Data Analytics (24CAI0107) (CHO)

Just a quick revision of python

Data type

classification-01

1. Number: int, float, complex, bool
2. Sequence: String, List, Tuple
3. Set: set
4. None: None
5. Mapping: Dictionary

classification-02

1. Mutable: List Set Dictionary
2. Immutable: int, float, complex, bool, String, Tuple, None

What is output of following code

```
a = 10
b = 10
print(a == b
print(a is b)
a = a + 1
a = a - 1
print(a == b)
print(a is b)

A. T T T F
B. T F T F
C. T T T T
D. T T F F
E. None of these

Explanation
stack            heap
|767868|        |__10__|
    a            767868

|767868|
   b

print(id(a) == id(b)) #True
```

What is output of following code

```
l1 = [10, 20, 30]
l2 = [10, 20, 30]
print(l1 == l2)
print(l1 is l2)
l1.append(40)
```

```
l1.pop()
print(l1 == l2)
print(l1 is l2)

A. T T T F
B. T F T F
C. T T T T
D. T T F F
E. None of these

Explanation
stack          heap
|67891|        |_10_20_30_|
  l1           67891

|78654|        |_10_20_30_|
  l2           78654

l3 = l1
print(l1 == l3)
print(l1 is l3)
print(l2 == l3)
print(l2 is l3)

Explanation
stack          heap
|67891|        |_10_20_30_|
  l1           67891

|67891|
   l3
```

Summarization

a is b	a is exactly b (same object)	a == b will also be True
a == b	a has a value equal to what b has	a is b may still be False

List:
Mutable, Ordered
Indexes in the list
- The first element is available is 0 and the last at N - 1 where N is the length of the list
- The last element is available is -1 and the first at -N

```
l1 = list(range(10, 20, 2))
#elements: [10, 12, 14, 16, 18]
#Indexes:    0   1   2   3   4
#Indexes:   -5  -4  -3  -2  -1
```

```
print(l1[0])  #10
print(l1[-5]) #10
print(l1[4])  #18
print(l1[-1]) #18
print(l1[2] + l1[-3]) #28

print(l1[:])  #[10, 12, 14, 16, 18]
print(l1[::-1]) #[18, 16, 14, 12, 10]
print(l1[0:len(l1):2])  #[10, 14, 18]
print(l1[len(l1)-1:0:-2])  #[18, 14]
print(l1[: -1])  #
```

list functions: append, extend, insert, index, clear, pop, remove

```
s1 = "G-6 G-7 G-8"
l1 = s1.split(" ")
print(l1) #[G-6, G-7, G-8]

l2 = ['DSA', 'DA', 'OS', 'SA']
s2 = ", ".join(l2)
print(s2) #DSA, DA, OS, SA
```

(i) use list comprehension to generate a list having elements from 0 to 9

```
l1 = [i for i in range(10)]
print(l1)
```

(ii) use list comprehension to generate a list that has all even elements from the list generate above

```
l2 = [j for j in l1 if j % 2 == 0]
print(l2)
```

(iii) use list comprehension to generate a list that has elements from -5 to 5 and then generate a list that has all elements from the list created earlier but instead of negative numbers it should have value 0

```
l3 = [k if k >= 0 else 0 for k in range(-5, 6)]
print(l3)
```

(iv) use the list from question (i) and generate a list of all odd numbers from that list [hint: use filter and lambda function]

```
l4 = list(filter(lambda x: x % 2 == 1, l1))
print(l4)
```

(v) use the list from question (iv) and generate a list of square of every element from that list [hint: use map and lambda function]

```
l5 = list(map(lambda x: x ** 2, l4))
print(l5)
```

What is output of following code

```
def fun(a, b):
  a = a + "!"
  b.append(30)

s1 = "Hello"
l1 = [10, 20]
fun(s1, l1)
print(s1, l1)

A. Hello [10, 20]
B. Hello! [10, 20, 30]
C. Hello! [10, 20]
D. Hello [10, 20, 30]
E. None of these
```

Type of parameter
1. Positional Parameter (see above example)
2. default Argument

```
def area_of_circle(radius, pi = 3.14):  #pi is default argument
  return pi * radius ** 2

print(area_of_circle(7.0))
print(area_of_circle(7.0, 3.142876))
```

3. Variable length argument

```
def get_average(*l1):
  total = 0
  for element in l1:
    total+=element
  return total/len(l1)

print(get_average(10, 20))
print(get_average(10, 20, 30))
print(get_average(20))
```

4. keyword argument

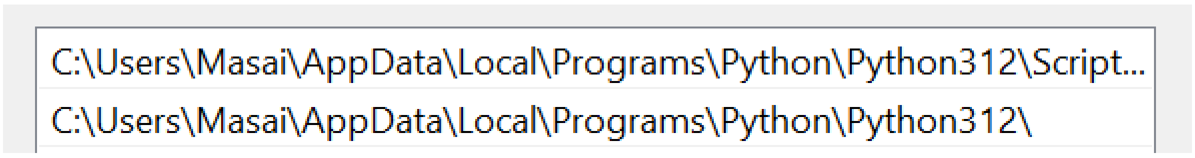
```
def print_details(**k):
  print(k)

print_details(name = "ABC", marks = 67.87)
print_details(city = "Chandigarh", population = 1334345435)
```

Installing jupyter notebook on Windows
Primary Requirement

1. Python 3.7 or later
2. Windows 7 or later

3. RAM: 4GB (atleast), 8GB or more is recommended
4. Storage: 1GB or more

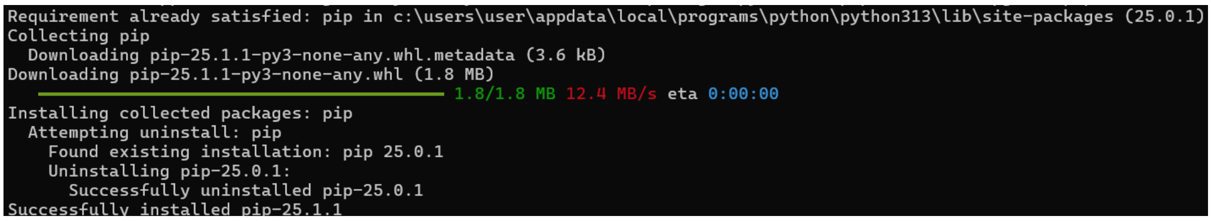
Step 1: Press window key, type ‘Edit Environment Variables for your account’ then double click on the path variable, it should have two values



```
C:\Users\Masai\AppData\Local\Programs\Python\Python312\Script...
C:\Users\Masai\AppData\Local\Programs\Python\Python312\
```

Step 2: Use the following command to update pip (to verify if pip is updated):

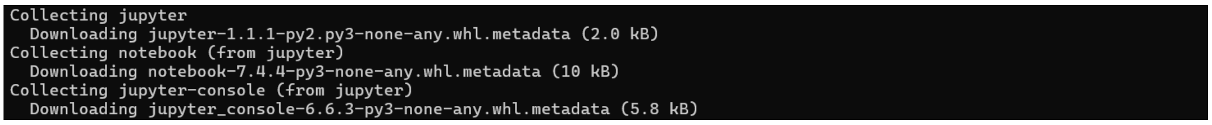
```
python -m pip install --upgrade pip

OR

py -m pip install --upgrade pip
```



```
Requirement already satisfied: pip in c:\users\user\appdata\local\programs\python\python313\lib\site-packages (25.0.1)
Collecting pip
  Downloading pip-25.1.1-py3-none-any.whl.metadata (3.6 kB)
Downloading pip-25.1.1-py3-none-any.whl (1.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 12.4 MB/s eta 0:00:00
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 25.0.1
    Uninstalling pip-25.0.1:
      Successfully uninstalled pip-25.0.1
Successfully installed pip-25.1.1
```

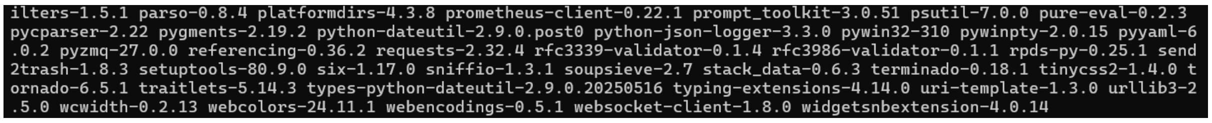
Step 3: Install Jupyter Notebook

```
python -m pip install jupyter

or

pip install jupyter
```



```
Collecting jupyter
  Downloading jupyter-1.1.1-py2.py3-none-any.whl.metadata (2.0 kB)
Collecting notebook (from jupyter)
  Downloading notebook-7.4.4-py3-none-any.whl.metadata (10 kB)
Collecting jupyter-console (from jupyter)
  Downloading jupyter_console-6.6.3-py3-none-any.whl.metadata (5.8 kB)
```

…



```
ilters-1.5.1 parso-0.8.4 platformdirs-4.3.8 prometheus-client-0.22.1 prompt_toolkit-3.0.51 psutil-7.0.0 pure-eval-0.2.3
pycparser-2.22 pygments-2.19.2 python-dateutil-2.9.0.post0 python-json-logger-3.3.0 pywin32-310 pywinpty-2.0.15 pyyaml-6
.0.2 pyzmq-27.0.0 referencing-0.36.2 requests-2.32.4 rfc3339-validator-0.1.4 rfc3986-validator-0.1.1 rpds-py-0.25.1 send
2trash-1.8.3 setuptools-80.9.0 six-1.17.0 sniffio-1.3.1 soupsieve-2.7 stack_data-0.6.3 terminado-0.18.1 tinycss2-1.4.0 t
ornado-6.5.1 traitlets-5.14.3 types-python-dateutil-2.9.0.20250516 typing-extensions-4.14.0 uri-template-1.3.0 urllib3-2
.5.0 wcwidth-0.2.13 webcolors-24.11.1 webencodings-0.5.1 websocket-client-1.8.0 widgetsnbextension-4.0.14
```

Step 4: Launching Jupyter from the folder of your choice. I have created a folder in documents named ‘data_analytics’. Get into this folder from the command prompt and then write the following command.

```
jupyter notebook

Or
```

```
Python -m notebook
```

Steps to Install Jupyter Notebook on Mac

Step 1: Open Terminal: Press Cmd + Space, type Terminal, hit Enter.
Step 2: Check if Python is installed

```
python3 --version
```

If not installed, download Python from:
https://www.python.org/downloads/mac-osx/

Note: During installation, ensure "Add to PATH" is checked (macOS does this automatically if using official installer).

