RESEARCH INNOVATION DISCOVERY

RID संस्था समस्या का समाधान

RUN BY TWKSAA WELFARE FOUNDATION

Reg.No: 048884
Foundation Day
05-09-2025

Numpy

E-Book

Er. Rajesh Prasad(B.E, M.E)
Founder: TWF & RID Org.

- **RID ORGANIZATION** यानि **Research, Innovation and Discovery** संस्था जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही मैं राजेश प्रसाद **इस RID संस्था** की स्थपना किया हूँ |
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

"त्वक्सा Numpy के इस ई-पुस्तक में आप कंप्यूटर से जुड़ी सभी बुनियादी अवधारणाएँ सीखेंगे। । मुझे आशा है कि इस ई -पुस्तक को पढ़ने के बाद आपके ज्ञान में वृद्धि होगी और आपको कंप्यूटर विज्ञान के बारे में और अधिक जानने में रुचि होगी"

In this E-Book of Numpy, you will learn all the basic concepts related to computers. I hope after reading this E-Book your knowledge will be improve and you will get more interest to know more thing about computer Science.

RID हमें क्यों करना चाहिए ?

(Research)

अनुसंधान हमें क्यों करना चाहिए ?

Why should we do research?

1. नई ज्ञान की प्राप्ति (Acquisition of new knowledge)
2. समस्याओं का समाधान(To Solving problems)
3. सामाजिक प्रगति (To Social progress)
4. विकास को बढ़ावा देने(To promote development)
5. तकनीकी और व्यापार में उन्नति(To advances in technology & business)
6. देश विज्ञान और प्रौद्योगिकी के विकास(To develop the country's science & technology)

(Innovation)

नवीनीकरण हमें क्यों करना चाहिए ?

Why should we do Innovation?

1. प्रगति के लिए(To progress)
2. परिवर्तन के लिए(For change)
3. उत्पादन में सुधार(To Improvement in production)
4. समाज को लाभ(To Benefit to society)
5. प्रतिस्पर्धा में अग्रणी (To be ahead of competition)
6. देश विज्ञान और प्रौद्योगिकी के विकास।
(To develop the country's science & technology)

(Discovery)

खोज हमें क्यों करना चाहिए?

Why should we do Discovery?

1. नए ज्ञान की प्राप्ति(Acquisition of new knowledge)
2. अविष्कारों की खोज(To Discovery of inventions)
3. समस्याओं का समाधान(To Solving problems)
4. ज्ञान के विकास में योगदान(Contribution to development of knowledge)
5. समाज के उन्नति के लिए (for progress of society)
6. देश विज्ञान और तकनीक के विकास
(To develop the country's science & technology)

New Technology पर **Research** करने के लिए सम्पर्क करें: **ridorg.in@gmail.com Mob.No: 9202707903**

❖ Research(अनुसंधान):

- अनुसंधान एक प्रणालीकरण कार्य होता है जिसमें विशेष विषय या विषय की नई ज्ञान एवं समझ को प्राप्त करने के लिए सिद्धांतिक जांच और अध्ययन किया जाता है। इसकी प्रक्रिया में डेटा का संग्रह और विश्लेषण, निष्कर्ष निकालना और विशेष क्षेत्र में मौजूदा ज्ञान में योगदान किया जाता है। अनुसंधान के माध्यम से विज्ञान, प्रोधोगिकी, चिकित्सा, सामाजिक विज्ञान, मानविकी, और अन्य क्षेत्रों में विकास किया जाता है। अनुसंधान की प्रक्रिया में अनुसंधान प्रश्न या कल्पनाएँ तैयार की जाती हैं, एक अनुसंधान योजना डिज़ाइन की जाती है, डेटा का संग्रह किया जाता है, विश्लेषण किया जाता है, निष्कर्ष निकाला जाता है और परिणामों को उचित दर्शाने के लिए समाप्ति तक पहुंचाया जाता है।

❖ Innovation(नवीनीकरण): -

- Innovation एक विशेषता या नई विचारधारा की उत्पत्ति या नवीनीकरण है। यह नए और आधुनिक विचारों, तकनीकों, उत्पादों, प्रक्रियाओं, सेवाओं या संगठनात्मक ढंगों का सृजन करने की प्रक्रिया है जिससे समस्याओं का समाधान, प्रतिस्पर्धा में अग्रणी होने, और उपयोगकर्ताओं के अनुकूलता में सुधार किया जा सकता है।

❖ Discovery (आविष्कार):

- Discovery का अर्थ होता है "खोज" या "आविष्कार"। यह एक विशेषता है जो किसी नए ज्ञान, अविष्कार, या तत्व की खोज करने की प्रक्रिया को संदर्भित करता है। खोज विज्ञान, इतिहास, भूगोल, तकनीक, या किसी अन्य क्षेत्र में हो सकती है। इस प्रक्रिया में, व्यक्ति या समूह नए और अज्ञात ज्ञान को खोजकर समझने का प्रयास करते हैं और इससे मानव सभ्यता और विज्ञान-तकनीकी के विकास में योगदान देते हैं।

नोट : अनुसंधान विशेषता या विषय पर नई ज्ञान के प्राप्ति के लिए सिस्टमैटिक अध्ययन है, जबकि आविष्कार नए और अज्ञात ज्ञान की खोज है।

सुविचार:	
1.	**समस्याओं का समाधान करने का उतम मार्ग हैं |** → शिक्षा ,RID, प्रतिभा, सहयोग, एकता एवं समाजिक-कार्य
2.	**एक इंसान के लिए जरूरी हैं |** → रोटी, कपड़ा, मकान, शिक्षा, रोजगार, इज्जत और सम्मान
3.	**एक देश के लिए जरूरी हैं |** - → संस्कृति-सभ्यता, भाषा, एकता, आजादी, संविधान एवं अखंडता
4.	**सफलता पाने के लिए होना चाहिए |** → लक्ष्य, त्याग, इच्छा-शक्ति, प्रतिबद्धता, प्रतिभा, एवं सतता
5.	**मरने के बाद इंसान छोडकर जाता हैं |** → शरीर, अन-धन, घर-परिवार, नाम, कर्म एवं विचार
6.	**मरने के बाद इंसान को इस धरती पर याद किया जाता हैं उनके**
→ नाम, काम, दान, विचार, सेवा-समपर्ण एवं कर्मो से...	

आशीर्वाद (बड़े भैया जी)

Mr. RAMASHANKAR KUMAR

मार्गदर्शन एवं सहयोग

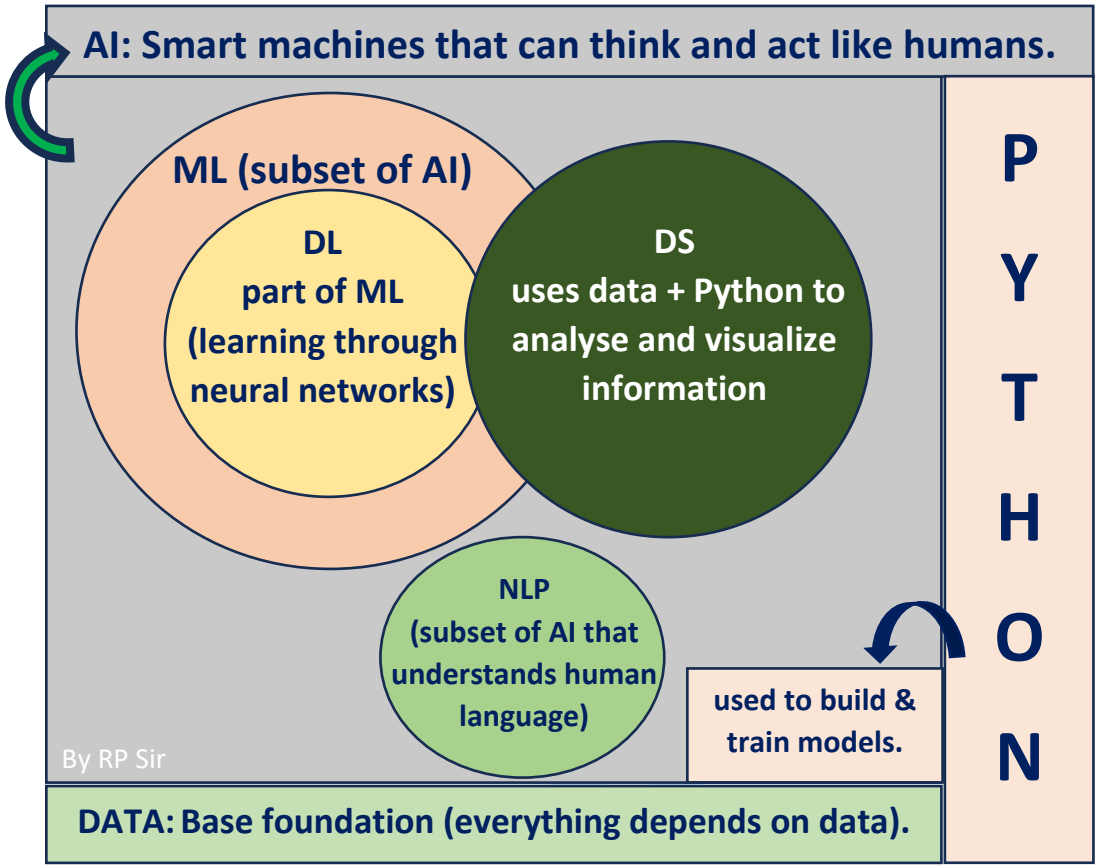
Mr. GAUTAM KUMAR

.....सोच है जिनकी नये कुछ कर दिखाने की, खोज हैं मुझे आप जैसे इंसान की.....

"अगर आप भी **Research, Innovation and Discovery** के क्षेत्र में रूचि रखतें हैं एवं अपनी प्रतिभा से दुनियां को कुछ नया देना चाहतें एवं अपनी समस्या का समाधान **RID** के माध्यम से करना चाहतें हैं तो **RID ORGANIZATION (रीड संस्था)** से जरुर जुड़ें" || धन्यवाद || **Er. Rajesh Prasad (B.E, M.E)**

AI vs ML vs DL vs DS vs NLP vs data vs python

1. **AI (Artificial Intelligence):** Machines that can perform tasks like humans (decision-making, problem-solving, learning).
2. **ML (Machine Learning):** Subset of AI. Uses data and algorithms to analyse patterns and make predictions.
3. **DL (Deep Learning):** Subset of ML. Uses **Neural Networks** to mimic the human brain for learning.
4. **DS (Data Science):** Field that works with data → collecting, cleaning, analysing, visualizing, and building models.
5. **NLP (Natural Language Processing):** Branch of AI/ML that helps machines understand and process human language
6. **Data:** Raw facts or information (numbers, text, images, audio, etc.).
7. **Python:** programming language used to handle data, build models, and develop AI/ML systems.



How to Develop Our Own AI

Module 1: Basics of AI
- Meaning & uses of AI
- AI vs ML vs Deep Learning

Module 2: Python for AI
- Python basics
- NumPy, Pandas, Matplotlib

Module 3: Mathematics for AI
- Linear Algebra
- Probability & Statistics
- Gradient & Optimization (basic)

Module 4: Machine Learning
- ML workflow
- Regression, Classification, Clustering algorithms
- Model training & testing

Module 5: Deep Learning
- Neural Networks
- CNN & RNN
- TensorFlow / PyTorch

Module 6: Build Our Own AI Model
- Data collection & preprocessing
- Training & tuning
- Saving & testing the model

Module 7: Deploy AI
- API using Flask / FastAPI
- Web & Mobile AI Integration
- Cloud deployment

Module 8: Final AI Product
- Build a complete AI application
- Live project with frontend + backend + AI model

Project-1 (RTS MCQ Generator AI)

Phase 1: Foundation

1. Learn Python basics
2. Learn NLP (Natural Language Processing)
3. Learn important libraries
    - NLTK, SpaCy
    - Transformers (HuggingFace)

Phase 2: Dataset Preparation

1. Collect MCQ datasets (English + Hindi)
2. Add theory paragraphs / chapters as input
3. Label / format dataset
4. Input: Paragraph
5. Output: Question + Options + Correct Answer

Phase 3: Model Development Two approaches:

Approach A — Fine-Tune Existing AI Model (Recommended)

- Use models like:
    - ✓ T5 / mT5
    - ✓ BERT / mBERT
    - ✓ GPT-2 / IndicGPT (for Hindi)
- Fine-tune model for:
    - ✓ Question generation
    - ✓ Distractor options generation

Approach B — Custom NLP Pipeline (Simple approach)

- Extract key sentences
- Convert statements → Questions
- Generate wrong options using:
    - ✓ Similar words
    - ✓ TF-IDF similarity
    - ✓ WordNet synonyms / antonyms

Phase 4: Build Bilingual Support

- Detect language automatically or allow user to select
- Use IndicNLP + transformer models for Hindi
- Train bilingual dataset:
- input_text = ["Hindi Paragraph", "English Paragraph"]
- output_mcq = ["MCQ Hindi", "MCQ English"]

Phase 5: Output Format

Final AI must generate:

Q: _____ ?

