NAME : DHEVISHREE GN

In [ ]:
```
REG NUM : 22MID0136
```

In [ ]:
```
LAB 13
```

In [2]:
```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]]) # Creating a 2-D Array
print(arr)
```

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

In [3]:
```
 print(arr.size) # Displays the number of elements in a array
```

```
6
```

In [4]:
```
 print(arr.shape) # Display the rows and columns
```

```
(2, 3)
```

In [5]:
```
 print(arr.ndim) # Displays the dimension of array
```

```
2
```

In [6]:
```
 # Create a 1D array
 arr = np.array([1, 2, 3, 4, 5, 6])
 print("Original array:", arr)
 print("Original shape:", arr.shape)
```

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

In [8]:
```
# Reshape to a 2D array (2 rows, 3 columns)
reshaped_arr = arr.reshape(2, 3) # Reshape a 1-D array to 2-D array
print(reshaped_arr)
print("New shape:", reshaped_arr.shape)
```

```
[[1 2 3]
 [4 5 6]]
New shape: (2, 3)
```

In [9]:
```
 import numpy as np
 def numpysum(n):
  a = np.arange(n) ** 2  #Creates an array [0,1,..n-1] and element-wise sqauring
  b = np.arange(n) ** 3
  c = a + b #element wise adding - NO LOOP Overhead, hence Memory Efficient
  return c
 numpysum(10) # function call
```

Out[9]:
```
array([  0,   2,  12,  36,  80, 150, 252, 392, 576, 810])
```

In [10]:
```
 #ARRAY CREATION FUNCTIONS
 import numpy as np
 np.arange(10) # 0 TO BEFORE 10, Step Value 1
```

Out[10]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [11]:
```
np.arange(2, 10, dtype=float) # 2. to Before 10., step value 1
```

Out[11]:
```
array([2., 3., 4., 5., 6., 7., 8., 9.])
```

In [12]:
```
 np.arange(2, 3, 0.1) # 2 to Before 3, step value 0.1
```

Out[12]:
```
array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
```

In [13]:
```
 #Creating Arrays from Sequences
 #a list of numbers will create a 1D array,
 a1D = np.array([1, 2, 3, 4])
 print(a1D)
```

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

In [18]:
```
#a list of lists will create a 2D array
a2D = np.array([[1, 2], [3, 4]])
print(a2D)
```

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

In [19]:
```
 #further nested lists will create higher-dimensional arrays.
#In general, any array object is called an ndarray in NumPy.
a3D = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

```
print(a3D)
```

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

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

In [20]:
```
import numpy as np
np.array([127, 128, 129], dtype=np.int32)
```

Out[20]:
```
array([127, 128, 129], dtype=int32)
```

In [22]:
```
import numpy as np
np.array([127, 128, 129], dtype=np.int8)
# Error: OverflowError because the np.int8 data type cannot hold the values 128 and 129
```

```
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
Cell In[22], line 2
      1 import numpy as np
----> 2 np.array([127, 128, 129], dtype=np.int8)

OverflowError: Python integer 128 out of bounds for int8
```

In [23]:
```
#2D Array Creation functions
#properties of special matrices represented as 2D arrays.
np.eye(3) # Identity matrice of 3X3
```

Out[23]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [24]:
```
np.eye(3, 5)   # Identity matrice of 3X5
```

Out[24]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

In [25]:
```
np.eye(3, 5, k=-2) #optional k argument
# When k is negative (e.g., k=-2), the diagonal shifts below the main diagonal
# when k is positive (e.g., k=1), the diagonal shifts above the main diagonal.
```

Out[25]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.]])
```

In [26]:
```
# It returns an array containing the elements of the specified diagonal.
np.diag([1, 2, 3])
```

Out[26]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

In [27]:
```
# To hold this upper diagonal, NumPy creates a (n + k) × (n + k) square matrix, where n is the length of the i
# So for [1, 2, 3]: Length = 3, k = 1, Matrix shape = 4 × 4
# Diagonal values are placed at positions: (0,1), (1,2), (2,3)
np.diag([1, 2, 3], 1)
```

Out[27]:
```
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])
```

In [28]:
```
# numpy.diag can define either a square 2D array with given values along the diagonal or if given a 2D array
# It returns a 1D array that is only the diagonal elements.
a = np.array([[1, 2], [3, 4]])
np.diag(a)
```

Out[28]:
```
array([1, 4])
```

In [29]:
```
#np.linspace(0, 2, 5) → [0. , 0.5, 1. , 1.5, 2. ]
# N = 2 → two columns: x**1, x**0
np.vander(np.linspace(0, 2, 5), 2)
```