Step 3: Install pip if not present; check if pip exists:

```
python3 -m pip --version
```

If pip is missing, run:
sudo easy_install pip

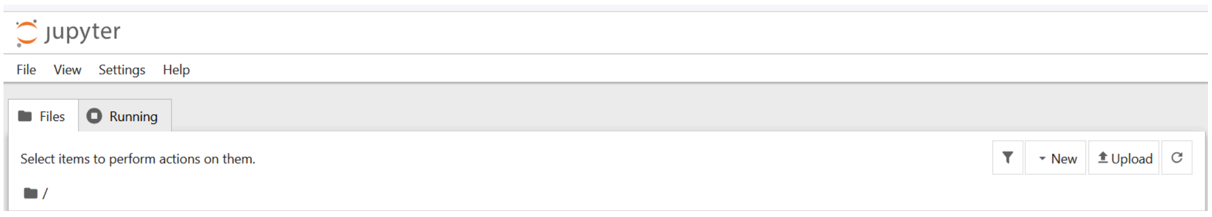
Step 4: Now install Jupyter Notebook using:

```
python3 -m pip install jupyter
```

Step 5: Launch Jupyter Notebook

```
jupyter notebook
```

It will open Jupyter in your default browser. If it doesn't open automatically, visit the link shown in the terminal (usually http://localhost:8888/tree). Now, the Jupyter Notebook will launch automatically in your default web browser. You can run your python code there



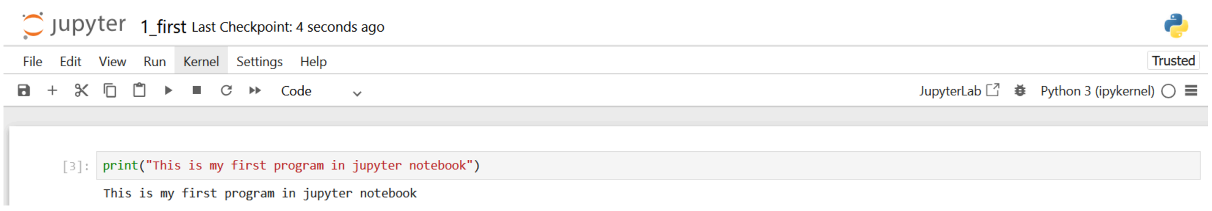
Select items to perform actions on them.

Use the new button to create a new file; set the file name to 1_first it will be saved with the name 1_first.pynb. The `ipynb` extension is for 'interactive python notebook'.

Type the code in the cell; say the code is

```
print("This is my first program in jupyter notebook")
```

To run, be in the same cell and press Shift + Enter you will get output with a new cell. You can continue to work in the same cell of in the new cell

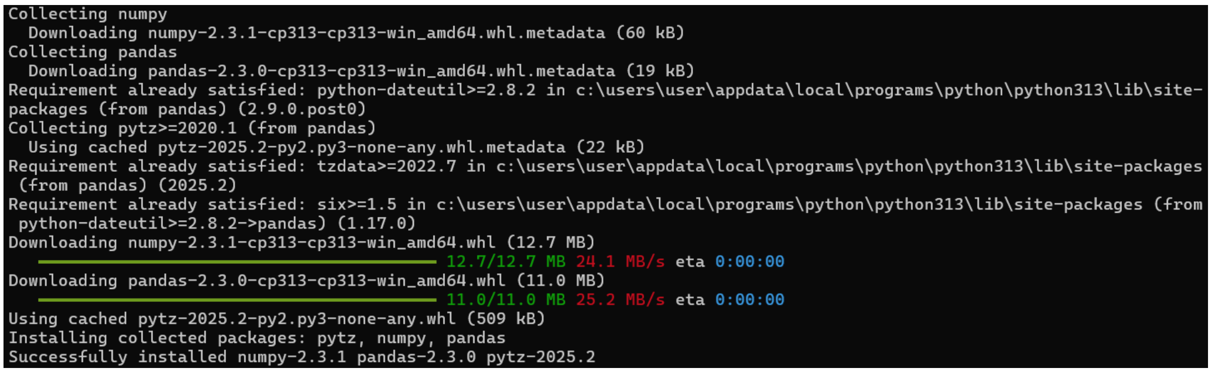

jupyter 1_first Last Checkpoint: 4 seconds ago
Trusted
File Edit View Run Kernel Settings Help
Code
JupyterLab
Python 3 (ipykernel)

```
[3]: print("This is my first program in jupyter notebook")
     This is my first program in jupyter notebook
```

To turn off the jupyter notebook File -> Shut Down or File -> Close and Shut DOwn Notebook in the web browser (of the jupyter notebook window). Another way to do the same is you have to press Ctrl + C on the command prompt. It will take some time to turn off.
Another option is also there File -> Log Out which is useful when multiple different users are working on a server.

Installing numpy and pandas
Step 1: same as to install the jupyter notebook
Step-2: Install both packages using command

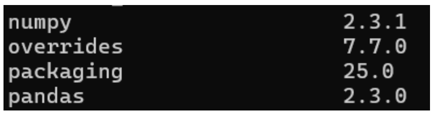
```
pip install numpy pandas
```



```
Collecting numpy
  Downloading numpy-2.3.1-cp313-cp313-win_amd64.whl.metadata (60 kB)
Collecting pandas
  Downloading pandas-2.3.0-cp313-cp313-win_amd64.whl.metadata (19 kB)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\user\appdata\local\programs\python\python313\lib\site-
packages (from pandas) (2.9.0.post0)
Collecting pytz>=2020.1 (from pandas)
  Using cached pytz-2025.2-py2.py3-none-any.whl.metadata (22 kB)
Requirement already satisfied: tzdata>=2022.7 in c:\users\user\appdata\local\programs\python\python313\lib\site-packages
 (from pandas) (2025.2)
Requirement already satisfied: six>=1.5 in c:\users\user\appdata\local\programs\python\python313\lib\site-packages (from
 python-dateutil>=2.8.2->pandas) (1.17.0)
Downloading numpy-2.3.1-cp313-cp313-win_amd64.whl (12.7 MB)
   ---------------------------------------- 12.7/12.7 MB 24.1 MB/s eta 0:00:00
Downloading pandas-2.3.0-cp313-cp313-win_amd64.whl (11.0 MB)
   ---------------------------------------- 11.0/11.0 MB 25.2 MB/s eta 0:00:00
Using cached pytz-2025.2-py2.py3-none-any.whl (509 kB)
Installing collected packages: pytz, numpy, pandas
Successfully installed numpy-2.3.1 pandas-2.3.0 pytz-2025.2
```

Step-3: You can check the list of installed packages using following command

```
pip list
```



```
numpy                2.3.1
overrides            7.7.0
packaging            25.0
pandas               2.3.0
```

Numpy and pandas will be available in your list.

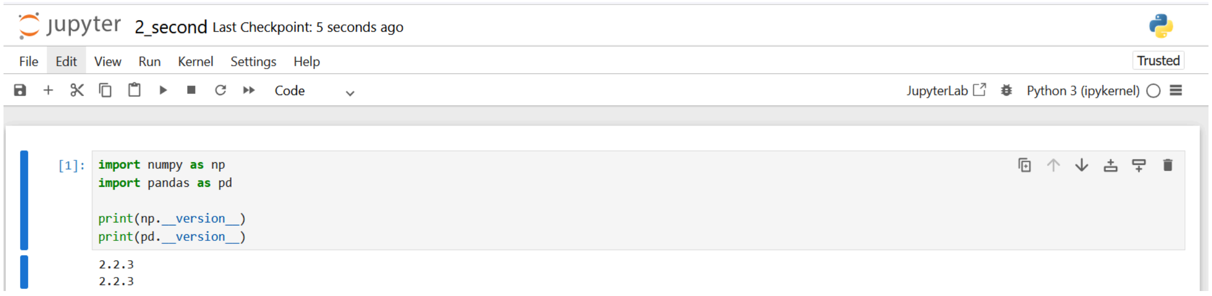
To use numpy and pandas we have to use the import statement in our code; let us take an example to print the version. Launch jupyter notebook and create a new file named 2_second and write following code in the same

```
import numpy as np
import pandas as pd
print(np.__version__)
print(pd.__version__)
```

Press Shift + Enter to run the code the output (might be different, subjected to version) is

```
2.2.3
2.2.3
```

Here is the screenshot of the same for your reference



```
import numpy as np
import pandas as pd

print(np.__version__)
print(pd.__version__)
```

```
2.2.3
2.2.3
```

Introduction to Numpy

NumPy stands for 'Numerical Python'. It is a Python library created in 2005. It is a package for data analysis and scientific computing with Python. NumPy uses a multidimensional array object, and has functions and tools for working with these arrays. The powerful n-dimensional array in NumPy speeds-up data processing. NumPy also includes a wide range of mathematical functions, such as linear algebra, Fourier transforms, and random number generation, which can be applied to arrays.

NumPy is an important library generally used for:

1. Machine Learning
2. Data Science
3. Image and Signal Processing
4. Scientific Computing
5. Quantum Computing

An array is a data type used to store multiple values using a single identifier (variable name). An array contains an ordered collection of data elements where each element is of the same type and can be referenced by its index (position).

The important characteristics of an array are:

- Each element of the array is of the same data type, though the values stored in them may be different. Each element of the array is identified or referred using the name of the Array along with the index of that element, which is unique for each element. The index of an element is an integral value associated with the element, based on the element's position in the array
- The entire array is stored contiguously in memory. This makes operations on arrays fast.
- The NumPy array is officially called ndarray but commonly known as array.