A) Option 1

B) Option 2

C) Option 3

D) Option 4

Correct Answer: ___

Explanation: ___ (optional)

Phase 6: Deployment

- Build Flask / FastAPI API
- Create simple UI (HTML, CSS, JavaScript)
- Add input box → user pastes paragraph
- Output → MCQ generated (Hindi / English)

Phase 7: Final Product (Goal)

- User selects the language + pastes content (PDF / text / topic) → AI produces:
- 10 / 20 / 50 MCQs
- Export to PDF / Word / Excel
- MCQ Difficulty level → Easy / Medium / Hard

NumPy Syllabus

Topic- 1: Introduction to NumPy

1. What is NumPy
2. Features and Advantages of NumPy
3. Installation and Import (pip install numpy, import numpy as np)
4. Difference between Python Lists and NumPy Arrays
5. Basic Array Creation
    - np.array()
    - np.zeros(), np.ones(), np.full()
    - np.arange(), np.linspace()
    - np.random.rand(), np.random.randint()
6. Array Attributes
    - shape, ndim, size, dtype, itemsize

Practical:

- Create 1D, 2D, 3D arrays
- Check their shape and data type

Topic-2: Array Indexing, Slicing & Iteration

1. Indexing arrays (a[0], a[1,2])
2. Slicing arrays (a[0:3], a[:,1:3])
3. Fancy Indexing (using lists or arrays of indices)
4. Boolean Indexing (a[a > 5])
5. Iterating over arrays using loops
6. Difference between shallow and deep copy (view() vs copy())

Practical:

- Access, modify, and print specific elements
- Extract rows, columns, and subarrays

Topic- 3: Array Operations and Mathematical Functions

1. Arithmetic Operations (+, -, *, /)
2. Broadcasting
3. Universal Functions (ufuncs):
    - np.add(), np.subtract(), np.multiply(), np.divide()
    - np.sqrt(), np.power(), np.exp(), np.log()
4. Trigonometric Functions: np.sin(), np.cos(), np.tan()
5. Rounding, Floor, Ceil functions

Practical:

- Perform matrix addition, subtraction, and multiplication
- Use broadcasting to apply operations on different-sized arrays

Topic-4: Statistical and Aggregation Functions

1. Aggregate functions:
    - np.sum(), np.mean(), np.median(), np.std(), np.var()
2. Min, Max and Argmin/Argmax (np.min(), np.max(), np.argmin(), np.argmax())
3. Axis-wise operations (axis=0 for column, axis=1 for row)

Practical: Calculate mean, median, and standard deviation for given data

Topic-5: Reshaping, Joining, and Splitting Arrays

1. Reshaping arrays (reshape(), ravel(), flatten())
2. Transpose and Axis swapping (.T, np.transpose())

3. Stacking arrays (hstack, vstack, concatenate)
4. Splitting arrays (hsplit, vsplit, split)

Practical:

- Reshape a 1D array into 2D
- Concatenate and split arrays

Topic-6: Random Numbers and Probability

1. np.random.rand(), np.random.randn(), np.random.randint()
2. Setting a seed (np.random.seed())
3. Random choice and shuffling (np.random.choice(), np.random.shuffle())
4. Generating random arrays and matrices

Practical: Generate random datasets for testing

Topic-7: Input and Output Operations

1. Saving and Loading arrays:
    - np.save(), np.load()
    - np.savetxt(), np.loadtxt()
2. Working with .csv files
3. Data conversion between lists and arrays

Practical: Save a NumPy array to file and load it again

Topic-8: Handling Missing Data

1. Representing missing data (np.nan)
2. Checking missing data (np.isnan())
3. Replacing NaN values (np.nan_to_num())
4. Functions ignoring NaN: np.nanmean(), np.nansum()

Practical: Replace NaN values and calculate mean ignoring them

Topic-9: Mini Project Ideas (Practice Section)

1. Create a simple **Matrix Calculator** using NumPy
2. Perform **Statistical Analysis** on sample data
3. Build a **Random Number Generator & Plot Graph**
4. **Normalize a dataset** using NumPy
5. Compute **Simple Interest / Compound Interest** with arrays

❖ What is NumPy:

- NumPy (Numerical Python) is a Python library used for **fast mathematical calculations** on large **arrays and matrices** of numbers.

2. Features and Advantages of NumPy:

- Supports **multi-dimensional arrays**
- Performs **fast mathematical operations**
- Uses **less memory** than Python lists
- Provides many **built-in functions** for math and statistics
- Works well with **pandas**, **matplotlib**, and **machine learning** libraries

❖ How to install the Install NumPy

- pip install numpy

❖ How to check the numpy is install or not

- import numpy as np
- print(np.__version__)

```
import numpy as np
print("NumPy is installed!")
print("Version:", np.__version__)
```

```
NumPy is installed!
Version: 2.1.3
```

❖ What is an Array:

An **array** is a collection of elements (like numbers or data) stored in a **single variable**.
In NumPy, arrays are called **ndarrays** (N-dimensional arrays).

❖ Why We Use Arrays:

- To **store multiple values** together instead of using many variables.
- To perform **fast mathematical operations** on data.
- To work easily with **large datasets** (like in data science or machine learning).
- Arrays make **data processing simple and efficient** compared to Python lists.

Example:

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

Output: [10 20 30 40]

```
#how to create the array in numpy
v=np.array([10,20,30,40,50])
print(v)
print(type(v))
for i in v:
    print(i,end=" ")
```

```
[10 20 30 40 50]
<class 'numpy.ndarray'>
```

Difference between Python List and NumPy Array

Point	Python List	NumPy Array
Data Type	Can store different data types	Stores same data type
Speed	Slower	Faster
Memory	Uses more memory	Uses less memory
Operations	No direct math operations	Supports math operations easily
Use	For general data storage	For numerical and scientific work

Basic Array Creation

1.np.array():- Creates an array from a list or tuple.

- **Use:** To make a normal NumPy array.
- **Syntax:** np.array([elements])
- **Example:**

```
import numpy as np
a = np.array([1, 2, 3])
print(a)    # [1 2 3]
```

- **np** → stands for **NumPy**, a Python library for numerical operations.
- **array()** → a **function** in NumPy used to create an array.
- **[elements]** → **data or values** (like numbers) you put inside the array.

❖ **Inside array(), we can pass only List, Tuple, Nested list (for 2D array) or same data type**

Example-1:

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a)
```

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

Example-2:

```
a = np.array([[1, 2, 3,"skills",]])    # Error reason
```

NumPy Rule all rows must have same length & same datatype

```
a = np.array([[1, 2, 3, "skills"], [4, 5, 6, "skills"]])  → print(a)
```

Note: inside list we can different data types but inside array we can pass different data.

Example-3: a = np.array([1, 2, 3,"skills"]) → print(a)

Example-4: a = np.array([1, 2, 3,"skills"],("hi")) → Error because we can pass different Data type

Limitation: All elements must be of the **same data type**. Cannot store mixed data types like lists can.

2. np.zeros(), np.ones(), np.full()

- Create arrays filled with 0s, 1s, or a specific number.it is Use To make fixed-value arrays.

2.1: np.zeros()

- **Syntax:** np.zeros(shape)
  - **np →** stands for NumPy, a Python library for numerical operations.
  - **zeros() →** function used to create an array filled with zeros.
  - **shape →** defines the size or dimensions of the array.

Note: Array filled means all positions in the array contain the **same value** (like all 0s, all 1s, or any number you choose).

Example-1:)

```
import numpy as np
a = np.zeros(3)
print(a)
```

→ **Output:** [0. 0. 0.] → all elements are **0**

2.2: np.ones(shape)

- **np →** stands for NumPy.
- **ones() →** function used to create an array filled with ones.
- **shape** → defines how many elements or rows/columns the array will have.

Note: Array filled means every element in the array has the same value.

Example:

```
a = np.ones(4)
print(a)
```

→ **Output:** [1. 1. 1. 1.] → all elements are **1**

2.3: np.full(shape, value) → : np.full(5, 9)

- **np** → stands for NumPy.
- **full()** → function used to create an array filled with a specific value.
- **shape** → defines the array's size.
- **value** → the number that fills all elements of the array.

Example:

```
a = np.full(5, 9)
print(a)
```

→ **Output:** [9 9 9 9 9] → all elements are **9**

Limitation: Only creates arrays with **same values**. Not useful for dynamic or real data.

3.np.arange(), np.linspace() :- Create arrays with a range of numbers.

Syntax:

- np.arange(start, stop, step)
- np.linspace(start, stop, num)

```
r = np.arange(0, 10, 2)
l = np.linspace(0, 1, 5)
print(r)
print(l)
```

```
[0 2 4 6 8]
[0.   0.25 0.5  0.75 1.  ]
```

➢ **np.arange(start, stop, step)**

- Creates values **by skipping with a fixed step**.
- **Stop value is not included**.
- You decide **how much gap** between numbers.
- Example: np.arange(0, 10, 2) → Output: [0 2 4 6 8]

➢ **np.linspace(start, stop, num)**

- Creates values **by dividing into a fixed number of parts**.
- **Stop value is included**.
- You decide **how many numbers you want**.
- Example: np.linspace(1, 5, 5) → Output: [1. 2. 3. 4. 5.]

Example: r = np.arange(0, 10, 2) or l = np.linspace(0, 1, 5)

Limitation:

- np.arange() may give **inaccurate float values**.
- np.linspace() is slower for very large arrays.

4.np.random.rand(), np.random.randint() :- Create arrays with random numbers.

Syntax:

- np.random.rand(rows, cols) → float numbers
- np.random.randint(low, high, size) → integers

Example:

randf = np.random.rand(2,3)
randi = np.random.randint(1,10,5)

```
randf = np.random.rand(5,2)
randi = np.random.randint(1,10,8)
print(randf)
print(randi)
```

```
[[0.85157795 0.23919502]
 [0.77373914 0.0089123 ]
 [0.49805062 0.69628099]
 [0.9838798  0.31189457]
 [0.30506976 0.34225046]]
[7 2 9 2 3 3 6 2]
```

Limitation:

- Results change every time (not fixed).
- Not suitable where **exact or repeatable results** are needed (unless you use np.random.seed()).

Array Attributes

NumPy arrays have some important properties called **attributes**. These attributes help you understand the structure and details of the array.

1) shape :- Tells the **dimensions** of the array. Shows how many rows and columns it has.

- **Example:** (3, 4) means 3 rows and 4 columns.

2) ndim :- Shows the **number of dimensions** of the array.

Example: 1 → one-dimensional array, 2 → two-dimensional array.

3) size:- Total **number of elements** in the array.

Example: If array shape is (3, 4), size = 3 × 4 = 12.

4) dtype:- Data type of array elements.

Example: int32, float64, bool, etc.

5) itemsize Size (in **bytes**) of each element in the array.

Example: int32 → uses 4 bytes per element.

Example-1

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)        # (2, 3)
print(arr.ndim)         # 2
print(arr.size)         # 6
print(arr.dtype)        # int64 (depends on system)
print(arr.itemsize)     # size of each element in bytes
```

Property	Meaning	Example Output	Explanation
shape	Rows & Columns	(2, 3)	2 rows × 3 columns
ndim	Number of dimensions	2	2D array
size	Total number of elements	6	2 × 3 = 6
dtype	Data type	int64	All elements are integers
itemsize	Memory used by each element	8 bytes	Each integer takes 8 bytes

What does (2, 3) mean in shape?

