used to create an array with a specified number of evenly spaced values between two endpoints, such as 0 and 1. This is useful for generating arrays with values distributed over a range.

In [12]:
```
#create an array evenly spread between 0 to 1
np.linspace(0,1,5)
```

Out[12]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Create 3 by 3 Identity Matrix

An identity matrix is a square matrix with ones on the diagonal and zeros elsewhere. The `np.eye()` function generates such a matrix of a specified size, which is often used in linear algebra operations.

In [13]:
```
#create 3 by 3 identity matrix
np.eye(3)
```

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

Numpy array attributes

In [14]:
```
#generate random matrix
np.random.seed(0)
```

Generate Random 1D Matrix

A one-dimensional (1D) random matrix is essentially a vector with random elements. The `np.random.rand()` function in NumPy generates an array of specified size with random values uniformly distributed between 0 and 1. These matrices are useful for simulations, random sampling, and other statistical applications.

In [15]:
```
#1Dimention
x1=np.random.randint(10, size=6)
x1
```

Out[15]:
```
array([5, 0, 3, 3, 7, 9])
```

Generate Random 2D Matrix

A two-dimensional (2D) random matrix is an array with rows and columns filled with random values. The `np.random.rand(m, n)` function creates an `m x n` matrix with random values uniformly distributed between 0 and 1. Such matrices are widely used in data science for creating synthetic datasets and testing algorithms.

In [16]:
```
#2Dimentions
x2=np.random.randint(10, size=(3,5))
x2
```

Out[16]:
```
array([[3, 5, 2, 4, 7],
       [6, 8, 8, 1, 6],
       [7, 7, 8, 1, 5]])
```

Generate Random 3D Matrix

A three-dimensional (3D) random matrix is an array with three dimensions filled with random values. The `np.random.rand(d1, d2, d3)` function generates a `d1 x d2 x d3` matrix with random values uniformly distributed between 0 and 1. These matrices are used in simulations, 3D modeling, and machine learning applications.

In [17]:
```
#3Dimentions
x3=np.random.randint(10, size=(3,4,5))
x3
```

Out[17]:
```
array([[[9, 8, 9, 4, 3],
        [0, 3, 5, 0, 2],
        [3, 8, 1, 3, 3],
        [3, 7, 0, 1, 9]],

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

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

Print Shape and Size of 3D Matrix

The shape and size of a 3D matrix can be obtained using the `shape` and `size` attributes of a NumPy array. The `shape` attribute returns the dimensions of the matrix, while the `size` attribute returns the total number of elements in the matrix. These attributes are useful for understanding the structure of the matrix and for debugging purposes.

In [18]:
```
#print x3 dimention, shape and size
dim=x3.ndim
shape=x3.shape
size=x3.size
print(f"Dimention of x3 is {dim}")
```

```
Dimention of x3 is 3
```

In [19]:
```
x3.shape
```

Out[19]:
```
(3, 4, 5)
```

In [20]:
```
x3.size
```

Out[20]:
```
60
```

Array Indexing

Array indexing involves accessing specific elements in an array using their indices. In NumPy, you can use integer indices to retrieve elements from 1D, 2D, and 3D arrays. For example, `array[i]` accesses the `i-th` element in a 1D array, `array[i, j]` retrieves the element at the `i-th` row and `j-th` column in a 2D array, and `array[i, j, k]` fetches the element in the `i-th` depth, `j-th` row, and `k-th` column in a 3D array.

Indexing is crucial for extracting and manipulating data efficiently.

Access First and Last Element

Accessing elements in a matrix can be done using indexing. To access the first element of a 1D matrix, use `array[0]` . To access the last element, use `array[-1]` . This allows quick retrieval of elements from the start or end of the matrix, which is often needed in various computations and analyses.

In [21]:
```
#access first and last element
first=x1[0]
last=x1[-1]
print(f"First element is {first} and Last element is {last}")
```