Difference Between List and Array

List	Array
List can have elements of different data types for example, [1,3.4, 'hello', 'a@']	All elements of an array are of same data type for example, an array of floats may be: [1.2, 5.4, 2.7]

Elements of a list are not stored contiguously in memory.	Array elements are stored in contiguous memory locations. This makes operations on arrays faster than lists.
You can create lists of lists, but there's no built-in support for true multi-dimensional operations.	Designed for multi-dimensional arrays (ndarray) with built-in support for slicing, reshaping, transposing, etc.
Lists do not support element wise operations, for example, addition, multiplication, etc. because elements may not be of the same type.	Arrays support element wise operations. For example, if A1 is an array, it is possible to say A1/3 to divide each element of the array by 3.
Slicing a list creates a copy.	Slicing returns a view (not a copy), unless explicitly copied.
Lists can contain objects of different datatypes that Python must store the type information for every element along with its element value. Thus lists take more space in memory and are less efficient.	NumPy array takes up less space in memory as compared to a list because arrays do not require to store datatype of each element separately.
List is a part of core Python.	Array (ndarray) is a part of NumPy library

Creating an numpy Array

1. **Using array() method:**
   An Example: create a file 3_array_creation.ipynb in jupyter notebook and put following code

```
import numpy as np
ar_1 = np.array([10, 20, 30, 40, 50])
print(ar_1)

ar_2 = np.array([10, 20, 30, 40, 50], dtype = float)
print(ar_2)

ar_3 = np.array([1, 2, '3', 4])
print(ar_3)

ar_4 = np.array([[10, 20], [100, 90]])
print(ar_4)

Output
[10 20 30 40 50]
[10. 20. 30. 40. 50.]
['1' '2' '3' '4']
[[ 10  20]
 [100  90]]
```

Note: The dtype is used to specify the desired data-type for the array. If not given, NumPy will try to use a default dtype that can represent the values (by applying promotion rules when necessary.)

2. **Using zeros() method**
   Continue the code in 3_array_creation.ipynb

```
ar_5 = np.zeros(3)
print(ar_5)

ar_6 = np.zeros((3,4))
print(ar_6)

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

   We can create an array with all elements initialised to 0 using the function zeros(). By default, the data type of the array created by zeros() is float.

3. **Using ones() method**
   Continue the code in 3_array_creation.ipynb

```
ar_7 = np.ones(5)
print(ar_7)

ar_8 = np.ones((2,3))
print(ar_8)

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

   We can create an array with all elements initialised to 1 using the function ones(). By default, the data type of the array created by zeros() is float.

4. **Using arange() function:**
   Continue the code in 3_array_creation.ipynb

```
ar_9 = np.arange(6)
print(ar_9)

ar_10 = np.arange(2, 15, 3)
print(ar_10)

Output
[0 1 2 3 4 5]
[ 2  5  8 11 14]
```

We can create an array with numbers in a given range and sequence using the arange() function. This function is analogous to the range() function of Python.

Head Scratcher

Which of the following is correct way to create an array in numpy

(A) `a = np.array([10, 20], [100, 90])`

(B) `a = np.array(10, 20, 30, 40)`

(C) `a = np.array([])`

(D) `a = np.array()`

(E) a = np.array(10)

1. A & B
2. B,C & D
3. C, E
4. Only D
5. None of these

RIGHT ANSWER: 3. C, E

Attributes of the numpy Array

1. **ndim:** gives the number of dimensions of the array as an integer value. Arrays can be 1-D, 2-D or n-D. NumPy calls the dimensions as axes (plural of axis). Thus, a 2-D array has two axes. The row-axis is called axis-0 and the column-axis is called axis-1. The number of axes is also called the array's rank.

```
print("The dimension of ", ar_3, "is", ar_3.ndim)
print("The dimension of ", ar_4, "is", ar_4.ndim)

Output
The dimension of  ['1' '2' '3' '4'] is 1
The dimension of  [[ 10  20]
 [100  90]] is 2
```

2. **shape:** It gives the sequence of integers indicating the size of the array for each dimension.

```
print("The shape of ", ar_3, "is", ar_3.shape)
      print("The shape of ", ar_4, "is", ar_4.shape)

Output
The shape of  ['1' '2' '3' '4'] is (4,)
The shape of  [[ 10  20]
 [100  90]] is (2, 2)
```

What is a 3D array then?

To be a 3D array, it must be a collection of 2D arrays (matrices). For example:

```
np.array([
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]]
])
Would give:

Shape: (2, 2, 3) → A 3D array
```

- `2 blocks`

- `each block with 2 rows and 3 columns`

3. **size:** It gives the total number of elements of the array. This is equal to the product of the elements of shape.

```
print("The size of ", ar_1, "is", ar_1.size)
print("The size of ", ar_4, "is", ar_4.size)

Output
The size of  [10 20 30 40 50] is 5
The size of  [[ 10  20]
 [100  90]] is 4
```

4. **dtype:** is the data type of the elements of the array. All the elements of an array are of same data type. Common data types are int32, int64, float32, float64, U32, etc

```
print("The data type of ", ar_2, "is", ar_2.dtype)
print("The data type of ", ar_3, "is", ar_3.dtype)
print("The data type of ", ar_4, "is", ar_4.dtype)

Output
The data type of  [10. 20. 30. 40. 50.] is float64
The data type of  ['1' '2' '3' '4'] is <U21
The data type of  [[ 10  20]
 [100  90]] is int64
```

5. **itemsize:** It specifies the size in bytes of each element of the array. Data type int32 and float32 means each element of the array occupies 32 bits in memory. 8 bits form a byte. Thus, an array of elements of type int32 has itemsize 32/8=4 bytes. Likewise, int64/float64 means each item has itemsize 64/8=8 bytes.

```
print("The itemsize of ", ar_2, "is", ar_2.itemsize)
print("The itemsize of ", ar_4, "is", ar_4.itemsize)

Output
```

```
      The itemsize of  [10. 20. 30. 40. 50.] is 8
      The itemsize of  [[ 10  20]
       [100  90]] is 8
```

Head Scratcher

What is output of following code

```
a = np.array(10)
b = np.array([])
print(a.ndim + b.ndim)
print(a.size + b.size)
```

(A) Error, Invalid way to create an array
(B) 1 1
(C) 1 2
(D) 2 1
(E) None of these

RIGHT ANSWER: B. 1 1

A note on the data-type in numpy for 64 bit system

Data Type	**Description**
int64	64-bit signed integers (e.g., 9223372036854775807 max)
uint64	64-bit unsigned integers (e.g., 0 to 18446744073709551615)
float64	64-bit floating-point numbers (default for real numbers)
complex128	Two 64-bit floats: one for real, one for imaginary part
bool_	Boolean type (True / False)
S	ASCII byte strings (S10 = 10-byte fixed-length string)
U	Unicode strings (U10 = 10-char fixed-length Unicode string)
datetime64	Dates/times (e.g., '2024-06-30')
timedelta64	Differences between two dates/times
object_	Generic Python object type (used for mixed types or non-native types)

Indexing and slicing in Array

The indexing and the slicing operation is almost the same as what we have already learned in the list. Let us take an example

```
import numpy as np
print("========Index and slicing operator in the 1-D
Array========")
ar_11 = np.arange(10, 25, 2)
```

```
print(ar_11)
print("ar_11[1] =",ar_11[1])
print("ar_11[-1] =",ar_11[-1])
print("ar_11[1 : 5] =", ar_11[1 : 5])
print("ar_11[1 : ar_11.size : 2] =", ar_11[1 : ar_11.size : 2])
print("ar_11[ar_11.size - 1 : 0 : -2] =", ar_11[ar_11.size - 1 : 0
: -2])

Output
========Index and slicing operator in the 1-D Array========
[10 12 14 16 18 20 22 24]
ar_11[1] = 12
ar_11[-1] = 24
ar_11[1 : 5] = [12 14 16 18]
ar_11[1 : ar_11.size : 2] = [12 16 20 24]
ar_11[ar_11.size - 1 : 0 : -2] = [24 20 16 12]


print("========Index and slicing operator in the 2-D
Array========")
ar_12 = np.array([np.arange(21, 10, -2), np.arange(10, -1, -2),
np.zeros(6, dtype = np.int64)])
print(ar_12)
print("ar_12[0, 0] =",ar_12[0, 0]) # 21
print("ar_12[-1,- 2] =",ar_12[-1,- 2]) # 0
print("ar_12[0, 0:6:2] =",ar_12[0, 0:6:2]) #[21 17 13]
print("ar_12[1:2, 5:0:-2] =",ar_12[1:2, 5:0:-2]) #[[0 4 8]]
print("ar_12[-2::-1, -3:-1] =",ar_12[-2::-1, -3:-1]) #[[ 4  2] [15
13]]

Output
========Index and slicing operator in the 2-D Array========
[[21 19 17 15 13 11]
 [10  8  6  4  2  0]
 [ 0  0  0  0  0  0]]
ar_12[0, 0] = 21
ar_12[-1,- 2] = 0
ar_12[0, 0:6:2] = [21 17 13]
ar_12[1:2, 5:0:-2] = [[0 4 8]]
ar_12[-2::-1, -3:-1] = [[ 4  2]
 [15 13]]
```

<u>Head Scratcher</u>

What is output of following code

```
a = np.arange(15, 4, -2)
print(a[6:0:-2])
```

(A) [7 11]

(B) [7 11 15]

(C) [11 7]
(D) [5 9 13]
(E) None of these

RIGHT ANSWER: D. [5 9 13]
Explanation: In Python slicing, if start is out of bounds, it doesn't throw an error but it just starts from the nearest valid index in the direction of the step so `a[6:0:-2]` is same as `a[5:0:-2]`.

Which of the following statements is false for 2-D array `ar`.
(A) ar[i][j] and ar[i, j] both return the same value
(B) ar[i, j] is faster than ar[i][j] hence preferred
(C) arr[r1:r2][c1:c2] first slices rows r1 to r2 - 1, then applies a second slice to the rows of the resulting subarray.
(D) arr[a:b, i:j] performs a true 2D slice, selecting rows and columns at once
(E) None of these is True

RIGHT ANSWER: (E) None of these is True

What is output of the following code
```
arr = np.array([
  [10, 11, 12],
  [20, 21, 22],
  [30, 31, 32],
  [40, 41, 42]
])
result = arr[1:3][0:1]
print(result)
```

(A) [[10 11 12]]
(B) [[20 21 22]]
(C) [[30 31 32]]
(D) [[20] [30]]
(E) None of these

RIGHT ANSWER: (B) [[20 21 22]]

Explanation:
1. arr[1:3]
This slices rows from index 1 to 2 (as 3 is excluded):

```
[
  [20, 21, 22],
  [30, 31, 32]
]
```
2. Now you do [0:1] on the result of arr[1:3]
That means:

Take row index 0 from the subarray [[20, 21, 22], [30, 31, 32]]

So:
result = [[20, 21, 22]]

What is output of the following code

```
arr = np.array([
  [10, 11, 12],
  [20, 21, 22],
  [30, 31, 32],
  [40, 41, 42]
])
result = arr[1:3, 0:2]
print(result)
```

(A) [[20 21] [30 31]]
(B) [[20 21 22] [30 31 32]]
(C) [[10 11] [20 21]]
(D) [[20] [30]]
(E) None of these

RIGHT ANSWER: (A) [[20 21] [30 31]]

Explanation:

arr[start_row:end_row, start_col:end_col]

1:3 → Selects row indices 1 and 2 (i.e., rows at index 1 and 2).