(2, 3)
↑ ↑
| └── 3 columns
└──── 2 rows

Visual view of the array:

Row 1 → [1, 2, 3]
Row 2 → [4, 5, 6]

❖ Practical: Create 1D, 2D, 3D Arrays and Check Their Shape & Data Type

1. Create a 1D Array

```
import numpy as np
arr1 = np.array([10, 20, 30, 40])
print("1D Array:", arr1)
print("Shape:", arr1.shape)
print("Data Type:", arr1.dtype)
```

2. Create a 2D Array

```
arr2 = np.array([[1, 2, 3],
                 [4, 5, 6]])
print("\n2D Array:\n", arr2)
print("Shape:", arr2.shape)
print("Data Type:", arr2.dtype)
```

3. Create a 3D Array

```
arr3 = np.array([
                 [[1, 2], [3, 4]],
                 [[5, 6], [7, 8]] ])
print("\n3D Array:\n", arr3)
print("Shape:", arr3.shape)
print("Data Type:", arr3.dtype)
```

```
arr3 = np.array([
                 [[1, 2], [3, 4]],
                 [[5, 6], [7, 8]] ])
print("\n3D Array:\n", arr3)
print("Shape:", arr3.shape)
print("Data Type:", arr3.dtype)
```

```
3D Array:
 [[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
Shape: (2, 2, 2)
Data Type: int64
```

Output Explanation

Array Type	Example Shape	Meaning
1D	(4,)	4 elements in a single row
2D	(2, 3)	2 rows × 3 columns
3D	(2, 2, 2)	2 blocks, each having 2×2 matrix

❖ Real Life Use of 1D, 2D & 3D Arrays

Array Type	Real-Life Use	Example
1D Array	Linear data storage	Marks of students: [80, 76, 90, 82]
2D Array	Tabular data / matrices	Excel table: rows = students, columns = subjects
3D Array	Images, videos, deep learning	Image has height × width × RGB channels (e.g., 256 × 256 × 3)

❖ Real-Life Example in Industry

Field	Which Array Used?	Purpose
Data Science / ML	2D Arrays	Dataset in rows & columns
Image Processing	3D Arrays	CNN model reads images
Medical	3D Arrays	MRI scan images stored as pixel blocks
Finance	1D / 2D Arrays	Stock price time-series and portfolios
Engineering	2D Arrays	Matrix calculations for design & simulation
Weather Forecasting	3D Arrays	Temperature data (location × day × time)

Topic- 2 Array Indexing, Slicing & Iteration

➢ **These NumPy concepts are used in:**

- ✓ Machine Learning
- ✓ Data Science
- ✓ Image Processing
- ✓ IoT and Sensor Data
- ✓ Financial data analysis
- ✓ Scientific research
- ✓ Big data preprocessing

❖ Where We Use These Concepts in Real Projects

1. Indexing (a[0], a[1,2]) Use: To pick specific data values.

Real example:

- Getting the pixel value at position (x, y) in image processing
- Selecting a student's marks from a table
- Extracting a sensor reading from IoT data

2. Slicing (a[0:3], a[:,1:3]) Use: To take a part of the dataset.

Real example:

- Taking first 100 rows from a CSV file
- Cutting a portion of an image
- Selecting specific columns (like age, salary) in ML preprocessing

3. Fancy Indexing Use: To pick multiple specific positions at once.

Real example:

- Selecting top 10 highest salary employees
- Choosing specific image channels (R-G-B)
- Picking selected students (roll numbers) from a big array

4. Boolean Indexing Use: Filtering data based on condition.

Real example:

- Select customers whose age > 30
- Find all pixels brighter than a threshold
- Filter rows in ML dataset where marks > 80

5. Iteration (Looping Over Arrays) Use: Processing each element step-by-step.

Real example:

- Applying formulas to each pixel in image
- Applying tax/discount on all prices
- Calculating new values for each sensor reading

6. Shallow Copy vs Deep Copy (view() vs copy()) Use: Memory management and performance.

Real example:

- Avoid accidental modification of original dataset
- Creating a backup of image/array before editing
- Working on large ML datasets without wasting memory

1. Indexing Arrays

- Indexing means accessing individual elements of a NumPy array using their position.
- NumPy supports:
  - **Positive indexing** → from start (0,1,2…)
  - **Negative indexing** → from end (-1, -2…)
  - **Multi-dimensional indexing** → (row, column)

Syntax a[index]

```
a[row, col]
a[-1]          # negative index
```

Examples

1D Array

```
a = np.array([10, 20, 30, 40])
print(a[0])   # 10
print(a[-1])  # 40 (last element)
```

2D Array

```
b = np.array([[5,6,7],
              [8,9,10]])
print(b[1, 2])   # 10 → row 1, col 2
print(b[-1, -1])  # 10 → last row, last col
```

2. Slicing Arrays Slicing is used to select a **range of elements**.

- **Format: start : end : step**
  - **start** → starting index
  - **end** → stops before this index
  - **step** → gap between elements
  - Works for both 1D and multi-dimensional arrays.

Syntax a[start:end]

```
a[start:end:step]
a[:, 1:3]    # all rows, columns 1 and 2
```

Examples

1D Slicing

```
a = np.array([1,2,3,4,5])
print(a[1:4])      # [2 3 4]
print(a[::-1])     # reverse array
```

2D Slicing

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

print(b[0:2, 1:3])  # [[2 3]
                    #  [5 6]]

print(b[:, 0])      # first column → [1 4 7]
```

3. Fancy Indexing

- Fancy indexing means selecting elements using **lists or arrays** of indices.
- You can choose **specific positions**, not ranges.

Syntax a[[i1, i2, i3]]

```
a[[rowList], [colList]]
```

Examples

1D Fancy Indexing

```
a = np.array([10, 20, 30, 40, 50])
print(a[[0, 2, 4]])      # [10 30 50]
```

2D Fancy Indexing

```
b = np.array([[11,12,13],
              [14,15,16],
              [17,18,19]])
print(b[[0,2], [1,2]])   # [12 19]
```

(0,1) and (2,2) elements selected.

4. Boolean Indexing

- Boolean indexing selects elements using **True/False conditions**.
- It is used for:
- Filtering data
- Selecting elements that meet a condition
- Applying masks

Syntax

```
a[a > value]
a[a % 2 == 0]     # even numbers
```

Examples

```
a = np.array([3,6,9,12,15])
print(a[a > 8])      # [ 9 12 15]
print(a[a % 3 == 0])  # all multiples of 3
```

- Boolean indexing returns a **new filtered array**.

5. Iterating Over Arrays: - Iteration means looping through array elements.

Ways to iterate:

- for loop for 1D & 2D
- np.nditer() to iterate element-by-element

Syntax

```
for x in array:
   print(x)
for row in array2D:
   print(row)
```

Examples

1D Iteration

```
a = np.array([1,2,3])
for x in a:
   print(x)
```

2D Iteration

```
b = np.array([[1,2],[3,4]])
for row in b:
   print(row)
```

Element-by-element

```
for x in np.nditer(b):
   print(x)
```

6. Shallow Copy vs Deep Copy (view() vs copy())

NumPy arrays can be copied in two ways:

- **Shallow Copy → view()**
  - Creates a **new array object**, but **shares the same data**
  - Changing one affects the other

Syntax: b = a.view()

Example

```
a = np.array([1,2,3])
b = a.view()
b[0] = 100
print(a)   # [100 2 3] → changed
```

- **Deep Copy → copy()**
  - Creates a **completely new array**
  - Does **not** share data
  - Changes do not affect original

Syntax: c = a.copy()

Example

```
a = np.array([1,2,3])
c = a.copy()
c[1] = 200
print(a)   # [1 2 3] → unchanged
```

Important Points to Remember

- NumPy supports **negative indexing**.
- Slicing end index is **excluded**.
- Boolean indexing always returns a **filtered new array**.
- Fancy indexing uses lists/arrays of indices.
- view() shares data; copy() does not.
- Iteration through nditer() gives element-by-element access.

Complete 3D NumPy Array Example

import numpy as np

1. Creating a 3D array

```
a = np.array([
        [[1, 2, 3],
         [4, 5, 6]],
        [[7, 8, 9],
         [10,11,12]]
    ])
print("3D Array:")
print(a)
```

2. Shape of the array

```
print("\nShape of array:", a.shape)
```

(2 layers, 2 rows per layer, 3 columns per row)

3. Indexing (accessing single elements)

```
print("\nAccess elements:")
print("a[0, 0, 1] =", a[0, 0, 1])   # 2
print("a[1, 1, 2] =", a[1, 1, 2])   # 12
```

4. Slicing

```
print("\nSlicing examples:") # Get full layer 0
print("Layer 0:\n", a[0, :, :])
# Get all layers, first row
print("\nAll layers, first row:\n", a[:, 0, :])
# Get all layers, second column
print("\nAll layers, column 1:\n", a[:, :, 1])
```

5. Iterating (looping)

```
print("\nIterating through 3D array:")
for layer in a:
   print("Layer:")
   print(layer)
print("\nElement-by-element:")
for x in np.nditer(a):
   print(x, end=" ")
```

Output (understanding-wise)

```
3D Array:
[[[ 1  2  3]
  [ 4  5  6]]
 [[ 7  8  9]
  [10 11 12]]]
Shape of array: (2, 2, 3)
Access elements:
a[0, 0, 1] = 2
a[1, 1, 2] = 12
Slicing examples:
Layer 0:
[[1 2 3]
 [4 5 6]]
All layers, first row:
[[1 2 3]
 [7 8 9]]
All layers, column 1:
[[ 2  5]
 [ 8 11]]
Iterating through 3D array:
Layer:
[[1 2 3]
 [4 5 6]]
Layer:
[[ 7  8  9]
 [10 11 12]]
Element-by-element:
1 2 3 4 5 6 7 8 9 10 11 12
```

Topic- 3

Array Operations & Mathematical Functions

❖ **Arithmetic Operations (+, -, *, /,//).**

Example:

```
import numpy as np
a = np.array([10, 20, 30])
b = np.array([1, 2, 3])
print("Add:", a + b)
print("Subtract:", a - b)
print("Multiply:", a * b)
print("Divide:", a / b)
print("int value:",a // b)
```

Output:
Add: [11 22 33]
Subtract: [9 18 27]
Multiply: [10 40 90]
Divide: [10. 10. 10.]
int value: [10 10 10]

```
a = np.array([10, 20, 30])
b = np.array([1, 2, 3,400])
```

- ValueError: operands could not be broadcast together with shapes (3,) (4,)
- Note:- **NumPy cannot broadcast 3 elements with 4 elements**.
- **NumPy can only add/multiply arrays if their shapes match(same)**

Note: Why output shows 10. instead of 10? Because **division (/) in NumPy always gives a float**, even when the answer is a whole number. So 10 becomes 10.0 **If you want integer result** Use integer division //:

Real-Life Use of Arithmetic Operations

- **+** → Add tax, increase prices **−** → Stock left, price difference
- ***** → Total bill (price × qty) **/** → Percentage, normalize data
- **//** → Batch grouping (items per batch)

Broadcasting

Broadcasting is a NumPy feature that **automatically expands smaller arrays to match larger array shapes during arithmetic operations**.

Example:

```
a = np.array([10, 20, 30])
b = 5    # single number
print(a + b)
```

Output: [15 25 35]

Note: NumPy automatically added **5 to every element**. This is called **broadcasting**.

Broadcasting – Real Life Example

Example 2: Increase product prices by 10% (real-life project use)

- In e-commerce or billing systems, we often need to **increase all product prices** by a fixed percentage. Broadcasting makes this very easy.

```
import numpy as np
prices = np.array([1200, 850, 600, 1500])  # product prices
increase = 0.10  # 10% increase
new_prices = prices + (prices * increase)
print(new_prices)
```

Output: [1320. 935. 660. 1650.]

→ Broadcasting automatically applies the **10% increase** to the entire price list without loops.

➢ **Where used in real-life?**

- E-commerce websites, supermarket billing systems, GST/tax calculation, discount calculation, etc.

Task 1: Add GST to product prices

```
import numpy as np
prices = np.array([100, 250, 300])
gst = 18
final_price = prices + (prices * gst / 100)
print(final_price)
```

Task 2: Find how many items fit in each box

```
items = np.array([120, 95, 210])
box_capacity = 50
boxes = items // box_capacity
print(boxes)
```

Task 3: Calculate discount price

```
mrp = np.array([500, 800, 1200])
discount = 0.20
final_price = mrp - (mrp * discount)
print(final_price)
```

Task 4: Normalize student marks

```
marks = np.array([45, 78, 90, 66])
normalized = marks / 100
print(normalized)
```

Task 5: Increase image brightness

```
image = np.array([[50, 100],
                  [80, 120]])
bright_image = image + 20
print(bright_image)
```

Broadcasting – Real-Life Example with 2D Array

Example 3: Increasing Image Brightness (Image Processing Project)

In image processing, an image is stored as a **2D NumPy array** (grayscale) or **3D array** (RGB). To increase brightness, we simply add a constant value using broadcasting.