Out[29]:
```
array([[0. , 1. ],
       [0.5, 1. ],
       [1. , 1. ],
       [1.5, 1. ],
       [2. , 1. ]])
```

In [30]:
```
#Creating a 1D array of size 5 filled with ones
arr1 = np.ones(5)  # shape=5 (1D), default dtype=float
print("1D Array of Ones:", arr1)
```

```
1D Array of Ones: [1. 1. 1. 1. 1.]
```

In [31]:
```
#Creating a 2D array (3 rows, 4 columns) filled with ones
arr2 = np.ones((3, 4))  # shape=(3,4) means 3x4 matrix
print("\n2D Array (3x4) of Ones:\n", arr2)
```

```
2D Array (3x4) of Ones:
 [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

In [32]:
```
#Specifying data type as integer
arr3 = np.ones((2, 3), dtype=int)  # shape=(2,3), dtype=int
print("\n2D Integer Array (2x3) of Ones:\n", arr3)
```

```
2D Integer Array (2x3) of Ones:
 [[1 1 1]
 [1 1 1]]
```

In [33]:
```
#Creating a 3D array filled with ones
arr4 = np.ones((2, 2, 3))  # shape=(2,2,3) means 3D array (2 blocks, 2 rows, 3 columns)
print("\n3D Array of Ones:\n", arr4)
```

```
3D Array of Ones:
 [[[1. 1. 1.]
  [1. 1. 1.]]

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

In [34]:
```
#Creating a boolean array (True represents 1)
arr5 = np.ones((3, 3), dtype=bool)
print("\nBoolean Array (3x3) of Ones (True values):\n", arr5)
```

```
Boolean Array (3x3) of Ones (True values):
 [[ True  True  True]
 [ True  True  True]
 [ True  True  True]]
```

In [36]:
```
#Creating a 1D array of size 5 filled with zeros
arr1 = np.zeros(5)  # shape=5 (1D), default dtype=float
print("1D Array of Zeros:", arr1)
```

```
1D Array of Zeros: [0. 0. 0. 0. 0.]
```

In [37]:
```
#Creating a 2D array (3 rows, 4 columns) filled with zeros
arr2 = np.zeros((3, 4))  # shape=(3,4) means 3x4 matrix
print("\n2D Array (3x4) of Zeros:\n", arr2)
```

```
2D Array (3x4) of Zeros:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

In [38]:
```
#Specifying data type as integer
arr3 = np.zeros((2, 3), dtype=int)  # shape=(2,3), dtype=int
print("\n2D Integer Array (2x3) of Zeros:\n", arr3)
```

```
2D Integer Array (2x3) of Zeros:
 [[0 0 0]
 [0 0 0]]
```

In [39]:
```
#Creating a 3D array filled with zeros
arr4 = np.zeros((2, 2, 3))  # shape=(2,2,3) means 3D array (2 blocks, 2 rows, 3 columns)
print("\n3D Array of Zeros:\n", arr4)
```

```
3D Array of Zeros:
 [[[0. 0. 0.]
  [0. 0. 0.]]

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

In [40]:
```
#Creating a boolean array (False represents 0)
arr5 = np.zeros((3, 3), dtype=bool)
print("\nBoolean Array (3x3) of Zeros (False values):\n", arr5)
```

```
Boolean Array (3x3) of Zeros (False values):
 [[False False False]
 [False False False]
 [False False False]]
```

In [41]:
```
#Creating indices for a 2D array of shape (3, 3)
indices_2d = np.indices((3, 3))  # shape=(3,3) means 3 rows and 3 columns
print("Indices for a 3x3 grid:\n", indices_2d)
```

```
Indices for a 3x3 grid:
 [[[0 0 0]
  [1 1 1]
  [2 2 2]]

 [[0 1 2]
  [0 1 2]
  [0 1 2]]]
```

In [42]:
```
# indices_2d is actually a 3D array of shape (2, 3, 3):
# - The first 3x3 array contains row indices
# - The second 3x3 array contains column indices
```

In [43]:
```
print("\nRow indices (first array):\n", indices_2d[0])  # Row positions
```

```
Row indices (first array):
 [[0 0 0]
 [1 1 1]
 [2 2 2]]
```

In [44]:
```
print("\nColumn indices (second array):\n", indices_2d[1])  # Column positions
```

```
Column indices (second array):
 [[0 1 2]
 [0 1 2]
 [0 1 2]]
```

In [46]:
```
#Creating indices for a 2x4 array
indices_2x4 = np.indices((2, 4))  # shape=(2,4) means 2 rows, 4 columns
```