0:2 → Selects column indices 0 and 1 (i.e., columns at index 0 and 1).

What it selects:

From Row 1: [20, 21]

From Row 2: [30, 31]

So result becomes:

```
[
 [20, 21],
 [30, 31]
]
```

What is output of the following code

```
arr = np.array([
  [5, 6, 7, 8],
  [15, 16, 17, 18],
  [25, 26, 27, 28],
  [35, 36, 37, 38]
])
result = arr[1:3, -3:-1]
print(result)
```
(A) [[6 7] [16 17]]
(B) [[16 17] [26 27]]
(C) [[7 8] [17 18]]
(D) [[15 16] [25 26]]
(E) None of these

RIGHT ANSWER: (B) [[16 17] [26 27]]

What is output of the following code
```
arr = np.array([
  [5, 6, 7, 8],
  [15, 16, 17, 18],
  [25, 26, 27, 28],
  [35, 36, 37, 38]
])
result = arr[-1:-3:-1, -2:-4:-1]
print(result)
```
(A) [[27 26] [37 36]]
(B) [[26 27] [36 37]]
(C) [[36 37] [26 28]]
(D) [[37 36] [27 26]]
(E) None of these

RIGHT ANSWER: (D) [[37 36] [27 26]]

Different Array Operations

The basic operations on numpy arrays are ample but we are discussing a few important here. Let us start with the basic mathematical operations like additions, subtraction, multiplication, division, exponential and modulus.

```
import numpy as np
ar_1 = np.array([[1, 2],[3, 4]])
ar_2 = np.array([[4, 3],[2, 1]])
print(ar_1 + ar_2, end = '\n\n')
print(ar_1 - ar_2, end = '\n\n')
print(ar_1 * ar_2, end = '\n\n')
print(ar_1 / ar_2, end = '\n\n')
print(ar_1 % ar_2, end = '\n\n')
print(ar_1 ** ar_2, end = '\n\n')
```

```
Output
[[5 5]
```

```
 [5 5]]

[[-3 -1]
 [ 1  3]]

[[4 6]
 [6 4]]

[[0.25       0.66666667]
 [1.5        4.        ]]

[[1 2]
 [1 0]]

[[1 8]
 [9 4]]
```

Broadcasting in NumPy lets you perform operations on arrays of different shapes by automatically stretching the smaller array so both arrays line up for element-wise operations.
Rules of Broadcasting

- ❖ If arrays have a different number of dimensions, the shape of the smaller-dimensional array is padded with ones on the left side until both shapes have the same length.
- ❖ Broadcasting is applied from the last dimension to the first dimension.
- ❖ The size of each dimension must either be the same or one of them must be one.

Head Scratcher

What is output of following code

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([10, 20, 30])
result = a + b
print(result)
```

(A) [[11, 12, 13], [4, 5, 6]]
(B) [[1, 2, 3], [14, 15, 16]]
(C) [[11, 12, 13], [14, 15, 16]]
(D) [[1, 1, 1], [4, 5, 6]]
(E) None of these

RIGHT ANSWER: (C) [[11, 12, 13], [14, 15, 16]]

Explanation
The first array a is of size (2, 3) and the second array is of size (3,) it will be (1, 3) after broadcasting because the shape of the smaller-dimensional array is padded with ones on the left side. The last dimension matches which is 3 in both arrays but the first is not so now broadcasting is applied from the last dimension to the first dimension. i.e. it will be applied on the row side hence the array b will look like [[10, 20, 30], [10, 20, 30]]. This is called row

broadcasting. After row broadcasting all dimensions of both the arrays are the same. Now addition will be applied hence the result will be [[11, 12, 13], [14, 15, 16]].

What is output of following code

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[10], [20]])
result = a + b
print(result)
```

(A) [[11, 22, 13], [24, 15, 26]]
(B) [[21, 22, 23], [14, 15, 16]]
(C) [[21, 12, 23], [14, 25, 16]]
(D) [[11, 12, 13], [24, 25, 26]]
(E) None of these

RIGHT ANSWER: (D) [[11, 12, 13], [24, 25, 26]]
Explanation
The first array a is of size (2, 3) and the second array is of size (2, 1). When one array's trailing dimension is 1 (here, columns), NumPy stretches that column across the larger array—hence the term column broadcasting. Here one of the dimensions of b is 1 s o the array b will look like [[10, 10, 10], [20, 20, 20]]. This is called column broadcasting. Now addition will be applied hence the result will be [[11, 12, 13], [24, 25, 26]].

What is output of following code

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([10, 20, 30, 40])
result = a + b
print(result)
```

(A) [[11, 22, 33], [4, 5, 6]]
(B) [[1, 2, 3], [14, 25, 36]]
(C) [[11, 22, 33], [14, 25, 36]]
(D) [[11, 22, 33, 40], [14, 25, 36, 40]]
(E) None of these

RIGHT ANSWER: (E) None of these
The first array a is of size (2, 3) and the second array is of size (4,) it will be (1, 4) after broadcasting because the shape of the smaller-dimensional array is padded with ones on the left side but last dimension is mismatched hence broadcasting cannot be applied.

What is output of following code

```
a = np.array([[1, 2, 3], [4, 5, 6]])
scalar = 100
result = a + scalar
print(result)
```

(A) [[101, 102, 103], [4, 5, 6]]
(B) [[1, 2, 3], [104, 105, 106]]

(C) [[101, 102, 103], [104, 105, 106]]
(D) [[1, 2, 3], [4, 5, 6]]
(E) None of these

RIGHT ANSWER: (C) [[101, 102, 203], [104, 105, 106]]
Explanation: The scalar value is added to every element.

What is output of following code
```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([10, 20])
result = a + b
print(result)
```

(A) [[11, 22, 3], [4, 5, 6]]
(B) [[1, 2, 3], [14, 25, 3]]
(C) [[11, 22, 3], [14, 25, 3]]
(D) [[11, 22], [14, 25]]
(E) None of these

RIGHT ANSWER: (E) None of these
The first array a is of size (2, 3) and the second array is of size (2,) it will be (1, 2) after broadcasting because the shape of the smaller-dimensional array is padded with ones on the left side but last dimension is mismatched hence broadcasting cannot be applied.

Now we will have a discussion using relational operator with numpy array
```
ar_2 = np.array([[4, 3],[2, 1]])
ar_3 = np.array([[5, 4], [3, 2]])
print(ar_2 == ar_3, end = '\n\n')
print(ar_2 != ar_3, end = '\n\n')
print(ar_2 >= ar_3, end = '\n\n')
print(ar_2 <= ar_3, end = '\n\n')
print(ar_2 > ar_3, end = '\n\n')
print(ar_2 < ar_3, end = '\n\n')
```

```
Output
[[False False]
 [False False]]

[[ True  True]
 [ True  True]]

[[False False]
 [False False]]

[[ True  True]
 [ True  True]]

[[False False]
```

```
 [False False]]

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

Reshaping the array
Reshaping a NumPy array means changing how the data is organized like turning a 1D array into a 2D array without changing the actual data. The only rule is that the total number of elements must stay the same.

1. The reshape() function: This function returns a new view of the array with the specified shape if possible. If the reshape is not possible with a view, a copy of the array is created.
   ```
   If
     The data is contiguous in memory
     The new shape is compatible with row-major memory layout
     then it returns a view
   else
     It returns a copy
   ```

   Converting a two-D array one-D array:
   ```
   ar_4 = np.array([[1, 2, 3], [4, 5, 6]])
   ar_5 = np.reshape(ar_4 , -1)
   print(ar_5)

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

   Converting a one-D array two-D array
   ```
   ar_6 = np.arange(0, 16)
   ar_7 = np.reshape(ar_6, (4, 4))
   print(ar_7)

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

   To check if the reshaped array returns a view or not we have to use shares_memory() function of the numpy. For the above example
   ```
   print(np.shares_memory(ar_6, ar_7)) #True
   ```

   What is output of following code
   ```
   a = np.zeros((16, ))
   b = np.reshape(a, (3, 5))
   print(b)
   ```

(A) Error because an array of 16 elements cannot be converted to array of shape (3, 5)
(B) 0. for fifteen times (as two-D array)
(C) 0. for fifteen times (as one-D array)
(D) Random values for fifteen times (as two-D array)
(E) None of these

RIGHT ANSWER: (A) Error because an array of 16 elements cannot be converted to array of shape (3, 5)

What is output of following code

```
a = np.empty((2, 3))
b = np.reshape(a, -1.0)
print(b)
```

(A) Random values for 6 times (as one-D array)
(B) Error because the new shape must be an integer or tuple of integer
(C) 0. for 6 times (as one-D array)
(D) 1. for 6 times (as one-D array)
(E) None of these

RIGHT ANSWER: (B) Error because the new shape must be an integer or tuple of integer

2. The resize() function: Return a new array with the specified shape. If the new array is larger than the original array, then the new array is filled with repeated copies of the source array.

```
ar_8 = np.array([[1, 2, 3], [4, 5, 6]])
ar_9 = np.resize(ar_8 , (1, 6))
ar_10 = np.resize(ar_8 , (3, 2))
ar_11 = np.resize(ar_8 , (2, 4))
ar_12 = np.resize(ar_8 , (2, 2))
ar_13 = np.resize(ar_9 , (1, ))
print(ar_9)
print(ar_10)
print(ar_11)
print(ar_12)
print(ar_13)
```

Output

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

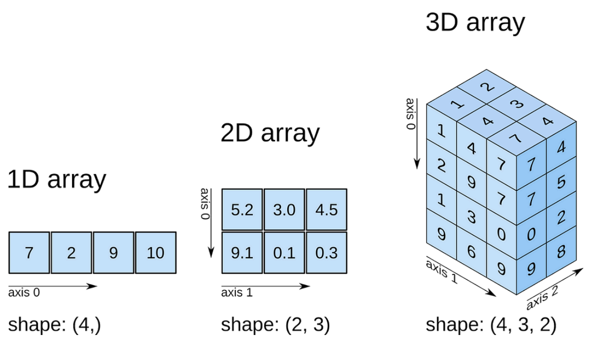
3. The flatten() method: Return a copy of the array collapsed into one dimension.
   ```
   ar_14 = ar_8.flatten()
   print(ar_14)
   ```
   Output
   ```
   [1 2 3 4 5 6]
   ```

4. The transpose() function: Returns an array with axes transposed.
   ```
   ar_15 = np.transpose(ar_8)
   print(ar_15)

   #the above is same as
   ar_16 = ar_8.T
   print(ar_16)
   ```
   Output
   ```
   [[1 4]
    [2 5]
    [3 6]]
   [[1 4]
    [2 5]
    [3 6]]
   ```

Know about Axis of array

Axes are simply the directions you can move along this grid. Think of them like the x, y, and z coordinates in a 3D space.