```
import numpy as np
# 2D grayscale image (3×3 pixels)
image = np.array([
   [50, 80, 120],
   [90, 100, 150],
   [70, 60, 110]
])
brightness = 30   # increase brightness by 30 units
bright_image = image + brightness
print(bright_image)
```

Output:

```
[[ 80 110 150]
 [120 130 180]
 [100  90 140]]
```

Where used in real-life?

➞ Photo editing software (brightness, contrast) Computer vision models (preprocessing)

➞ Face detection, license plate detection Mobile camera filters (e.g., Instagram, Snapchat)

Broadcasting Example with 3D Array (RGB Image)

In an RGB image, each pixel has **3 values** → (Red, Green, Blue). So the image shape becomes **(height, width, 3)** → a **3D NumPy array**. We can increase brightness, color intensity, etc., using broadcasting.

Example: Increase Brightness of an RGB Image

```
import numpy as np
image = np.array([     # 3×3 RGB image (each pixel = [R, G, B])
   [[120, 80, 60], [200, 150, 100], [90, 40, 30]],
   [[100, 70, 50], [180, 120, 90],  [60, 30, 20]],
   [[140, 100, 80], [220, 180, 130], [110, 70, 60]]
])
brightness = np.array([20, 20, 20])   # add 20 to R, G, B
bright_image = image + brightness
print(bright_image)
```

Output:

```
[[[140 100  80]
  [220 170 120]
  [110  60  50]]

 [[120  90  70]
  [200 140 110]
  [ 80  50  40]]

 [[160 120 100]
  [240 200 150]
  [130  90  80]]]
```

How broadcasting works here?

- image → shape **(3, 3, 3)**
- brightness → shape **(3,)**

NumPy expands (3,) → (1, 1, 3) and adds it to every pixel.

Use:- Mobile camera filters. Instagram/Snapchat brightness effects, Computer vision preprocessing, Increasing or decreasing RGB values

Universal Functions (ufuncs)

These are **fast, element-wise mathematical functions in NumPy**.

1. **np.add(a, b)** → Adds corresponding elements of arrays.
2. **np.subtract(a, b)** → Subtracts elements of second array from first.
3. **np.multiply(a, b)** → Multiplies corresponding elements.
4. **np.divide(a, b)** → Divides elements (returns float).
5. **np.floor_divide(a, b)** → Integer division (a // b).
6. **np.mod(a, b)** → Returns remainder of division.
7. **np.power(a, b)** → Raises elements of a to the power of b.
8. **np.sqrt(a)** → Square root of each element.
9. **np.exp(a)** → Exponential of each element (e^a).
10. **np.log(a)** → Natural logarithm of each element.
11. **np.sin(a)** → Sine of each element (in radians).
12. **np.cos(a)** → Cosine of each element.
13. **np.tan(a)** → Tangent of each element.
14. **np.abs(a)** → Absolute value of each element.
15. **np.round(a)** → Rounds each element to nearest int.

Basic ufuncs

```
import numpy as np
a = np.array([5, 10, 15])
print(np.add(a, 5))          # a + 5
print(np.subtract(a, 2))     # a - 2
print(np.multiply(a, 3))     # a * 3
print(np.divide(a, 5))       # a / 5
print(np.floor_divide(a, 5)) # a // 5
```

Output:

```
[10 15 20]
[ 3  8 13]
[15 30 45]
[1. 2. 3.]
[1 2 3]
```

What are Scalars?

A **scalar** is a **single number**, not a list or array.

Examples of scalars:

```
x = 5      # integer scalar  y = 3.14    # float scalar
z = -10    # negative scalar
```

- Scalars are **single values**.
- Arrays are **collections of values**, e.g., [1, 2, 3].

Example:

```
import numpy as np
a = np.array([1, 2, 3])  # array
b = 5                    # scalar
print(a + b)  # broadcasting adds scalar to every element
# Output: [6 7 8]
```

- **Scalar = single number**
- **Array = multiple numbers**
- NumPy ufuncs can work on **both scalars and arrays**.

Sample array

```
import numpy as np
a = np.array([4, 9, 16, 25])
print("Add 5:", np.add(a, 5))                          # [ 9 14 21 30]          # Addition
print("Subtract 2:", np.subtract(a, 2))                # [ 2  7 14 23]          # Subtraction
print("Multiply by 3:", np.multiply(a, 3))             # [12 27 48 75]          # Multiplication
print("Divide by 2:", np.divide(a, 2))                 # [ 2.   4.5  8.  12.5]  #  Division
print("Floor Divide by 2:", np.floor_divide(a, 2))     # [2 4 8 12]             #Floor Division
print("Remainder when divided by 5:", np.mod(a, 5))    # [4 4 1 0] # Modulus
print("Square:", np.power(a, 2))                       # [ 16  81 256 625]      # Power
print("Square root:", np.sqrt(a))                      # [2. 3. 4. 5.]          # Square Root
print("Exponential:", np.exp([1, 2, 3]))               # [ 2.71828183  7.3890561  20.08553692]  # Exponential
print("Natural log:", np.log(a))                       # [1.38629436 2.19722458 2.77258872 3.21887582]
# Trigonometric Functions
angles = np.array([0, np.pi/2, np.pi])
print("Sine:", np.sin(angles))                         # [0. 1. 0.]
print("Cosine:", np.cos(angles))                       # [ 1.  0. -1.]
print("Tangent:", np.tan([0, np.pi/4]))                # [0. 1.]
# Absolute value
b = np.array([-10, -5, 0, 5])
print("Absolute:", np.abs(b))                          # [10 5 0 5]
```

❖ NumPy Rounding Functions:

1. **np.round(a)** → Rounds each element to the nearest integer.
2. **np.floor(a)** → Rounds each element **down** to the nearest integer.
3. **np.ceil(a)** → Rounds each element **up** to the nearest integer.

Uses of NumPy Rounding Functions

1. **np.round(a)** → Used in **grading marks**, **financial calculations**, or **displaying averages**.
2. **np.floor(a)** → Used in **calculating complete batches**, **stock rounding down**, or **flooring prices**.
3. **np.ceil(a)** → Used in **ticket booking**, **rounding up quantities**, or **ceil of total bill**.

Example-1:

```
nums = np.array([2.1, 2.5, 2.9])
print("Round:", np.round(nums))   # Round: [2. 2. 3.]
print("Floor:", np.floor(nums))  # lowest integer   Floor: [2. 2. 2.]
print("Ceil:", np.ceil(nums))    # highest integer   Ceil:  [3. 3. 3.]
```

Example-2:

```
import numpy as np
values = np.array([5.3, 7.8, 9.1, 12.6])
print("Original:", values)                  # Original: [ 5.3  7.8  9.1 12.6]
print("Round :", np.round(values))          # nearest integer  Round : [ 5.  8.  9. 13.]
print("Floor :", np.floor(values))          # lower integer  Floor : [ 5.  7.  9. 12.]
print("Ceil  :", np.ceil(values))           # upper integer Ceil  : [ 6.  8. 10. 13.]
```

Topic- 4
Statistical and Aggregation Functions

- NumPy provides **built-in functions** to perform **mathematical and statistical operations** on arrays. These are very useful in **data analysis, machine learning, and scientific computations**.
- **Statistical operations** mean the **basic methods used to analyze and understand data**, like average, highest–lowest, most common value, and spread of numbers.

❖ **Aggregate Functions** : this function perform **overall operations** on all elements of an array.

Function	Description
np.sum()	:- Sum of all elements
np.mean()	:- Average of elements
np.median()	:- Middle value of sorted array
np.std()	:- Standard deviation (spread of data)
np.var()	:- Variance (square of std deviation)
np.min()	:- Minimum value
np.max()	:- Maximum value
np.argmin()	:- Index of minimum value
np.argmax()	:- Index of maximum value

np.sum()

this is a NumPy function used to **add all elements** of an array.

- **axis=None** → Adds *all* elements (default)
- **axis=0** → Column-wise sum (top to bottom)
- **axis=1** → Row-wise sum (left to right)

Syntax: np.sum(array, axis=None) → Works for **1D, 2D, and 3D arrays**

Examples of np.sum() → axis is a fixed keyword — you must use the word axis only.

1) Sum of a 1D Array

```
import numpy as np
arr = np.array([10, 20, 30])
print("Sum:", np.sum(arr))                          # 60
```

2) Sum of a 2D Array (All Elements)

```
matrix = np.array([[1, 2, 3],  [4, 5, 6]])
print("Total Sum:", np.sum(matrix))                 # 21
```

3) Column-wise Sum (axis=0)

```
print("Column-wise Sum:", np.sum(matrix, axis=0))   # Output: [5 7 9]
```

4) Row-wise Sum (axis=1)

```
print("Row-wise Sum:", np.sum(matrix, axis=1))      # Output: [ 6 15]
```

5) Sum of a 3D Array

```
arr3d = np.array([
   [[1, 2], [3, 4]],
   [[5, 6], [7, 8]] ])
print("3D Total Sum:", np.sum(arr3d))               # 36
```

6) 3D Array Sum Across Axis=0

```
print("Axis 0 Sum:", np.sum(arr3d, axis=0))         # [[ 6  8]  #  [10 12]]
```

7) 3D Array Sum Across Axis=1

```
print("Axis 1 Sum:", np.sum(arr3d, axis=1))         # [[ 4  6]  #  [12 14]]
```

8) 3D Array Sum Across Axis=2

```
print("Axis 2 Sum:", np.sum(arr3d, axis=2))         # [[ 3  7]  #  [11 15]]
```

2.np.mean()

This is a NumPy function used to **calculate the average value** of elements in an array.
It adds all values and divides by the total number of elements.

- If **axis=None** → mean of *all* elements
- If **axis=0** → mean **column-wise**
- If **axis=1** → mean **row-wise**

Syntax: np.mean(array, axis=None) → It works on **1D, 2D, and 3D arrays**.

1) Mean of a 1D Array (All Elements)

```
import numpy as np
data = np.array([10, 20, 30, 40])
print("Mean:", np.mean(data))                    # 25.0
```

2) Mean of a 2D Array (All Elements)

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print("Mean:", np.mean(matrix))                  # 3.5
```

3) Mean of Each Column (axis=0)

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print("Column-wise Mean:", np.mean(matrix, axis=0))   # [2.5 3.5 4.5]
```

4) Mean of Each Row (axis=1)

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print("Row-wise Mean:", np.mean(matrix, axis=1))      # [2. 5.]
```

5) Mean of a 3D Array

```
arr3d = np.array([
   [[1, 2], [3, 4]],
   [[5, 6], [7, 8]] ])
print("Mean:", np.mean(arr3d))                   # 4.5
```

6) Mean of 3D Array Across Axis=0 (Depth-wise Mean)

```
print("Axis 0 Mean:", np.mean(arr3d, axis=0))    # [[3. 4.]  #  [5. 6.]]
```

7) Mean of 3D Array Across Axis=1 (Row-wise Mean)

```
print("Axis 1 Mean:", np.mean(arr3d, axis=1))    # [[2. 3.]  #  [6. 7.]]
```

8) Mean of 3D Array Across Axis=2 (Column-wise Mean)

```
print("Axis 2 Mean:", np.mean(arr3d, axis=2))    # [[1.5 3.5]  #  [5.5 7.5]]
```

3.np.median()

np.median() returns the **middle value** of a sorted array.
If the array has even number of elements → it returns the **average of the two middle values**.

Syntax: np.median(array)

Examples

1) Median of 1D Array

```
import numpy as np
data = np.array([10, 20, 30, 40])
print("Median:", np.median(data))   # 25.0
```

```
import numpy as np
data = np.array([10, 20, 30, 40,60])
print("Median:", np.median(data))   # 30.0
```

2) Median of 2D Array (All Elements)

```
matrix = np.array([[1, 5, 3], [7, 2, 9]])
print("Median:", np.median(matrix))   # 4.0
```

3) Row-wise Median (axis=1)

```
print("Row-wise Median:", np.median(matrix, axis=1))   # [3. 7.]
```

4) Column-wise Median (axis=0)

```
print("Column-wise Median:", np.median(matrix, axis=0))   # [4. 3. 6.]
```

4. np.std() — Standard Deviation

Standard deviation measures **how spread out the data is** from the mean.

- Low std → values are close to mean
- High std → values are spread far out

Syntax: np.std(array)

Examples

1) STD of 1D Array

```
data = np.array([10, 20, 30, 40])
print("Standard Deviation:", np.std(data))                    # 11.1803
```

2) STD of 2D Array

```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])
print("STD:", np.std(matrix))                                 # 1.7078
```

3) Row-wise STD

```
print("Row-wise STD:", np.std(matrix, axis=1))                # [0.816 0.816]
```

4) Column-wise STD

```
print("Column-wise STD:", np.std(matrix, axis=0))             # [1.5 1.5 1.5]
```

5.np.var() — Variance

Variance shows **how much the values differ from the mean**.

It is the **square of standard deviation**.

[\text{Variance} = (\text{Standard Deviation})^2]

Syntax np.var(array)

Examples

1) Variance of 1D Array

```
data = np.array([10, 20, 30, 40])
print("Variance:", np.var(data))   # 125.0
```

2) Variance of 2D Array

```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])
print("Variance:", np.var(matrix))   # 2.9166
```

3) Row-wise Variance

```
print("Row-wise Variance:", np.var(matrix, axis=1))
# [0.666 0.666]
```

4) Column-wise Variance

```
print("Column-wise Variance:", np.var(matrix, axis=0))
# [2.25 2.25 2.25]
```

Min, Max, Argmin, Argmax

1) np.min() :- Finds the **minimum (lowest)** value in the array.