In [47]:
```
print("\nIndices for a 2x4 grid:\n", indices_2x4)
```

```
Indices for a 2x4 grid:
 [[[0 0 0 0]
  [1 1 1 1]]

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

In [48]:
```
print("\nRow indices:\n", indices_2x4[0])
```

```
Row indices:
 [[0 0 0 0]
 [1 1 1 1]]
```

In [49]:
```
print("\nColumn indices:\n", indices_2x4[1])
```

```
Column indices:
 [[0 1 2 3]
 [0 1 2 3]]
```

In [50]:
```
#Generate a single random float between 0 and 1
random_float = np.random.random()
print("Random Float (0 to 1):", random_float)
```

```
Random Float (0 to 1): 0.34084275159525823
```

In [51]:
```
#Generate a 1D array of 5 random floats between 0 and 1
random_array = np.random.random(5)
print("\n1D Array of Random Floats:", random_array)
```

```
1D Array of Random Floats: [0.73009679 0.46059969 0.12029021 0.03861604 0.69150241]
```

In [52]:
```
#Generate a 2D array (3x3) of random floats between 0 and 1
random_matrix = np.random.random((3, 3))

print("\n2D Array (3x3) of Random Floats:\n", random_matrix)
```

```
2D Array (3x3) of Random Floats:
 [[0.78038619 0.45346296 0.92939059]
 [0.37551905 0.40383647 0.47406553]
 [0.73208494 0.89440378 0.73634312]]
```

In [53]:
```
#Generate random integers between 10 and 50 (5 numbers)
random_integers = np.random.randint(10, 50, size=5)
print("\nRandom Integers between 10 and 50:", random_integers)
```

```
Random Integers between 10 and 50: [46 24 21 25 45]
```

In [54]:
```
#Generate random numbers from a standard normal distribution (mean=0, std=1)
random_normal = np.random.randn(4)
print("\nRandom Numbers from Normal Distribution:", random_normal)
```

```
Random Numbers from Normal Distribution: [ 0.70827001 -0.72915831 -0.66503172  0.26473817]
```

In [55]:
```
#Reproducibility - Set a random seed
np.random.seed(42)
```

```
print("\nRandom Numbers with Seed=42 (Reproducible):", np.random.random(3))
```

```
Random Numbers with Seed=42 (Reproducible): [0.37454012 0.95071431 0.73199394]
```

In [57]:
```
#Create a 1D array
arr1d = np.array([10, 20, 30, 40, 50, 60])
print("1D Array:", arr1d)
```

```
1D Array: [10 20 30 40 50 60]
```

In [58]:
```
print("\nIndexing on 1D Array:")
print("Element at index 0:", arr1d[0])   # First element
print("Element at index -1:", arr1d[-1]) # Last element (negative index)
```

```
Indexing on 1D Array:
Element at index 0: 10
Element at index -1: 60
```

In [60]:
```
print("\nSlicing on 1D Array:")
print("Elements from index 1 to 4:", arr1d[1:5])  # From index 1 up to 4
print("Every second element:", arr1d[::2])        # Step of 2
print("Reverse array:", arr1d[::-1])
```

```
Slicing on 1D Array:
Elements from index 1 to 4: [20 30 40 50]
Every second element: [10 30 50]
Reverse array: [60 50 40 30 20 10]
```

In [61]:
```
#Create a 2D array (3x4 matrix)
arr2d = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])
print("\n2D Array:\n", arr2d)
```

```
2D Array:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

In [62]:
```
print("\nIndexing on 2D Array:")
print("Element at row 1, column 2:", arr2d[1, 2])   # 7 (row=1, col=2)
print("First row:", arr2d[0])                       # Whole first row
print("First column:", arr2d[:, 0])
```

```
Indexing on 2D Array:
Element at row 1, column 2: 7
First row: [1 2 3 4]
First column: [1 5 9]
```

In [63]:
```
print("\nSlicing on 2D Array:")
print("Subarray (first 2 rows, columns 1 to 3):\n", arr2d[0:2, 1:3])
print("Last two rows:\n", arr2d[-2:, :])           # Last 2 rows
print("Every second column:\n", arr2d[:, ::2])     # All rows, step of 2 columns
```

```
Slicing on 2D Array:
Subarray (first 2 rows, columns 1 to 3):
 [[2 3]
 [6 7]]
Last two rows:
 [[ 5  6  7  8]
 [ 9 10 11 12]]
Every second column:
 [[ 1  3]
 [ 5  7]
 [ 9 11]]
```

In [ ]:
```
```