1. For 1-D array: The only axis available. Operations will apply across all elements of the array.
2. For 2-D array: The axis 0 operates along rows (vertically) but the axis 1 Operates along columns (horizontally)
3. For 3-D array: The axis=0 operates along the depth (the outermost dimension). Each operation is applied to the 2D arrays stacked along this axis. The axis 1 operates along rows within each depth level. The axis 2 operates along columns within each depth level.



Joining Array in Numpy

1. **concatenate():** Join a sequence of arrays along an existing axis.

```
import numpy as np
ar_1 = np.array([[1, 2], [3, 4]])  #No of Rows: 2 No of Col.:
2
ar_2 = np.array([[5, 6]])  #No of Rows: 1 No of Col.: 2
ar_3 = np.concatenate((ar_1, ar_2), axis=0)
print(ar_3, end = '\n\n')  #No of Rows: 3 No of Col.: 2
```

We are adding rows (i.e., stacking vertically along axis 0).
ar_1 is 2×2
ar_2 is 1×2 → same number of columns as ar_1

Output

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

```
ar_4 = np.concatenate((ar_1, ar_2.T), axis=1)  #No of Rows: 2
No of Col.: 3
print(ar_4, end = '\n\n')
```

We are adding columns (i.e., stacking horizontally along axis 1).
ar_2.T is the transpose of ar_2:
Original shape: (1, 2)
Transposed shape: (2, 1)

Now:
ar_1: shape (2, 2)
ar_2.T: shape (2, 1) → same number of rows

Output

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

```
ar_5 = np.concatenate((ar_1, ar_2), axis=None)  #No of Rows:
1 No of Col.: 6
print(ar_5, end = '\n\n')
```

When axis=None, NumPy flattens all arrays and concatenates them into a 1D array.
ar_1.flatten() = [1, 2, 3, 4]
ar_2.flatten() = [5, 6]

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

What is output of following code
```
ar_3d_1 = np.arange(8).reshape(2, 2, 2)
ar_3d_2 = np.arange(8, 16).reshape(2, 2, 2)
result = np.concatenate((ar_3d_1, ar_3d_2), axis=None)
print(result.shape)
```

(A) (16,)
(B) (2, 2, 2)
(C) (2, 2, 2, 2)
(D) (2, 8)
(E) None of these

RIGHT ANSWER: (A) (16,) In concatenate method if axis=None then it will create a 1-D array.

Student Activity :
Check output for axis=0 and axis=1 for above problem.

Hint :

For axis=0
We"re **concatenating along axis 0 → stacking along the outermost dimension**
Since both arrays are of shape `(2, 2, 2)`, this will result in a shape of `(4, 2, 2)`
Output:
```
([[[ 0,  1],
    [ 2,  3]],
   [[ 4,  5],
    [ 6,  7]],
   [[ 8,  9],
    [10, 11]],
   [[12, 13],
    [14, 15]]])
```

For axis=1
We are concatenating along the second dimension, i.e., adding more rows within each block (each 2D slice in the 3D array).
Since both arrays are of shape `(2, 2, 2)`, this will result in a shape of `(2, 4, 2)`
So:
axis=0 → increases number of blocks (outermost arrays)
axis=1 → increases number of rows inside each block
axis=2 → increases number of columns inside each row

Output:
```
array([[[ 0,  1],
        [ 2,  3],
        [ 8,  9],
        [10, 11]],

       [[ 4,  5],
        [ 6,  7],
        [12, 13],
        [14, 15]]])
```

2. **hstack() & vstack():**
    - The hstack() stack arrays in sequence horizontally (column wise) such that the No of Rows of the array to be merged must be the same. In the result the axis-0 will have the same number of elements but the axis-1 will have more elements.
    - The vstack() stack arrays in sequence vertically (row wise) such that the No of Col. of the array to be merged must be the same. In the result the axis-1 will have the same number of elements but the axis-0 will have more elements.

Example 1
```
ar_6 = np.array((1,2,3))  #No of Rows: 1 No of Col.: 3
ar_7 = np.array((4,5,6))  #No of Rows: 1 No of Col.: 3
ar_8 = np.hstack((ar_6, ar_7))
print(ar_8)  #No of Rows: 1 No of Col.: 6
```

- For 1D arrays, it’s like simple concatenation.

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

Example 2
```
a = np.array([[1, 2],[3, 4]])  #No of Rows: 2 No of Col.: 2
b = np.array([[5, 6],[7, 8]])  #No of Rows: 2 No of Col.: 2
c = np.hstack((a, b))
print(c)   #No of Rows: 2 No of Col.: 4
```

```
Output:

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

🔁 Shape before: (2, 2) + (2, 2) → After: (2, 4)

➡️ Horizontal stack means same number of rows, columns increase.

Example 3

```
ar_9 = np.array([[1],[2],[3]])  #No of Rows: 3 No of Col.: 1
ar_10 = np.array([[4],[5],[6]])  #No of Rows: 3 No of Col.: 1
ar_11 = np.hstack((ar_9, ar_10))
print(ar_11)  #No of Rows: 3 No of Col.: 2
```

🔁 Shape before: (3, 1) + (3, 1) → After: (3, 2)
➡️ Horizontal stack means same number of rows, columns increase.

Output

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

Example 4

```
ar_12 = np.array([1, 2, 3])  #No of Rows: 1 No of Col.: 3
ar_13 = np.array([4, 5, 6])  #No of Rows: 1 No of Col.: 3
ar_14 = np.vstack((ar_12, ar_13))
print(ar_14)  #No of Col.: 3 No of Rows: 2
```

Output

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

🔁 Shape before: (1, 3) + (1, 3) → After: (2, 3)
⬇️ Vertical stack means **same number of columns**, rows increase.

Example 5

```
ar_15 = np.array([[1],[2],[3]])  #No of Rows: 3 No of Col.: 1
ar_16 = np.array([[4],[5],[6]])  #No of Rows: 3 No of Col.: 1
ar_17 = np.vstack((ar_15, ar_16))
print(ar_17)  #No of Rows: 6 No of Col.: 1
```

Output

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

Head Scratcher

What is output of following code

```
import numpy as np
a = np.array([10, 20, 30])
b = np.array([100, 200])
c = np.hstack((a, b))
print(c.ndim + c.size)
```

(A) 7
(B) 6
(C) 0
(D) 1
(E) None of these

RIGHT ANSWER: (B) 6

- 1 (ndim) + 5 (size) = 6

What is output of following code

```
import numpy as np
a = np.array([10, 20, 30])
b = np.array([[100, 200], [70, 90]])
c = np.hstack((a, b))
print(c.ndim + c.size)
```

(A) 7
(B) 6
(C) 0
(D) 1
(E) None of these

RIGHT ANSWER: (E) None of these because vertical dimension is different

`np.hstack()` horizontally stacks arrays **along columns** (i.e., axis=1). All input arrays **must have the same number of dimensions**.

What is output of following code

```
import numpy as np
a = np.array([10, 20])
b = np.array([[100, 200], [300, 400]])
c = np.vstack((a, b))
print(c.ndim + c.size)
```

(A) 7
(B) 6
(C) 8
(D) 2
(E) None of these

RIGHT ANSWER: (C) 8

```
c =     [[ 10  20]
        [100 200]
        [300 400]]
```

c.ndim + c.size = 2 + 6 = 8

`np.vstack()` automatically **treats the 1D array as a row vector**:
→ `a` becomes `[[10, 20]]` → shape `(1, 2)`

What is output of following code

```
import numpy as np
a = np.array([10, 20, 30])
b = np.array([100, 200])
c = np.vstack((a, b))
print(c.ndim + c.size)
```

(A) 7
(B) 6
(C) 0
(D) 1
(E) None of these

RIGHT ANSWER: (E) None of these because horizontal dimension is different

3. **stack():** Join a sequence of arrays along a new axis. The arrays must be of same shape and dimensions. It creates a new array which has 1 more dimension than the input arrays. here is the difference between the concatenate and the stack function-

   When concatenate function is applied on
   - ❖ Two or more 1-D arrays then we get a 1-D array as result
   - ❖ Two or more 2-D arrays then we get a 2-D array as result

   When stack function is applied on
   - ❖ Two or more 1-D arrays then we get a 2-D array as result
   - ❖ Two or more 2-D arrays then we get a 3-D array as result

```
ar_18 = np.array([1, 2, 3])    #No of Rows: 1 No of Col.: 3
ar_19 = np.array([4, 5, 6])    #No of Rows: 1 No of Col.: 3
```

```
ar_20  = np.stack((ar_18, ar_19)) #The default value for the
axis is 0.
print(ar_20, end = '\n\n')      #No of Rows: 2 No of Col.: 3

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

ar_21 = np.stack((ar_18, ar_19), axis = 1)
print(ar_21, end = '\n\n')

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

ar_22 = np.array([[1,2,3], [4,5,6]])   #No of Rows: 2 No of
Col.: 3
ar_23 = np.array([[7,8,9], [10,11,12]])   #No of Rows: 2 No of
Col.: 3

ar_24 = np.stack((ar_22,ar_23), axis = 0)   #No of depth: 2 No
of Rows: 2 No of Col.: 3
ar_25 = np.stack((ar_22,ar_23), axis = 1)   #No of depth: 2 No
of Rows: 2 No of Col.: 3

print(ar_24, end = '\n\n')
print(ar_25, end = '\n\n')

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

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

Splitting Array

1. **hsplit() & vsplit():**
   This hsplit() function is used to split array along horizontal axis (i.e. column wise) into equal sub-parts and the vsplit() function is used to split array along vertical axis (i.e. row wise) into equal sub-parts.