Syntax: np.min(array, axis=None)

Examples:

```
arr = np.array([4, 1, 9])
print(np.min(arr))                  # 1
mat = np.array([[3, 8], [5, 2]])
print(np.min(mat))                  # 2
print(np.min(mat, axis=0))          # [3 2]
print(np.min(mat, axis=1))          # [3 2]
```

2) np.max():- Finds the **maximum (highest)** value in the array.

Syntax: np.max(array, axis=None)

Examples:

```
print(np.max(arr))                  # 9
print(np.max(mat))                  # 8
print(np.max(mat, axis=0))          # [5 8]
print(np.max(mat, axis=1))          # [8 5]
```

3) np.argmin() :- Returns the **position (index)** of the smallest value.

Syntax: np.argmin(array, axis=None)

Examples:

```
print(np.argmin(arr))          # index of 1 → 1
print(np.argmin(mat))          # flattened index → 3
print(np.argmin(mat, axis=0))  # [0 1]
print(np.argmin(mat, axis=1))  # [1 1]
```

4) np.argmax():- Returns the **position (index)** of the biggest value.

Syntax: np.argmax(array, axis=None)

Examples:

```
print(np.argmax(arr))          # index of 9 → 2
print(np.argmax(mat))          # index of 8 → 1
print(np.argmax(mat, axis=0))  # [1 0]
print(np.argmax(mat, axis=1))  # [1 0]
```

1) 1D Array Example

```
import numpy as np
arr1 = np.array([10, 5, 30, 2])
print("Min:", np.min(arr1))       # 2
print("Max:", np.max(arr1))       # 30
print("Argmin:", np.argmin(arr1))  # index of 2 → 3
print("Argmax:", np.argmax(arr1))  # index of 30 → 2
```

2) 2D Array Example

```
arr2 = np.array([[3, 8, 1],
                 [6, 4, 9]])
```

All elements

```
print("Min:", np.min(arr2))       # 1
print("Max:", np.max(arr2))       # 9
print("Argmin:", np.argmin(arr2))  # index 2 (1 is at position)
print("Argmax:", np.argmax(arr2))  # index 5 (9 is at position)
```

Row-wise (axis=1)

```
print("Row-wise Min:", np.min(arr2, axis=1))     # [1 4]
print("Row-wise Max:", np.max(arr2, axis=1))     # [8 9]
print("Row-wise Argmin:", np.argmin(arr2, axis=1))  # [2 1]
print("Row-wise Argmax:", np.argmax(arr2, axis=1))  # [1 2]
```

Column-wise (axis=0)

```
print("Column-wise Min:", np.min(arr2, axis=0))  # [3 4 1]
print("Column-wise Max:", np.max(arr2, axis=0))  # [6 8 9]
print("Column-wise Argmin:", np.argmin(arr2, axis=0))  # [0 1 0]
print("Column-wise Argmax:", np.argmax(arr2, axis=0))  # [1 0 1]
```

3) 3D Array Example

```
arr3 = np.array([
        [[1, 5], [3, 9]],
        [[2, 8], [4, 6]] ])
```

All elements

```
print("Min:", np.min(arr3))       # 1
print("Max:", np.max(arr3))       # 9
print("Argmin:", np.argmin(arr3))  # index of 1 → 0
print("Argmax:", np.argmax(arr3))  # index of 9 → 3
```

Axis Examples

Axis = 0 (compare blocks)

```
print("Axis 0 Min:", np.min(arr3, axis=0)) # [[1 5]  #  [3 6]]
print("Axis 0 Max:", np.max(arr3, axis=0)) # [[2 8] #  [4 9]]
```

Axis = 1 (compare rows inside each block)

```
print("Axis 1 Min:", np.min(arr3, axis=1)) # [[1 5] #  [2 6]]
print("Axis 1 Max:", np.max(arr3, axis=1)) # [[3 9] #  [4 8]]
```

Axis = 2 (compare columns inside each row)

```
print("Axis 2 Min:", np.min(arr3, axis=2)) # [[1 3] #  [2 4]]
print("Axis 2 Max:", np.max(arr3, axis=2)) # [[5 9] #  [8 6]]
```

Real-Life Uses of Min, Max, Argmin, Argmax

1) np.min() – Minimum Value

- Finding the lowest **temperature** in a week
- Finding the **minimum marks** in a class
- Detecting lowest **sensor reading** in IoT

2) np.max()

Real-life use:

- Highest **sales** of a shop
- Maximum **height/weight** from a dataset
- Highest **stock price** in a month

3) np.argmin() Shows **where the smallest value is located**.

Real-life use:

- Finding which **day had the lowest temperature**
- Finding **student index** who got the lowest marks
- Locating minimum **error** in machine learning predictions

4) np.argmax() Shows **where the largest value is located**.

Real-life use:

- Finding which **product sold the most**
- Identifying **peak hour** in traffic data
- Locating maximum **profit point** in a business dataset

Topic-5

Reshaping, Joining, and Splitting Arrays with theory, syntax, and examples.

1) Reshaping Arrays:- Change the shape of an array without changing its data.

Functions & Syntax: reshape(): Reshape to new shape

- new_array = arr.reshape(rows, cols)

Example-1:

```
arr = np.array([1, 2, 3, 4, 5, 6])
arr2d = arr.reshape(2,3)
print(arr2d)      # [[1 2 3] [4 5 6]]
```

Example 2 – 2D Array

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50, 60])
arr2d = arr.reshape(3, 2)  # 3 rows, 2 columns
print(arr2d) # [[10 20] [30 40] [50 60]]
```

Example 3 – 3D Array

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
arr3d = arr.reshape(2, 2, 2)        # 2 blocks, 2 rows, 2 columns
print(arr3d)                         # [[[1 2] [3 4]] [[5 6] [7 8]]]
```

Example 4 – 2D Array

```
arr = np.array([7, 14, 21, 28, 35, 42])
arr2d = arr.reshape(2, 3)           # 2 rows, 3 columns
print(arr2d)                        # [[ 7 14 21] [28 35 42]]
```

Example 5 – 3D Array

```
arr = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
arr3d = arr.reshape(2, 2, 3)  # 2 blocks, 2 rows, 3 columns
print(arr3d) # [[[ 1  2  3] [ 4  5  6]][[ 7  8  9] [10 11 12]]]
```

How to convert 2D or 3D arrays to 1D arrays

You can **convert 2D or 3D arrays to 1D arrays** using **ravel()** or **flatten()**. Both flatten the array, but:

- **ravel()** → returns a **view** (faster, changes affect original array)
- **flatten()** → returns a **copy**

Example – Convert 2D Array to 1D

```
import numpy as np
arr = np.array([7, 14, 21, 28, 35, 42])
arr2d = arr.reshape(2, 3)   # 2 rows, 3 columns
print("2D Array:\n", arr2d)
# Using ravel()
arr1d_ravel = arr2d.ravel()
print("1D Array (ravel):", arr1d_ravel)
# Using flatten()
arr1d_flat = arr2d.flatten()
print("1D Array (flatten):", arr1d_flat)
```

Output:

```
2D Array:
 [[ 7 14 21]
  [28 35 42]]
1D Array (ravel): [ 7 14 21 28 35 42]
1D Array (flatten): [ 7 14 21 28 35 42]
```

Example – Convert 3D Array to 1D

```
arr = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
arr3d = arr.reshape(2, 2, 3)  # 2 blocks, 2 rows, 3 columns
print("3D Array:\n", arr3d)
# Using ravel()
arr1d_ravel = arr3d.ravel()
print("1D Array (ravel):", arr1d_ravel)
# Using flatten()
arr1d_flat = arr3d.flatten()
print("1D Array (flatten):", arr1d_flat)
```

Output:

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

How to convert 1D → 2D → 3D → back to 1D

Example.

```
import numpy as np
arr1d = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  # Step 1: Create 1D array
print("Original 1D Array:\n", arr1d)                # Shape: (12,)
arr2d = arr1d.reshape(3, 4)                         # Step 2: Convert 1D → 2D (3 rows, 4 columns)
print("\nConverted to 2D Array (3x4):\n", arr2d) # Shape: (3, 4)
arr3d = arr1d.reshape(2, 2, 3)              # Step 3: Convert 2D → 3D (2 blocks, 2 rows, 3 columns)
print("\nConverted to 3D Array (2x2x3):\n", arr3d)       # Shape: (2, 2, 3)
arr2d_to_1d = arr2d.ravel()                             # Step 4: Convert 2D → 1D
print("\n2D → 1D using ravel():\n", arr2d_to_1d)
arr3d_to_1d = arr3d.flatten()                           # Step 5: Convert 3D → 1D
print("\n3D → 1D using flatten():\n", arr3d_to_1d)
```

Output:

```
Original 1D Array: [ 1  2  3  4  5  6  7  8  9 10 11 12]
Converted to 2D Array (3x4):      [[ 1  2  3  4][ 5  6  7  8] [ 9 10 11 12]]
Converted to 3D Array (2x2x3):   [[[ 1  2  3][ 4  5  6]][[ 7  8  9][10 11 12]]]
2D → 1D using ravel(): [ 1  2  3  4  5  6  7  8  9 10 11 12]
3D → 1D using flatten(): [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

2) Transpose and Axis Swapping:

Swap rows and columns in an array.

Syntax: arr.T # Transpose
np.transpose(arr) # Transpose function

Example: arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d.T) # [[1 4] [2 5] [3 6]]

Example 1 – 2D Array

```
import numpy as np
arr2d = np.array([[7, 8, 9],  [10, 11, 12]])
# Using .T
print("Transpose using .T:\n", arr2d.T)    # [[ 7 10][ 8 11][ 9 12]]
#Using np.transpose()
print("Transpose using np.transpose():\n", np.transpose(arr2d)) # [[ 7 10][ 8 11][ 9 12]]
```

Example 2 – 3D Array

```
arr3d = np.array([[[1, 2], [3, 4]], [[5, 6],[7, 8]]])
# Using np.transpose to swap axes (0, 2, 1)
arr3d_t = np.transpose(arr3d, (0, 2, 1))
print("3D Array Transposed (axes 0,2,1):\n", arr3d_t) # [[[1 3][2 4]][[5 7] [6 8]]]
```

Array Type	Code Example	Output	Notes
1D Array	arr = np.array([1,2,3])print(arr.T)print(np.transpose(arr))	[1 2 3][1 2 3]	.T has **no effect** on 1D arrays
2D Array	arr2d = np.array([[1,2,3],[4,5,6]])print(arr2d.T)print(np.transpose(arr2d))	[[1 4] [2 5] [3 6]]	.T and np.transpose() give **same result**
3D Array	arr3d = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])arr3d_t = np.transpose(arr3d, (0,2,1))print(arr3d_t)	[[[1 3] [2 4]] [[5 7] [6 8]]]	.T **cannot be used** for 3D; must use np.transpose() with axes

3.Stacking / Joining Arrays

- **"Stacking" means joining two or more arrays together** along a new axis or an existing axis.
- Combine arrays along **horizontal** or **vertical** axis.
- you can **add (stack) more than two arrays** in **np.hstack() , np.vstack(),np.concatenate()**.

Vertical Stacking	One array on top of another	np.vstack()
Horizontal Stacking	Arrays side by side	np.hstack()
Depth Stacking	Stack arrays along a 3rd axis	np.dstack()
General Stacking	Stack along any axis	np.stack()

Functions & Syntax:

- np.hstack((arr1, arr2)) → Horizontal stacking (side by side)
- np.vstack((arr1, arr2)) → Vertical stacking (top to bottom)
- np.concatenate((arr1, arr2), axis=0/1) → General stacking

1. Vertical Stack

```
import numpy as np
a = np.array([1, 2])
b = np.array([3, 4])
print(np.vstack((a, b)))
```

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

2. Horizontal Stack

```
print(np.hstack((a, b)))
```

Output: [1 2 3 4]

3. Depth Stack

```
print(np.dstack((a, b)))
```

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

4. Stack (choose axis)

```
print(np.stack((a, b), axis=1))
```

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

Example:

```
import numpy as np
a = np.array([1, 2])
b = np.array([3, 4])
c = np.array([5, 6])
print(np.hstack((a, b, c)))
```

Output: [1 2 3 4 5 6]

Example:- Depth stacking two 1D arrays