```
First element is 5 and Last element is 9
```

Access Second Last Element

To access the second last element of a 1D matrix, use the index `array[-2]` . This indexing method helps in quickly accessing elements from the end of the array, which is useful in many algorithms where the focus is on the tail elements of the data.

In [22]:
```
#access second last element
x1[-2]
```

Out[22]:
```
7
```

Access Any Element from 2D Matrix

In a two-dimensional (2D) matrix, elements can be accessed using a pair of indices. For example, `array[i, j]` accesses the element in the `i-th` row and `j-th` column. This indexing is essential for manipulating and analyzing specific entries within a matrix.

In [60]:
```
#access any element from x2
x2[1,2]
```

Out[60]: 8

Modify the Value in 2D Matrix

Modifying the value of an element in a 2D matrix is straightforward. By using the index `array[i, j]` , you can assign a new value to the specified element. For instance, `array[i, j] = new_value` updates the element at the `i-th` row and `j-th` column. This capability is crucial for updating data and conducting iterative algorithms.

In [62]:
```
#modify the value from x2
x2[0,0]=12
print(f'Modify to value {x2[0,0]}')
```

```
Modify to value 12
```

Array Slicing

Array slicing allows you to extract a subset of elements from an array. In NumPy, you can use the slice notation `array[start:stop:step]` to select elements from an array. This is useful for data manipulation, such as selecting rows, columns, or specific ranges of elements in a matrix. For example, `array[1:4]` selects elements from index 1 to 3 (excluding index 4) in a 1D array.

In [65]:
```
x=np.arange(20)
x
```

Out[65]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

In [67]:
```
#extract 1st five elements
x[0:5]
```

Out[67]:
```
array([0, 1, 2, 3, 4])
```

In [69]:
```
#extract elements after index 5
x[5:10]
```

Out[69]:
```
array([5, 6, 7, 8, 9])
```

In [71]:
```
#extract middle array 4 to 7
x[5:7]
```

Out[71]:
```
array([5, 6])
```

In [73]:
```
#display every other element
x[0:19:2]
```

Out[73]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [75]:
```
#display every other element starting at index 1
x[1:19:2]
```

Out[75]:
```
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17])
```

In [77]:
```
#display all element reverse
x[::-1]
```

Out[77]:
```
array([19, 18, 17, 16, 15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,
        2,  1,  0])
```

In [79]:
```
#diplay reverse everyother from index 5
x[:4:-1]
```

Out[79]:
```
array([19, 18, 17, 16, 15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5])
```

In [81]:
```
#2D display every other element
x2[0:19:2]
```

Out[81]:
```
array([[12,  5,  2,  4,  7],
       [ 7,  7,  8,  1,  5]])
```

In [83]:
```
read=np.arange(1,10).reshape(3,3)
read
#reshape
```

Out[83]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Array Concatenate

Array concatenation involves combining two or more arrays into a single array. In NumPy, you can use functions like `np.concatenate()` to join arrays along a specified axis. For example, `np.concatenate((array1, array2), axis=0)` stacks `array1` and `array2` vertically, while `axis=1` stacks them horizontally. This operation is useful for merging datasets or aggregating results from multiple sources.

In [86]:
```
np.concatenate([x,x1])
```

Out[86]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19,  5,  0,  3,  3,  7,  9])
```

Universal Functions & Aggregation Functions on Dataset

In [151]:
```
import pandas as pd
data=pd.read_csv("president_heights.csv")
```

In [153]:
```
data.head()
```

Out[153]:

	order	name	height(cm)
0	1	George Washington	189
1	2	John Adams	170
2	3	Thomas Jefferson	189
3	4	James Madison	163
4	5	James Monroe	183

Universal Functions on Dataset

Universal functions (ufuncs) are functions that operate element-wise on arrays. In NumPy, ufuncs allow you to perform operations like addition, subtraction, multiplication, and division efficiently. Functions such as `np.add()` , `np.subtract()` , `np.multiply()` , and `np.divide()` perform these arithmetic operations. Additionally, `np.floor_divide()` performs integer division, `np.power()` raises elements to a power, and `np.mod()` computes the modulus. Ufuncs enable fast, vectorized computations on arrays without the need for explicit loops

In [156]:
```
value_to_add = 5
data['height(cm)'] = np.add(data['height(cm)'], value_to_add)
```

In [158]:
```
data.head()
```

Out[158]:

	order	name	height(cm)
0	1	George Washington	194
1	2	John Adams	175
2	3	Thomas Jefferson	194
3	4	James Madison	168
4	5	James Monroe	188

In [160]:
```
value_to_subtract = 5
data['height(cm)'] = np.subtract(data['height(cm)'], value_to_subtract)
```