   ```
   import numpy as np
   ar_1 = np.arange(24.0).reshape(4, 6)
   ```

```
print(np.hsplit(ar_1, 2), end = "\n\n")
print(np.hsplit(ar_1, 3), end = "\n\n")
#print(np.hsplit(ar_1,  4))  Error(  This  is  not  possible
because 6 is not divisible by 4)
```

Explanation:
np.hsplit(ar_1, 2)
hsplit = horizontal split = split columns
Total columns = 6
You are splitting into 2 equal parts → each will have 3 columns

Output
```
[array([[ 0.,  1.,  2.],
       [ 6.,  7.,  8.],
       [12., 13., 14.],
       [18., 19., 20.]]),
 array([[ 3.,  4.,  5.],
       [ 9., 10., 11.],
       [15., 16., 17.],
       [21., 22., 23.]])]
```

```
print(np.vsplit(ar_1, 2), end = "\n\n")
print(np.vsplit(ar_1, 4), end = "\n\n")
#print(np.vsplit(ar_1, 3)) Error(Not valid, because 4 is not
divisible by 3)
```

Explanation:
np.vsplit(ar_1, 2)
Vertical split = split along rows

4 rows split into 2 → each part will have 2 rows

Output
```
[array([[ 0.,  1.],
       [ 6.,  7.],
       [12., 13.],
       [18., 19.]]),
 array([[ 2.,  3.],
       [ 8.,  9.],
       [14., 15.],
       [20., 21.]]),
 array([[ 4.,  5.],
       [10., 11.],
        [16., 17.],
        [22., 23.]]])]
```

```
print(np.vsplit(ar_1, 2), end = "\n\n")
print(np.vsplit(ar_1, 4), end = "\n\n")
#print(np.vsplit(ar_1, 3))
```

Output

```
[
 array([[ 0.,  1.,  2.,  3.,  4.,  5.], [ 6.,  7.,  8.,  9.,
10., 11.]]),
 array([[12., 13., 14., 15., 16., 17.], [18., 19., 20., 21.,
22., 23.]])
]

[
 array([[0., 1., 2., 3., 4., 5.]]),
 array([[ 6.,  7.,  8.,  9., 10., 11.]]),
 array([[12., 13., 14., 15., 16., 17.]]),
 array([[18., 19., 20., 21., 22., 23.]])
]
```

Using np.hsplit() or np.vsplit() does **not increase the array's dimension**; it returns a **list of arrays** with the **same number of dimensions** as the original.

2. **split():**
   The split function is used to divide the array in equal parts along both the axes. By default it divides array long axis 0.
   The first parameter is the array to be split and the second parameter is integer or one-D array. If integer (Say N), the array will be divided into N equal arrays. If such a split is not possible, an error is raised. If it is a 1-D array of sorted integers, the entries indicate where the array is split. For example, [a, b] result in array[:a], array[a:b] & array[b:]. If an index exceeds the dimension of the array, an empty sub-array is returned correspondingly. You can define the axis also as parameter by default it is 0.

```
ar_2 = np.arange(8)
print(np.split(ar_2, 2))
```

Explanation:
np.split(ar_2, 2)
You are splitting the array into 2 equal parts
Since 8 is divisible by 2 → ✅ Works fine

```
Output:
[array([0, 1, 2, 3]), array([4, 5, 6, 7])]


print(np.split(ar_2, 4))

print(np.split(ar_2, (2,6)))
```

Explanation:

```
np.split(ar_2, (2, 6))
Here, you're giving indices where the array should be split —
not the number of parts.

We’re saying:
Split at index 2 → [0, 1]
Then at index 6 → [2, 3, 4, 5]
Remaining part → [6, 7]

🧾 Output:
[array([0, 1]), array([2, 3, 4, 5]), array([6, 7])]

print()


ar_3 = np.arange(16).reshape(4, 4)
print(np.split(ar_3, 2))
```

Explanation:

```
np.split(ar_3, 2)
Splitting along axis 0 (rows) by default
4 rows ÷ 2 → ✅ results in 2 arrays of shape (2, 4)

print(np.split(ar_3, 2, axis = 1))
```

Explanation:

```
np.split(ar_3, 2, axis=1)
Now splitting along axis 1 (columns)
4 columns ÷ 2 → ✅ results in 2 arrays of shape (4, 2)


print(np.split(ar_3, [1, 4]))
```

Explanation:

np.split(ar_3, [1, 4])
specifying indices [1, 4] → it will split along axis 0 (rows) at those indices:
First part: rows [0]
Second part: rows [1, 2, 3]
Third part: ❌ empty slice after index 4 (because 4 is the last row)

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

[array([[0, 1, 2, 3], [4, 5, 6, 7]]),
 array([[ 8,  9, 10, 11], [12, 13, 14, 15]])]
[array([[ 0,  1], [ 4,  5], [ 8,  9], [12, 13]]),
 array([[ 2,  3], [ 6,  7], [10, 11], [14, 15]])]
[array([[0, 1, 2, 3]]), array([[ 4,  5,  6,  7], [ 8,  9, 10,
11], [12, 13, 14, 15]]),
 array([], shape=(0, 4), dtype=int64)]
```

1. **array_split()**
   Split an array into multiple sub-arrays such that the sub-arrays are not required to be of equal size. For an array of length L that should be split into n sections, it returns L % n sub-arrays of size L//n + 1 and the rest of size L//n.

```
ar_4 = np.arange(8.0)
print(np.array_split(ar_4, 3))

ar_5 = np.arange(12).reshape((3, 4))
print(np.array_split(ar_5, 3))

print(np.array_split(ar_5, 3, axis = 1))
```

Output
```
[array([0., 1., 2.]), array([3., 4., 5.]), array([6., 7.])]
[array([[0, 1, 2, 3]]), array([[4, 5, 6, 7]]), array([[ 8,
9, 10, 11]])]
[array([[0, 1],
       [4, 5],
       [8, 9]]), array([[ 2],
       [ 6],
       [10]]), array([[ 3],
       [ 7],
       [11]])]
```

```
Explanation 1:
ar_4 = [0. 1. 2. 3. 4. 5. 6. 7.] → shape (8,)

You want to split it into 3 parts
Since 8 isn’t divisible by 3, NumPy will distribute elements
as evenly as possible
```

Adding/Removing element from array

1. **append():** Append values to the end of an array.

The first parameter is the array to which value to be appended The second parameter is the values to be appended to. The third parameter is the axis along which values are appended. **If the axis is not given, both are and values are flattened before use**. It returns a copy of the array with values appended to the axis.

```
import numpy as np
ar_1 = np.arange(1, 4)
ar_2 = np.arange(4, 10).reshape(2,3)
ar_3 = np.append(ar_1, ar_2)
print(ar_3)
```

Output

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

```
ar_4 = np.arange(1, 7).reshape(2, 3)
ar_5 = np.arange(7, 10).reshape(1, 3)
ar_6 = np.append(ar_4, ar_5, axis=0)
print(ar_6)
```

Output

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

```
ar_7 = np.array([[1, 2], [3, 4]])
ar_8 = np.array([[5], [6]])
result = np.append(ar_7, ar_8, axis=1)
print(result)
```

Output

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

What is output of following code

```
print(np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0))
```

(A) [1, 2, 3, 4, 5, 6, 7, 8, 9]
(B) [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
(C) [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
(D) Error
(E) None of these

RIGHT ANSWER: (D) Error

When using `np.append` **with `axis` specified**, both arrays must be **of the same dimension and shape** along all axes **except** the one you're appending on.

2. **insert():** Insert values along the given axis before the given indices.

The first parameter is the array to which value to be appended. The second parameter is the Object that defines the index or indices before which values are inserted. The third parameter is the values to be appended to. The fourth parameter is the axis along which values are appended. **If the axis is not given, both arr and values are flattened before use**. It returns a copy of the array with values appended to the axis.

```
ar_9 = np.arange(6).reshape(3, 2)
ar_10 = np.insert(ar_9, 1, 6)
print(ar_10)
```

Output
```
[0 6 1 2 3 4 5]
```

```
ar_11 = np.insert(ar_9, 1, 6, axis=1)
print(ar_11)
```
Output
```
[[0 6 1]
 [2 6 3]
 [4 6 5]]
```

```
ar_12 = np.insert(ar_9, [1], [[7],[8],[9]], axis=1)
print(ar_12 )
```
Output
```
[[0 7 1]
 [2 8 3]
 [4 9 5]]
```

```
Explanation:
[1] is a list → tells NumPy: "Insert 1 column"
[[7],[8],[9]] is used row-wise as the new column inserted at
index 1
```

What is output of following code
```
a = np.arange(6).reshape(3, 2)
b = np.insert(a, 1, [[7],[8],[9]], axis=1)
```

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

(B) Error

(C) [[0 7 1]
     [2 8 3]
     [4 9 5]]
```

```
(D) [[7 0 1]
     [8 2 3]
     [9 4 5]]

(E) [[0 1 7]
     [2 3 8]
     [4 5 9]]
```

RIGHT ANSWER: (A)
Explanation:
1 is scalar → NumPy flattens `[[7],[8],[9]]` → becomes `[7, 8, 9]`
That 1D array is then broadcast into **each row**, inserted at index 1

What is output of following code
```
b = np.array([0, 1, 2, 3, 4, 5])
a = np.insert(b, [2, 2], [7.13, False])
print(a)
```

(A) [0, 1, 7, 0, 2, 3, 4, 5]
(B) [0, 1, 2, 7, 0, 2, 3, 4, 5]
(C) [0, 1, 7, 1, 2, 3, 4, 5]
(D) [0, 1, 7, 2, 1, 2, 3, 4, 5]
(E) None of these

RIGHT ANSWER: (A)

Explanation:
If obj is an array of indices and values is an array of the same shape, the elements are inserted sequentially, one after the other, with earlier insertions shifting the indices for the later ones.