```
import numpy as np
a = np.array([5, 10, 15])
b = np.array([20, 25, 30])
print(np.dstack((a, b)))
```

Output: [[[5 20]
 [10 25]
 [15 30]]]

Example 1: axis = 0 (Vertical Join)

Stacks arrays top-to-bottom

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6],[7, 8]])
result = np.concatenate((arr1, arr2), axis=0)
print(result)
```

Output

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

Example 2: axis = 1 (Horizontal Join)

Stacks arrays side-by-side

```
result = np.concatenate((arr1, arr2), axis=1)
print(result)
```

Output

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

Simple Meaning

- **axis=0 → add rows**
- **axis=1 → add columns**

Example 2: Depth stacking two 2D arrays

```
import numpy as np
a = np.array([[1, 2],
              [3, 4]])
b = np.array([[10, 20],
              [30, 40]])
print(np.dstack((a, b)))
```

Output: [[[1 10]
 [2 20]]
 [[3 30]
 [4 40]]]

Example:

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(np.hstack((a,b))) # [1 2 3 4 5 6]
print(np.vstack((a,b)))   #[[1 2 3] [4 5 6]]
```

Example 1 – 1D Arrays

```
import numpy as np
a = np.array([7, 8, 9])
b = np.array([10, 11, 12])
print("Horizontal Stack:", np.hstack((a, b)))  # Horizontal stacking
print("Vertical Stack:\n", np.vstack((a, b)))  # Vertical stacking
# [ 7  8  9 10 11 12]
#[[ 7  8  9][10 11 12]]
```

Example 2 – 2D Arrays

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print("Horizontal Stack:\n", np.hstack((a, b))) # Horizontal stacking (side by side)
# [[1 2 5 6] [3 4 7 8]]
print("Vertical Stack:\n", np.vstack((a, b)))  # Vertical stacking (top to bottom)
# [[1 2] [3 4] [5 6] [7 8]]
```

3D Array Example

```
import numpy as np
a = np.array([[[1, 2],[3, 4]]])
b = np.array([[[5, 6], [7, 8]]])
```

1) axis = 0 → Stack depth-wise (add new blocks)

(Think: putting b **below** a as a new block)

```
result0 = np.concatenate((a, b), axis=0)
print("Stack axis=0:\n", result0)
```

Output

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

2) axis = 1 → Stack row-wise inside each block (vertical stacking inside 3D)

(Think: adding rows inside the same block)

```
result1 = np.concatenate((a, b), axis=1)
print("Stack axis=1:\n", result1)
```

Output: [[[1 2] [3 4] [5 6] [7 8]]]

3) axis = 2 → Stack column-wise inside each block (horizontal stacking inside 3D)

(Think: adding columns inside the same block)

```
result2 = np.concatenate((a, b), axis=2)
print("Stack axis=2:\n", result2)
```

Output

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

Simple Meaning in 3D

- **axis=0 → add new blocks**
- **axis=1 → add new rows**
- **axis=2 → add new columns**

4.Splitting Arrays

Functions & Syntax:

- np.hsplit(arr, n) → Split horizontally
- np.vsplit(arr, n) → Split vertically
- np.split(arr, n) → General split

Why does np.hsplit() return arrays inside a list? — Proper Explanation

np.hsplit(arr, 2) means:

"Split the array into 2 equal parts horizontally."

- When you split an array, you don't get **one** array — you get **multiple smaller arrays**. Since the result contains **more than one array**, NumPy must return them in a structure that can hold multiple values.The most suitable structure is a **Python list**.
- So, the output looks like this:
- [array(part1), array(part2)]

Example

Input: arr = [[1 2 3 4],
[5 6 7 8]]

After splitting into 2 parts:

Part 1: [[1 2]
[5 6]]

Part 2: [[3 4]
[7 8]]

Since there are **two arrays**, NumPy returns:
[array([[1, 2], [5, 6]]),
array([[3, 4], [7, 8]])]

1. np.hsplit(arr, n) → Horizontal Split (column-wise) (Used for 2D arrays)

Example-1:

```
import numpy as np
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8]])
print(np.hsplit(arr, 2))
```

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

Example 2

```
arr = np.array([[10, 20, 30, 40],
                [50, 60, 70, 80]])
print(np.hsplit(arr, 4))
```

Output [array([[10], [50]]),
array([[20], [60]]),
array([[30],[70]]),
array([[40], [80]])]

Method 1: Print each array separately

```
import numpy as np
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8]])
parts = np.hsplit(arr, 2)
for p in parts:
    print(p)
```

Output (NO array(...) wrapper):
[[1 2]
[5 6]]
[[3 4]
[7 8]]

Method 2: Access and print without list formatting

```
print(parts[0])
print(parts[1])
```

Output: [[1 2] [5 6]]
[[3 4][7 8]]

2. np.vsplit(arr, n) → Vertical Split (row-wise)

(Used for 2D arrays)

Example 1

```
import numpy as np
arr = np.array([[1, 2],
                [3, 4],
                [5, 6],
                [7, 8]])
print(np.vsplit(arr, 2))
```

Output

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

Example 2

```
arr = np.array([[10, 20],
                [30, 40],
                [50, 60]])
print(np.vsplit(arr, 3))
```

Output

```
[array([[10, 20]]),
 array([[30, 40]]),
 array([[50, 60]])]
```

3. np.split(arr, n) → General Split :- Works on **1D, 2D, 3D** arrays depending on axis.

Example 1 — Split 1D array

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
print(np.split(arr, 3))
```

Output

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

Example 2 — Split 2D array (by default axis=0 → row-wise)

```
arr = np.array([[10, 20],
                [30, 40],
                [50, 60],
                [70, 80]])
print(np.split(arr, 2))
```

Output

```
[array([[10, 20],
        [30, 40]]),
 array([[50, 60],
        [70, 80]])]
```

	Function	What it splits	Real-Life Use
1.	hsplit()	Columns	ML feature separation, image left-right split
2.	vsplit()	Rows	Train/test split, time-series chunking
3.	split()	General (1D,2D,3D)	Signal processing, parallel computation

Topic 6: Random Numbers and Probability

1. Random Number Functions

- NumPy provides many functions to create random numbers.These numbers are very useful in **testing, ML, simulation, games, statistics**.

Function	Use
1. **rand()**	: Random decimal 0–1
2. **randn()**	: Random normal numbers (−∞ to +∞)
3. **randint()**	: Random integers
4. **seed()**	: Repeat random results
5. **choice()**	: Pick random element
6. **shuffle()**	: Shuffle array

1. np.random.rand()

- Generates **random decimal numbers**
- Range is always **0 to 1**
- Uniform distribution: **Equal probability for every number in the range.**
- You can specify any shape

A). Basic: Random number between 0 and 1

```
import numpy as np
print(np.random.rand())
```

Output 0.374540

B). Random 2×3 matrix

```
print(np.random.rand(2, 3))
```

Output: [[0.156 0.867 0.398]
[0.672 0.234 0.912]]

Using rand() for Custom Ranges (0–1 → any range)

rand() always gives **0–1** but we can **convert to any range** using:

```
new_value = a + (b - a) * rand()
```

C). Random number between 5 and 10

```
x = 5 + (10 - 5) * np.random.rand()
print(x)
```

Output: 7.834251

D). Random array between 100 and 200

```
arr = 100 + (200 - 100) * np.random.rand(2, 3)
print(arr)
```

Output: [[126.45 173.89 159.22]
[143.77 198.66 102.55]]

E). Random values between -1 and 1

```
x = -1 + (1 - (-1)) * np.random.rand(5)
print(x)
```

Output: [-0.55 0.12 -0.88 0.44 0.96]

F). Random number between 0 and 10 (shortcut)

```
print(10 * np.random.rand())
```

Output: 6.32987

G). Random number between -5 and 5 (shortcut)

```
print(-5 + 10 * np.random.rand())
```

Output: 1.89234

Uniform distribution means:

Every value in the range has an equal chance of occurring. No value is more likely or less likely.

Example:

If you generate numbers between **0 and 1** with uniform distribution:

- 0.1 → equal chance
- 0.5 → equal chance All values are **equally likely**.

Real-life example

A **fair dice** is a uniform distribution:
Numbers 1, 2, 3, 4, 5, 6
→ all have equal chance **1/6**

In NumPy np.random.rand() generates numbers with **uniform distribution**, meaning:
Any decimal between 0 and 1 is equally possible.

8. Random 4×4 matrix between 50 and 100

```
print(50 + 50 * np.random.rand(4, 4))
```

Output (example):

[[73.11 92.45 69.88 58.55]
[84.22 65.10 77.66 96.88]
[51.55 99.34 68.45 82.19]
[58.90 71.22 95.44 67.33]]

Note: Works for **scalars, 1D arrays, 2D matrices, 3D arrays**

1. np.random.randn() — Random Normal Distribution

(Mean = 0, SD = 1, values can be negative or positive)

a). Basic: One random normal number

```
import numpy as np
print(np.random.randn())
```

Output: -0.8246

b). Random 2×3 matrix (normal distribution)

```
print(np.random.randn(2, 3))
```

Output: [[0.812 -1.442 0.556]
[-0.234 1.112 -0.987]]

c). Normal distribution with custom mean & std

Formula: new = mean + std * randn()

Example: mean=50, std=10

```
x = 50 + 10 * np.random.randn()
print(x)
```

Output: 43.8821

SD (Standard Deviation)

- Measures how **spread out numbers** are from the mean.
- **Mean = 0, SD = 1** → most values lie between -1 and 1.
- Numbers can be **negative or positive**.

One-line: SD = how far numbers are from the mean.

2. np.random.randint() —Generate Random Integers

Syntax: np.random.randint(low, high=None, size=None, dtype=int)

1. low Minimum integer (inclusive)
2. high Maximum integer (exclusive). If None, random integers are from 0 to low-1
3. size Shape of the output (scalar, tuple, or array)
4. dtype Data type of output integers (default = int)

Examples:

A. Single random integer between 0 and 9

```
np.random.randint(10)
```

Output:7

B. **Random integer between 5 and 15**

```
np.random.randint(5, 16)
```

Output:12

C. **Random 2×3 matrix of integers between 1 and 100**

```
np.random.randint(1, 101, (2, 3))
```

Output: [[12 78 45]
[34 90 66]]

3. np.random.seed() — Repeat the Same Random Results

→ **np.random.seed()** fixes the **starting point** for NumPy's random number generator.

- This ensures that every time you run the code, you get the **same random numbers**.
- Useful for **testing, debugging, or reproducible results**.

A). Seed for repeatable output (single number)

```
import numpy as np
np.random.seed(10)
print(np.random.rand())
```

Output: 0.77132064

- Every time you run this with seed(10), the output is **always the same**.

B). Seed with multiple random numbers

```
np.random.seed(5)
print(np.random.randint(1, 20, 5))
```

Output: [4 1 7 1 4]

- Using the same seed generates the **same sequence of random integers**.
- Change the seed → different sequence.

4. np.random.choice() — Pick Random Elements

np.random.choice() selects **random elements from a list or array**.

- You can pick **one or many elements**.
- Can allow **repetition** (default) or **no repetition** (replace=False).

A). Pick one random value

```
import numpy as np
items = [10, 20, 30, 40]
print(np.random.choice(items))
```

Output:30 → Randomly selects **one item** from the list.

B). Pick multiple values (with repetition)

```
print(np.random.choice(items, 5))
```

Output: [20 40 10 40 30]

- Picks **5 items** randomly.
- **Repetition allowed**, same item can appear more than once.

C). Pick multiple values (no repetition)

```
print(np.random.choice(items, 3, replace=False))
```

Output: [30 10 40]

- Picks **3 items** randomly.
- **No repetition** (replace=False).

Key Points

- items → list or array to choose from
- size → number of elements to pick
- replace=False → ensures unique values

Note: np.random.choice() = randomly pick element(s) from a list, with or without repetition.

5. np.random.shuffle() — Shuffle Array (in-place)

→ np.random.shuffle() **randomly rearranges elements of an array**.

- Works **in-place** → modifies the original array.
- Can shuffle **1D arrays** or **rows of 2D arrays**.

A). Shuffle a 1D array

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
np.random.shuffle(arr)
print(arr)
```

Output: [3 5 1 4 2]

- Randomly reorders the elements of the array.
- Original array arr is changed.

B). Shuffle rows of a 2D matrix

```
mat = np.array([[1, 2], [3, 4], [5, 6]])
np.random.shuffle(mat)
print(mat)
```

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

- Randomly rearranges **rows**, but columns **stay in order**.

Key Points

- **In-place operation** → array is changed, nothing is returned.
- Useful for **randomizing datasets** before training machine learning models.

Note: np.random.shuffle() = randomly rearrange elements of an array (in-place).

- Combined project: "Generate fake student dataset"
- How to generate random matrices for ML
- Random probability examples

Topic 7: Input and Output Operations in NumPy

- NumPy provides easy ways to **save, load, and convert data**. This is useful when you want to **store arrays for later use** or **share datasets**.