In [162]:
```
data.head()
```

Out[162]:

	order	name	height(cm)
0	1	George Washington	189
1	2	John Adams	170
2	3	Thomas Jefferson	189

3 4 James Madison 163
4 5 James Monroe 183

In [164]:
```
value_to_multiply = 2
data['height(cm)'] = np.multiply(data['height(cm)'], value_to_multiply)
```

In [166]:
```
data.head()
```

Out[166]:

	order	name	height(cm)
0	1	George Washington	378
1	2	John Adams	340
2	3	Thomas Jefferson	378
3	4	James Madison	326
4	5	James Monroe	366

In [168]:
```
value_to_divide = 2
data['height(cm)'] = (np.divide(data['height(cm)'], value_to_divide)).astyp
```

In [170]:
```
data.head()
```

Out[170]:

	order	name	height(cm)
0	1	George Washington	189
1	2	John Adams	170
2	3	Thomas Jefferson	189
3	4	James Madison	163
4	5	James Monroe	183

In [172]:
```
divisor = 10
data['height_floor_divide'] = np.floor_divide(data['height(cm)'], divisor)
```

In [174]:
```
data.head()
```

Out[174]:

	order	name	height(cm)	height_floor_divide
0	1	George Washington	189	18
1	2	John Adams	170	17
2	3	Thomas Jefferson	189	18
3	4	James Madison	163	16
4	5	James Monroe	183	18

In [176]:
```
power = 2
data['height_power'] = np.power(data['height(cm)'], power)
```

In [178]:
```
data.head()
```

Out[178]:

	order	name	height(cm)	height_floor_divide	height_power
0	1	George Washington	189	18	35721
1	2	John Adams	170	17	28900
2	3	Thomas Jefferson	189	18	35721
3	4	James Madison	163	16	26569
4	5	James Monroe	183	18	33489

In [180]:
```
divisor_mod = 7
data['height_mod'] = np.mod(data['height(cm)'], divisor_mod)
```

In [182]:
```
data.head()
```

Out[182]:

	order	name	height(cm)	height_floor_divide	height_power	height_mod
0	1	George Washington	189	18	35721	0
1	2	John Adams	170	17	28900	2
2	3	Thomas Jefferson	189	18	35721	0
3	4	James Madison	163	16	26569	2
4	5	James Monroe	183	18	33489	1

Aggregation Functions on Dataset

Aggregation functions summarize data by performing operations such as sum, mean, or standard deviation. In NumPy, functions like `np.sum()` , `np.mean()` , and `np.std()` are used to calculate the total, average, and variability of data, respectively. Additionally, functions such as `np.min()` and `np.max()` find the minimum and maximum values, `np.prod()` calculates the product of elements, `np.median()` finds the median, `np.percentile()` calculates percentiles, and `np.all()` and `np.any()` test whether all or any elements satisfy a condition. These functions are essential for data analysis, providing insights into the overall characteristics of a dataset.

In [189]:
```
min_height = np.min(data['height(cm)'])
print("Minimum Height:", min_height)
```

```
Minimum Height: 163
```

In [191]:
```
max_height = np.max(data['height(cm)'])
print("Maximum Height:", max_height)
```

```
Maximum Height: 193
```

In [193]:
```
total_height = np.sum(data['height(cm)'])
print("Sum of Heights:", total_height)
```

```
Sum of Heights: 7549
```

In [195]:
```
product_height = np.prod(data['height(cm)'])
print("Product of Heights:", product_height)
```

```
Product of Heights: -570425344
```

In [197]:
```
median_height =  np . median ( data [ 'height(cm)' ])
print( "Median Height:" ,  median_height )
```

```
Median Height: 182.0
```

In [199]:
```
percentile_25 = np.percentile(data['height(cm)'], 25)
percentile_75 = np.percentile(data['height(cm)'], 75)
print("25th Percentile Height:", percentile_25)
print("75th Percentile Height:", percentile_75)
```

```
25th Percentile Height: 174.25
75th Percentile Height: 183.0
```

In [201]:
```
all_heights_above_150 = np.all(data['height(cm)'] > 150)
print("All heights above 150 cm:", all_heights_above_150)
```

```
All heights above 150 cm: True
```

In [203]:
```
any_height_below_160 = np.any(data['height(cm)'] < 160)
print("Any height below 160 cm:", any_height_below_160)
```

```
Any height below 160 cm: False
```