So:
Insert 7.13 at position 2 → [0, 1, 7, 2, 3, 4, 5]
Insert False at position 2 again → it's now index 3 (due to shift), so it lands after 7

✅ Data Type Notes:
Original array b is of type int

7.13 gets truncated to 7
False becomes 0

Final output : [0, 1, 7, 0, 2, 3, 4, 5]

3. **remove():**
It is a built-in Python list method that removes the **first occurrence** of a specified value from a list.

The first parameter is the Input array on which deletion takes place. The second parameter indicates indices of sub-arrays to remove along the specified axis. The third parameter is the axis.
If the axis is not given, both arr and values are flattened before use for delete.

```
ar_13 = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
ar_14 = np.delete(ar_13, 1, 0)
print(ar_14)
```

Output
```
[[ 1  2  3  4]
 [ 9 10 11 12]]
```

```
Explanation:
np.delete(array, obj, axis)
array: ar_13 → a 2D array with shape (3, 4)
obj: 1 → index of the item to delete
axis: 0 → delete row at index 1 (since axis=0 means row-wise)
```

What is output of following code
```
a = np.arange(1, 13).reshape(3, 4)
b = np.delete(a, [1,3,5], None)
print(b)
```

(A) [1 2 3 4 5 6 7 8 9 10 11 12]
(B) [1 3 5 7 8 9 10 11 12]
(C) [1 2 4 5 8 9 10 11 12]
(D) [2 4 6 7 8 9 10 11 12]
(E) None of these

Answer : (B)

Explanation:
`arr`: The array `a`
`obj`: The indices `[1, 3, 5]` to delete
`axis=None`: This means **flatten the array into 1D** first, delete the elements, then return a 1D result.

Sorting element of array

1. **sort():** Return a sorted copy of an array. The first parameter is an array to be sorted. The second parameter is the axis along which to sort. This method returns a sorted copy of an array.

```
import numpy as np
ar_1 = np.array([[1,4],[3,1]])
ar_2 = np.sort(ar_1)
```

```
print(ar_2)
```

Output
```
[[1 4]
 [1 3]]
```

```
ar_3 = np.sort(ar_1, axis = 0)
print(ar_3)
```

Output
```
[[1 1]
 [3 4]]
```

```
ar_4 = np.sort(ar_1, axis = None)
print(ar_4)
```

Output
```
[1 1 3 4]
```

Say we have an array in which every element of the array is a tuple, the tuple has three elements that are name, height (in meters) and age. Here is a list of value
```
values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38),
('Galahad', 1.7, 38)]
```

The data type for name is U10 (Unicode string), height of type float and age is of type int.
```
dtype = [('name', 'U10'), ('height', float), ('age', int)]
```

Let us create array from it
```
ar_5 = np.array(values, dtype=dtype)
print(ar_5, end = '\n\n')
```

Output
```
[('Arthur', 1.8, 41) ('Lancelot', 1.9, 38) ('Galahad', 1.7,
38)]
```

Now we want to sort the array in the the ascending order of height
```
ar_6 = np.sort(ar_5, order='height')
print(ar_6, end = '\n\n')
```

Output
```
[('Galahad', 1.7, 38) ('Arthur', 1.8, 41) ('Lancelot', 1.9,
38)]
```

Now we want to sort the array in the the ascending order of age, If age is same second criteria of sorting will be height
```
ar_7 = np.sort(ar_5, order=['age', 'height'])
print(ar_7)
```

Output
```
[('Galahad', 1.7, 38) ('Lancelot', 1.9, 38) ('Arthur', 1.8,
41)]
```

Linear Algebra operation

1. The @ operator
   Used for computing the matrix product between 2d arrays

```
import numpy as np
ar_1 = np.array([[1, 2], [3, 4]])
ar_2 = np.array([[3, 4], [2, 3]])
print(ar_1 @ ar_2)
```

Output
```
[[ 7 10]
 [17 24]]
```

```
Explanation:
Element-wise calculation
Position   Dot Product     Result
[0, 0]     (1×3 + 2×2) = 3 + 4   7
[0, 1]     (1×4 + 2×3) = 4 + 6   10
[1, 0]     (3×3 + 4×2) = 9 + 8   17
[1, 1]     (3×4 + 4×3) = 12 + 12 24
```

```
ar_1 = np.array([[1, 2]])
ar_2 = np.array([[2], [3]])
print(ar_1 @ ar_2)
```

Output
```
[[8]]
Matrix multiplication between a (1×2) and (2×1) array gives a
(1×1) result: [[8]].
```

```
ar_1 = np.array([1, 3, 5])
ar_2 = np.array([2, 3, 6])
print(ar_1 @ ar_2)
```

Output
```
41
```

The @ operator is not applicable for the scalar values i.e. 10 @ 20 is an error. Also with the multiplication of the 2-D array it is compulsory that the number of columns in the first matrix should be the same as the number of rows in the second matrix.

```
ar_7 = np.array([[1, 3], [5, 7]])
ar_8 = np.array([[2, 3]])
print(ar_7 @ ar_8)    #Error
```

```
Matrix multiplication A @ B is only valid if:

A.shape = (m, n)
B.shape = (n, p)

That is: columns of A = rows of B
```

What is output of following code

```
ar_7 = np.array([[1, 3], [5, 7]])
ar_8 = np.array([2, 3])
print(ar_7 @ ar_8)
```

(A) [11 31]
(B) Error because the dimension is not same
(C) [31 11]
(D) [11 11]
(E) None of these

RIGHT ANSWER: (A) [11 31] If the first operand is an N-D array and the second operand is a 1-D array, it is a sum product over the last axis of a and b.

NumPy treats the 1D array like a column vector (shape `(2, 1)`) during multiplication.
1st row: `1×2 + 3×3 = 2 + 9 = 11`
2nd row: `5×2 + 7×3 = 10 + 21 = 31`

2. The **dot()** function
   For multiplication of the 1-D arrays and the 2-D arrays it is the same as for the @ operator but you can use dot operator for the scalar values too.

```
a = 10
b = 20
print(np.dot(a, b))
```

Output
```
200
```

3. The **matmul()** function: it works 100% same as for the @

```
ar_9 = np.arange(24).reshape(2, 3, 4)
ar_10 = np.arange(16).reshape(2, 4, 2)
result1 = ar_9 @ ar_10
```

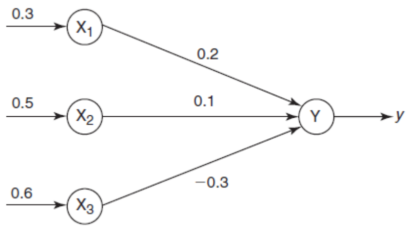
```
result2 = np.matmul(ar_9, ar_10)
print("\nAre they equal?", np.array_equal(result1, result2))
```

Output

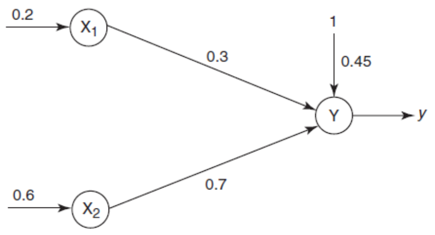
```
Are they equal? True
```

An Example

Q.1 For the network shown in Figure below, calculate the net input to the output neuron.



Q.2 For the network shown in Figure below, calculate the net input to the output neuron.



4. **The multi_dot() function:**

Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order. multi_dot chains numpy.dot and uses optimal parenthesization of the matrices. Depending on the shapes of the matrices, this can speed up the multiplication a lot. If the first argument is 1-D it is treated as a row vector. If the last argument is 1-D it is treated as a column vector. The other arguments must be 2-D.

```
ar_15 = np.array([[1, 2], [3, 4]])
ar_16 = np.array([[5, 6], [7, 8]])
ar_17 = np.array([[1, 0], [0, 1]])
result = np.linalg.multi_dot([ar_15, ar_16, ar_17])
print(result)
```

```
Output
[[19, 22],
[43, 50]]
```

The `result = np.linalg.multi_dot([ar_15, ar_16, ar_17])` can be written as `result = np.dot(np.dot(ar_15, ar_16), ar_17)` but the former is preferred because it will have better performance.

5. The **linalg.cross()** function: Returns the cross product of 3-element vectors. If x1 and/or x2 are multi-dimensional arrays, then the cross-product of each pair of corresponding 3-element vectors is independently computed. Used in 3D graphics, physics, and robotics for perpendicular vectors.

   ```
   a = np.array([1, 0, 0])
   b = np.array([0, 1, 0])
   result = np.linalg.cross(a, b)
   print(result)
   ```

   Output
   ```
   [0, 0, 1]
   ```

   Explanation

   $c_1 = a_2 * b_3 - a_3 * b_2 = 0 * 0 - 0 * 1 = 0$
   $c_2 = a_3 * b_1 - a_1 * b_3 = 0 * 0 - 1 * 0 = 0$
   $c_3 = a_1 * b_2 - a_2 * b_1 = 1 * 1 - 0 * 0 = 1$

   ```
   x = np.array([[1, 2], [3, 4], [5, 6]])
   y = np.array([[4, 5], [6, 1], [2, 3]])
   print(np.linalg.cross(x, y, axis=0))
   ```

   The cross product is computed column-wise because axis=0. Each column is treated as a 3D vector, and the cross product is computed between corresponding columns of x and y.