1. Saving and Loading Arrays

1.1 Using .npy format (binary)

- .npy is **NumPy's native binary format**.
- Saves arrays **efficiently** with all metadata.

Syntax:

```
np.save(filename, array)        # Save array to .npy file
np.load(filename)               # Load array from .npy file
```

Example:

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
# Save array
np.save("my_array.npy", arr)
# Load array
loaded_arr = np.load("my_array.npy")
print(loaded_arr)
```

Output: [10 20 30 40 50]

Note: Binary format is fast and retains **array shape & type**.

1.2 Using .txt format (text)

- Saves arrays as **plain text**.
- Can specify **delimiter** for columns.

Syntax

```
np.savetxt(filename, array, delimiter=',')  # Save array to .txt or .csv
np.loadtxt(filename, delimiter=',')         # Load array from text file
```

Example

```
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
# Save as text file
np.savetxt("my_array.txt", arr2, delimiter=',')
# Load text file
loaded_arr2 = np.loadtxt("my_array.txt", delimiter=',')
print(loaded_arr2)
```

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

Note: Useful for **CSV files** or sharing with other programs like Excel.

2. Working with .csv files

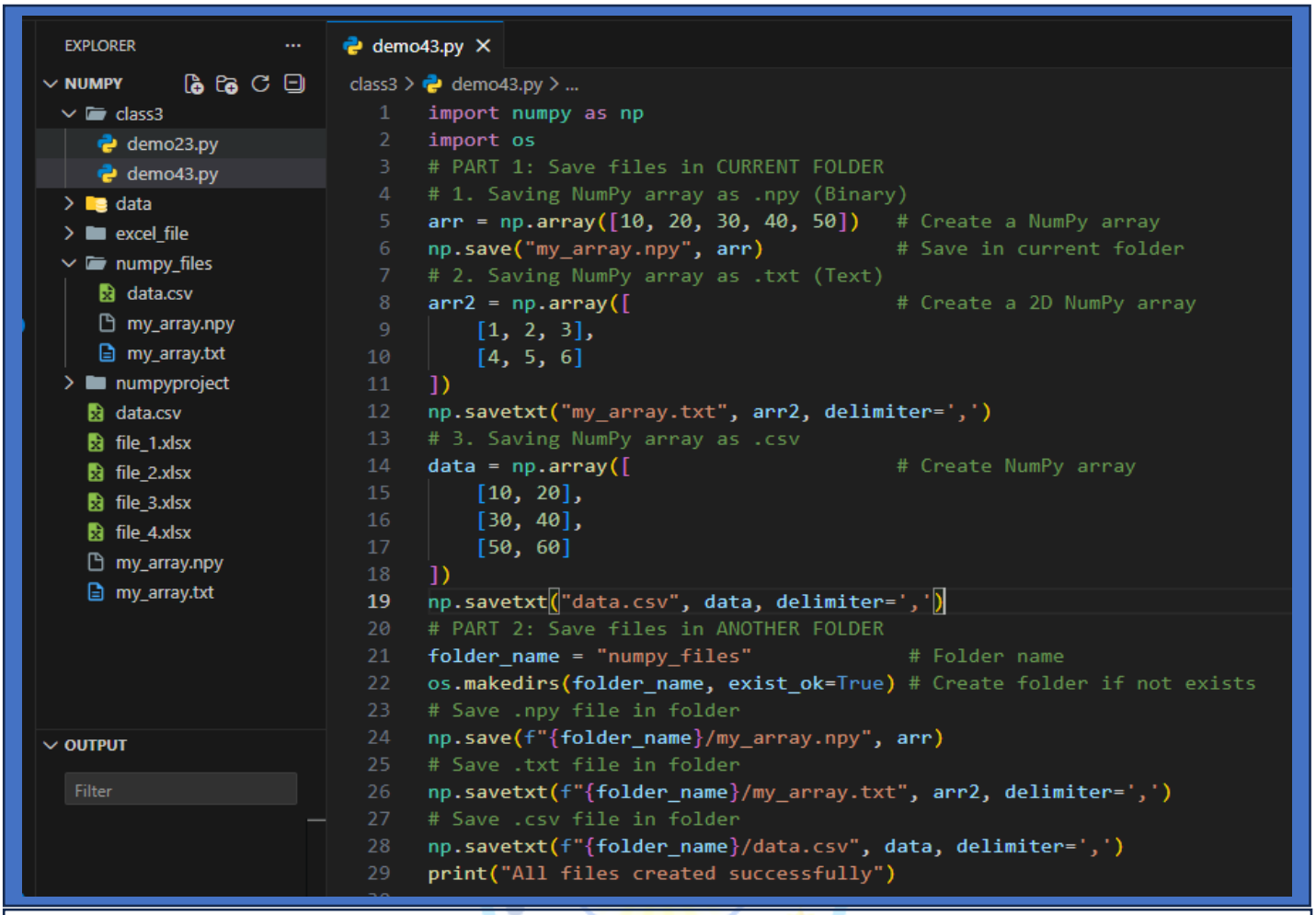
- .csv = Comma-Separated Values
- Can **save and load numeric data** using savetxt and loadtxt

Example:

```
data = np.array([[10, 20], [30, 40], [50, 60]])
# Save as CSV
np.savetxt("data.csv", data, delimiter=',')
# Load CSV
loaded_data = np.loadtxt("data.csv", delimiter=',')
print(loaded_data)
```

Output: [[10. 20.][30. 40.][50. 60.]]

Example: -1 How to create the different kind of file in NumPy



Example-1:

```
import numpy as np
import os
# PART 1: Save files in CURRENT FOLDER
# 1. Saving NumPy array as .npy (Binary)
arr = np.array([10, 20, 30, 40, 50])  # Create a NumPy array
np.save("my_array.npy", arr)          # Save in current folder
# 2. Saving NumPy array as .txt (Text)
arr2 = np.array([ [1, 2, 3],  [4, 5, 6] ])    ## Create a 2D NumPy array
np.savetxt("my_array.txt", arr2, delimiter=',')
# 3. Saving NumPy array as .csv
data = np.array([ [10, 20], [30, 40], [50, 60] ])
np.savetxt("data.csv", data, delimiter=',')
# PART 2: Save files in ANOTHER FOLDER
folder_name = "numpy_files"           # Folder name
os.makedirs(folder_name, exist_ok=True) # Create folder if not exists
# Save .npy file in folder
np.save(f"{folder_name}/my_array.npy", arr)
# Save .txt file in folder
np.savetxt(f"{folder_name}/my_array.txt", arr2, delimiter=',')
# Save .csv file in folder
np.savetxt(f"{folder_name}/data.csv", data, delimiter=',')
print("All files created successfully")
```

How to Save Data in Excel Format Using NumPy

- NumPy itself cannot directly save an Excel (.xlsx) file.
- But you can easily do it by combining NumPy with pandas, because pandas supports Excel.

Why use pandas for Excel? Because:

- NumPy saves only **.npy**, **.txt**, **.csv**
- Excel (.xlsx) needs a special writer → pandas handles it

Step 1: Install Required Libraries

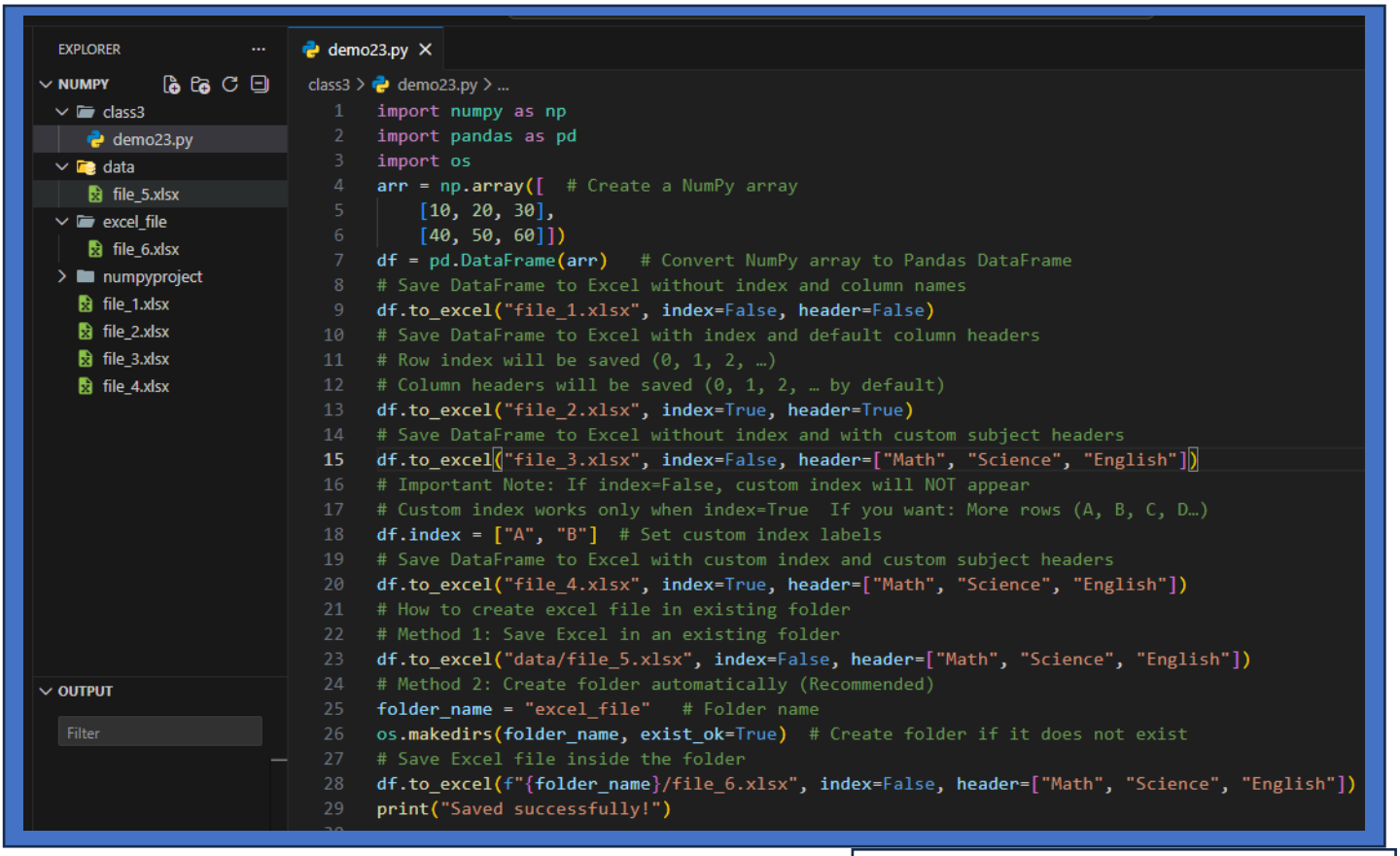
❖ Install NumPy and Pandas:

➢ pip install numpy pandas

❖ Install **openpyxl** (required for writing Excel files):

➢ pip install openpyxl

Step-2: Example-1: How to create Excell file and Save array data in Excell.



➢ df = pd.DataFrame(arr)

❖ **What it does?**

- Converts your **NumPy array** into a **pandas DataFrame**.
- A DataFrame is like an Excel table (rows + columns).

Example If arr = [[10,20,30],[40,50,60]]

Then the DataFrame looks like:

0	1	2
10	20	30
40	50	60

df.to_excel("my_array.xlsx", index=False, header=False)

- **Function used:** to_excel()
- **What it does:**
  Saves DataFrame into an **Excel (.xlsx) file**.
- **Why we use it:**
  To store data in Excel format for sharing or reporting.

Parameters explained:

- "my_array.xlsx" → File name
- index=False → Removes row numbers
- header=False → Removes column names

➢ **df.to_excel("my_array.xlsx", index=False, header=False)**

❖ **What it does?:-** Creates an Excel file named **my_array.xlsx**

- Saves your DataFrame into that file

➢ **Meaning of the parameters:**

- **index=False** → Don't write row numbers (0,1,2…)
- **header=False** → Don't write column names (0,1,2…)

❖ What can we write instead of False?

- Both **index** and **header** can be:
- **True** Allows row numbers or column names to be saved.
- **Example:** df.to_excel("my_array.xlsx", index=True, header=True)

Output Excel:

	A	B	C
0	10	20	30
1	40	50	60

❖ **Custom Column Names (Only for header)**

- Instead of False, you can give a list:
- df.to_excel("my_array.xlsx", index=False, header=["Math", "Science", "English"])

Output:

Math	Science	English
10	20	30
40	50	60

➢ **Custom Index Labels**

- Instead of index=False, you can give labels:

```
df.to_excel("my_array.xlsx", index=True)
df.index = ["Row1", "Row2"]
df.to_excel("my_array.xlsx")
```

Excel Output:

	0	1	2
Row1	10	20	30
Row2	40	50	60

Parameter	What You Can Use	Meaning
index	False / True	Hide or show row numbers
	Custom labels	Replace row labels
header	False / True	Hide or show column names
	List of names	Replace column names

Example-1: How to create Excell file and Save array data in Excel.