   Column 0:
   a = [1, 3, 5] (first column of x)
   b = [4, 6, 2] (first column of y)

   Column 1:
   a = [2, 4, 6] (second column of x)
   b = [5, 1, 3] (second column of y)

   What is output of following code
   ```
   print(np.linalg.cross(np.array([1, 0]), np.array([0, 1])))
   ```
   (A) [0, 0, 1]
   (B) 1
   (C) -1
   (D) Error
   (E) None of these

6. The **vdot()** function: It handles complex numbers differently than `dot`. If the first argument is complex, it is replaced by its complex conjugate in the dot product calculation. It also handles multidimensional arrays differently than `dot`: it does not perform a matrix product, but flattens the arguments to 1-D arrays before taking a vector dot product.

$$\mathrm{vdot}(a, b) = \sum_i \overline{a_i} \cdot b_i$$

An Example

```
a = np.array([1+2j, 3+4j])
b = np.array([5+6j, 7+8j])
result = np.vdot(a, b) # (1-2j)*5+6j + (3-4j)*7+8j
print(result)
```

`It` **`flattens`** `both arrays.`

`Then, it computes the` **`complex conjugate(only for complex no.) of the first argument`** `a, and performs element-wise multiplication with` `b, and sums the result.`
`conj(a) = [1-2j, 3-4j]`

Output

```
(70-8j)
```

What is output of following code

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

(A) 28
(B) 36
(C) 30
(D) 32
(E) None of these

Explanation:
`np.vdot(a, b)` **flattens both arrays into 1D**, and then performs the **dot product(real no.)**.

a.flatten() → [1, 4, 5, 6]
b.flatten() → [4, 1, 2, 2]

1×4+4×1+5×2+6×2=4+4+10+12=30

7. The **vecdot()** function: Finds vector dot product of two arrays.
   np.vecdot(a, b, axis=n) computes the dot product of vectors along the specified axis.
   - It does not flatten the arrays.
   - It does not apply complex conjugation (unlike np.vdot)
   - Returns an array with one less dimension than input.

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} \overline{a_i} b_i$$

```
ar_18 = np.array([[1, 2], [3, 4]])
```

```
ar_19 = np.array([[5, 6], [7, 8]])
result = np.vecdot(ar_18, ar_19, axis=1)
print(result)
```

Output
```
[17 53]
```

```
Explanation:
Axis = 1 → compute dot product row-wise (along each row)
```

- `Row 0:` `[1, 2] · [5, 6] = 1×5 + 2×6 = 5 + 12 = 17`
- `Row 1:` `[3, 4] · [7, 8] = 3×7 + 4×8 = 21 + 32 = 53`

Matrix Decompositions

1. The **linarg.qr()** function:
   Performs QR decomposition, splitting a matrix into an orthogonal matrix (Q) and an upper-triangular matrix (R). Used in solving linear systems and least-squares problems.

```
ar_20 = np.array([[1, 2], [3, 4]])
Q, R = np.linalg.qr(ar_20)
print("Q:", Q, "\nR:", R)
```

Output
```
Q: [[-0.31622777 -0.9486833 ]
 [-0.9486833   0.31622777]]
R: [[-3.16227766 -4.42718872]
 [ 0.          -0.63245553]]
```

```
ar_21 = np.array([[1, 1], [1, 2], [1, 3]])
ar_22 = np.array([1, 2, 2])
Q, R = np.linalg.qr(ar_21)
x = np.linalg.solve(R, Q.T @ ar_22)
print(x)
```

Output
```
[0.66666667 0.5       ]
```

```
ar_23 = np.random.rand(100, 100)  # Simulated image
U, s, Vh = np.linalg.svd(ar_23)
k = 10  # Keep top 10 singular values
A_approx = U[:, :k] @ np.diag(s[:k]) @ Vh[:k, :]
print(A_approx.shape)
```

Output

```
(100, 100)
```

Eigenvalues and Eigenvectors

1. The **linalg.eig()** function: Computes eigenvalues and eigenvectors of a square matrix. Used in PCA, stability analysis, and graph algorithms.

```
ar_24 = np.array([[1, 2], [3, 4]])
eigenvalues, eigenvectors = np.linalg.eig(ar_24)
print("Eigenvalues:",     eigenvalues,     "\nEigenvectors:",
eigenvectors)
```

```
Output
Eigenvalues: [-0.37228132  5.37228132]
Eigenvectors: [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

   What happens if you apply `np.linalg.eig` to a non-square matrix?
   (A) Returns partial results
   (B) Raises an error
   (C) Returns complex eigenvalues
   (D) Returns zeros
   (E) None of these

   Answer: (B) Raises an error
   Explanation: `eig` requires a square matrix. Non-square matrices cause a LinAlgError. Options A, C, and D are incorrect as no partial or default results are returned.

2. The **linalg.eigh()** function: Computes eigenvalues and eigenvectors for symmetric or Hermitian matrices. Faster and more stable than eig for such matrices.

```
ar_25 = np.array([[4, 2], [2, 3]])
eigenvalues, eigenvectors = np.linalg.eigh(ar_25)
print("Eigenvalues:", eigenvalues)
```

   Output

```
Eigenvalues: [1.43844719 5.56155281]
```

```
ar_26 = np.array([[1, 1+1j], [1-1j, 2]])
eigenvalues, eigenvectors = np.linalg.eigh(ar_26)
print("Eigenvalues:", eigenvalues)
```

   Output

```
Eigenvalues: [-3.33066907e-16  3.00000000e+00]
```

   Which function is better for a symmetric matrix?
   (A) np.linalg.eig
   (B) np.linalg.eigh

(C) Both are equivalent
(D) Neither

RIGHT ANSWER: (B) np.linalg.eigh
eigh is optimized for symmetric/Hermitian matrices, ensuring real eigenvalues and faster computation. Option A is less efficient, C is incorrect due to performance differences, and D is wrong as eigh is suitable.

3. The **linalg.eigvals()** and the **linalg.eigvalsh()** function: Compute only eigenvalues (no eigenvectors) for general (eigvals) or symmetric/Hermitian (eigvalsh) matrices. Used when eigenvectors are not needed.

   ```
   ar_27 = np.array([[1, 2], [2, 1]])
   print(np.linalg.eigvals(ar_27))
   print(np.linalg.eigvalsh(ar_27))

   Output
   [ 3. -1.]
   [-1.  3.]
   ```

Norms and Matrix Properties

1. The **linalg.norm()** function: Computes the norm of a vector or matrix (e.g., Euclidean, Frobenius). Used in ML for regularization and error measurement.

   ```
   ar_28 = np.array([3, 4])
   norm = np.linalg.norm(ar_28)
   print(norm)
   ```

   Output
   ```
   5.0

   ar_29 = np.array([[1, 2], [3, 4]])
   norm = np.linalg.norm(ar_29, ord='fro')
   print(norm)
   ```

   Output
   ```
   5.477225575051661
   Explanation: sqrt(1^2 + 2^2 + 3^2 + 4^2) ≈ 5.477225575051661
   ```

   What is `np.linalg.norm(np.array([1, 1]), ord=2)`?
   (A) 1
   (B) 2
   (C) 1.414
   (D) Error

   Output
   1.414

2. The **linalg.matrix_norm()** function: Computes matrix-specific norms (e.g., spectral norm). Used in stability analysis and optimization.

```
ar_30 = np.array([[1, 2], [3, 4]])
norm = np.linalg.matrix_norm(ar_30, ord='fro')
print(norm)

Output
5.477

norm = np.linalg.matrix_norm(ar_30, ord=2)
print(norm)

Output
5.464985704219043
```

3. The **linalg.vector_norm()** function: Computes vector-specific norms. Used for vector magnitude in ML.

```
ar_31 = np.array([1, 1])
norm = np.linalg.vector_norm(ar_31, ord=2)
print(norm)
```

Output
```
1.414

norm = np.linalg.vector_norm(ar_31, ord=1)
print(norm)
```

Output
```
2
```

4. The **linalg.det()** function: Computes the determinant of a square matrix. Used to check invertibility or in change-of-variable formulas.

```
ar_32 = np.array([[1, 2], [3, 4]])
det = np.linalg.det(ar_32)
print(det)
```

Output
```
-2.0

ar_33 = np.array([[1, 2, 3], [2, 4, 6], [3, 6, 9]])
det = np.linalg.det(ar_33)
print(det)
```

Output
```
0.0
```

5. The linalg.matrix_rank() function: Computes the rank of a matrix (number of linearly independent rows/columns). Used in feature selection and dimensionality analysis.

   ```
   ar_34 = np.array([[1, 2], [3, 4]])
   rank = np.linalg.matrix_rank(ar_34)
   print(rank)
   ```

   Output
   ```
   2
   ```

   ```
   ar_35 = np.array([[1, 2, 3], [2, 4, 6], [3, 6, 9]])
   rank = np.linalg.matrix_rank(ar_35)
   print(rank)
   ```

   Output
   ```
   1
   ```

6. The **numpy.trace()** function: Computes the sum of diagonal elements of a square matrix. Used in ML for loss functions and matrix analysis.

   ```
   ar_36 = np.array([[1, 2], [3, 4]])
   result = np.trace(ar_36)
   print(result)
   ```

   Output:
   ```
   5
   ```

   ```
   ar_37 = np.array([[1, 0, 0], [0, 2, 0], [0, 0, 3]])
   trace = np.trace(ar_37)
   print(trace)
   ```

   Output
   ```
   6
   ```

7. The **linalg.solve()**: Solves a linear system (Ax = b) for (x). Used in regression and optimization.

   ```
   ar_38 = np.array([[3, 1], [1, 2]])
   ar_39 = np.array([5, 4])
   result = np.linalg.solve(ar_38, ar_39)
   print(result)

   Output: [1., 1.]

   ar_40 = np.array([[1, 1, 1], [0, 2, 3], [0, 0, 1]])
   ar_41 = np.array([6, 8, 1])
   result = np.linalg.solve(ar_40, ar_41)
   ```

```
print(result)

Output: [1., 2., 1.]
```

8. The **linalg.inv()**: Computes the inverse of a square matrix. Used in linear regression and transformation reversal.

   ```
   ar_42 = np.array([[4, 7], [2, 6]])
   inv_ar_42 = np.linalg.inv(ar_42)
   print(inv_ar_42)
   ```

   Output
   ```
   [[ 0.6 -0.7]
    [-0.2  0.4]]
   ```

9. The **linalg.matrix_transpose()** function: Transposes a matrix (swaps rows and columns). Used in data preprocessing and transformations.

   ```
   ar_43 = np.array([[1, 2], [3, 4]])
   trans_ar_43 = np.linalg.matrix_transpose(ar_43)
   print(trans_ar_43)
   ```

   Output:
   ```
   [[1, 3], [2, 4]]
   ```

   ```
   ar_44 = np.array([[1, 2, 3], [4, 5, 6]])
   trans_ar_44 = np.linalg.matrix_transpose(ar_44)
   print(trans_ar_44)
   ```

   Output:
   ```
   [[1, 4], [2, 5], [3, 6]]
   ```