```
import numpy as np
import pandas as pd
import os
arr = np.array([
   [10, 20, 30],
   [40, 50, 60]
])
df = pd.DataFrame(arr)
df.to_excel("file_1.xlsx", index=False, header=False)
df.to_excel("file_2.xlsx", index=True, header=True)
df.to_excel("file_3.xlsx", index=False, header=["Math", "Science", "English"])
df.index = ["A", "B"]
df.to_excel("file_4.xlsx", index=True, header=["Math", "Science", "English"])
df.to_excel("data/file_5.xlsx", index=False, header=["Math", "Science", "English"])
folder_name = "excel_file"
os.makedirs(folder_name, exist_ok=True)
df.to_excel(f"{folder_name}/file_6.xlsx", index=False, header=["Math", "Science", "English"])
print("Saved successfully!")
```

Example: How to save the 3-D NumPy array data in Excel

Excel works with **2-D tables**, so a **3-D array must be reshaped or converted slice-by-slice** before saving.

```
import numpy as np
import pandas as pd
import os
arr_3d = np.array([   # Create a 3-D NumPy array (2 blocks, 2 rows, 3 columns)
   [
      [10, 20, 30],
      [40, 50, 60]
   ],
   [
      [70, 80, 90],
      [100, 110, 120]
   ]
])
df = pd.DataFrame(arr_3d[0])                                  # Convert first 2-D slice of 3-D array to DataFrame
df.to_excel("file_1.xlsx", index=False, header=False)         # Save DataFrame to Excel without index and header
df.to_excel("file_2.xlsx", index=True, header=True)           # Save DataFrame to Excel with index and default headers
df.to_excel("file_3.xlsx", index=False, header=["Math", "Science", "English"])  # with custom headers
df.index = ["A", "B"]                                         # Set custom index
# Save DataFrame with custom index and headers
df.to_excel("file_4.xlsx", index=True, header=["Math", "Science", "English"])
os.makedirs("data", exist_ok=True)                            # Save Excel file in existing folder
df.to_excel("data/file_5.xlsx", index=False, header=["Math", "Science", "English"])
folder_name = "excel_file"  # Save Excel file in automatically created folder
os.makedirs(folder_name, exist_ok=True)
df.to_excel(f"{folder_name}/file_6.xlsx", index=False, header=["Math", "Science", "English"])
print("3-D array example saved successfully!")
```

How to save the 3-D array in CSV file.

- **3-D array *can* be saved to a CSV file, but not directly.**
- CSV supports **only 2-D data**, so a **3-D array must be reshaped or flattened** before saving.

Method 1: Reshape 3-D Array to 2-D (Most Common)

Example

```
import numpy as np
arr_3d = np.array([   # Create a 3-D NumPy array (2 × 2 × 3)
   [   [10, 20, 30],
      [40, 50, 60] ],
   [ [70, 80, 90],
      [100, 110, 120] ]])
arr_2d = arr_3d.reshape(arr_3d.shape[0], -1)   # Reshape 3-D array to 2-D
# Save reshaped data to CSV file
np.savetxt("3d_to_csv.csv", arr_2d, delimiter=',')
print("3-D array saved to CSV using reshape!")
```

Output CSV Structure

```
10,20,30,40,50,60
70,80,90,100,110,120
```

Method 2: Save Each 2-D Slice as a Separate CSV File

Example

```
import numpy as np
import os
# Create a 3-D NumPy array
arr_3d = np.array([
   [ [1, 2, 3],
      [4, 5, 6] ],
   [ [7, 8, 9],
      [10, 11, 12]
   ] ])
os.makedirs("csv_slices", exist_ok=True)  # Create folder for CSV files
for i in range(arr_3d.shape[0]):  # Save each slice as a separate CSV
   np.savetxt(f"csv_slices/slice_{i}.csv", arr_3d[i], delimiter=',')
print("Each 3-D slice saved as a separate CSV file!")
```

Output

```
csv_slices/
 ├── slice_0.csv
 └── slice_1.csv
```

Method 3: Flatten Entire 3-D Array into One Column

Example

```
import numpy as np
arr_3d = np.array([
   [[1, 2], [3, 4]],
   [[5, 6], [7, 8]]
]) # Flatten 3-D array
flat_arr = arr_3d.flatten()
# Save flattened data to CSV
np.savetxt("3d_flat.csv", flat_arr, delimiter=',')
print("Flattened 3-D array saved to CSV!")
```

Output

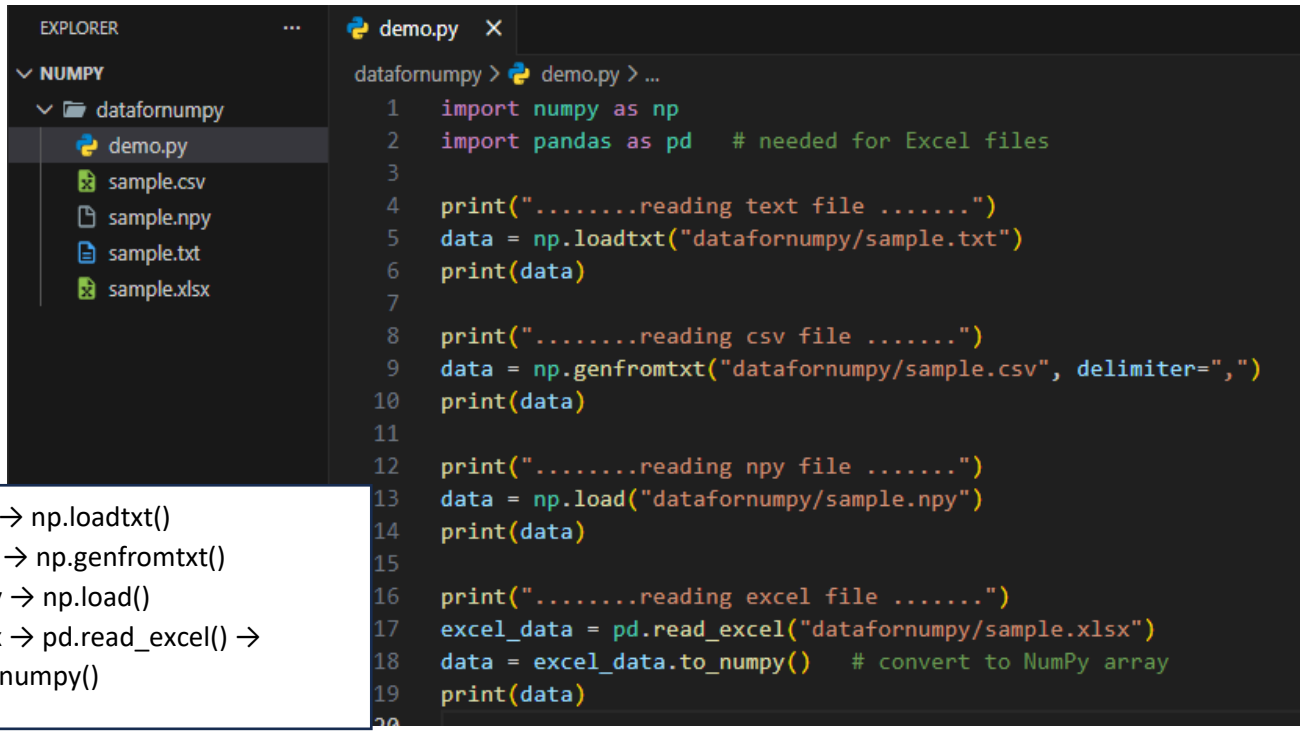
```
1
2
3
4
5
6
7
8
```

How to read files using NumPy

(from **same folder**, **another folder**, **multiple files**, **URL**, and **API data)**

1. **np.loadtxt()** – Used to read numerical data from text or CSV files where values are well-formatted and consistent.
2. **np.load()** – Used to load NumPy binary files (.npy or .npz) efficiently into NumPy arrays.
3. **pd.read_excel()** – Used to read Excel files (.xls or .xlsx) into a DataFrame, which can be converted to a NumPy array.
4. **np.genfromtxt()** – Used to read text or CSV files that may contain missing values or mixed data types.

1.How to Read .txt or .csv or .npy or .xlsx file from the same folder



- .txt → np.loadtxt()
- .csv → np.genfromtxt()
- .npy → np.load()
- .xlsx → pd.read_excel() → .to_numpy()

Example-1:

```
import numpy as np
import pandas as pd   # needed for Excel files
print(".....reading text file .....")
data = np.loadtxt("datafornumpy/sample.txt")
print(data)
print(".....reading csv file .....")
data = np.genfromtxt("datafornumpy/sample.csv", delimiter=",")
print(data)
print(".....reading npy file .....")
data = np.load("datafornumpy/sample.npy")
print(data)
```

❖ What is Working Directory?

- **Working directory** is the folder from which Python runs the program and searches for files.

❖ How to Check Working Directory

```
import os
print(os.getcwd())
```

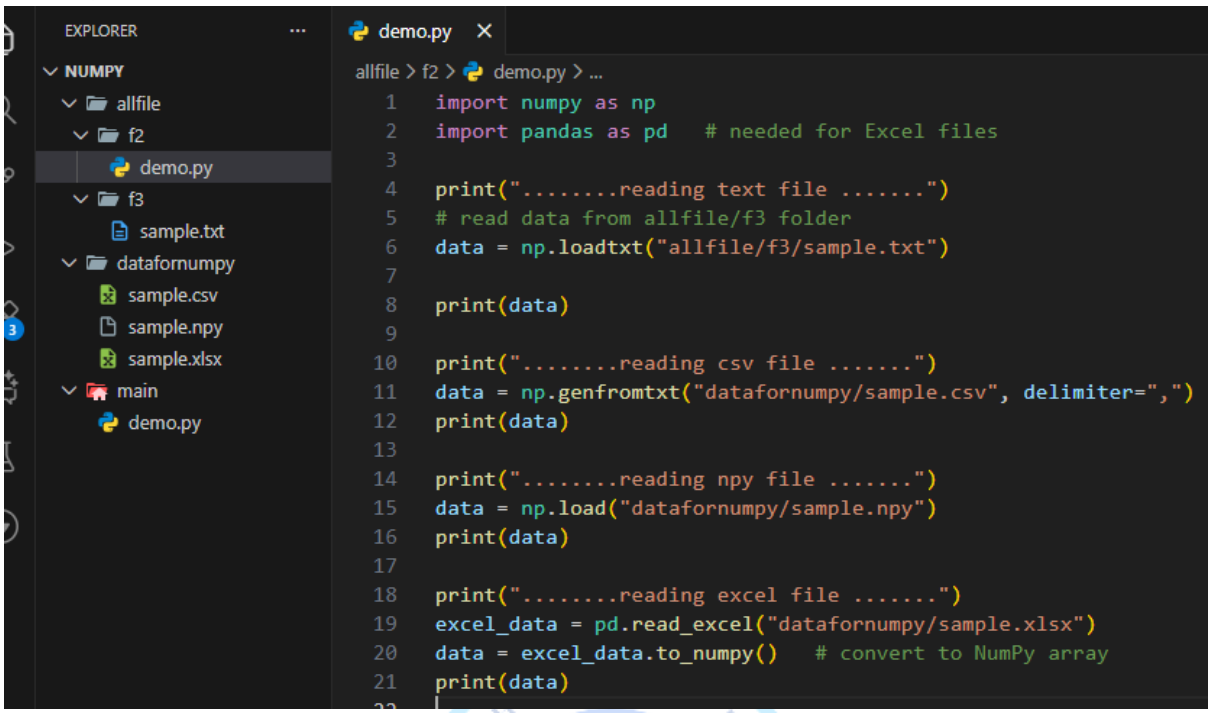
- If the file is not in the working directory, Python gives **FileNotFoundError**.

➔ Use correct **relative path** or change directory using:

➔ os.chdir("folder_name")

2. How to Read .txt or .csv or .npy or .xlsx file from another folder

Example: - Keep data in this different folder and run this code



3.How to read the .csv file from any URLs Api and multiple file

- **Reading from URL**

  np.loadtxt("https://example.com/file.csv", delimiter=",")

- **Reading from API (JSON)**

  requests.get(url).json()

  np.array(data)

- **Reading multiple files**

  glob.glob("path/*.csv")

  np.loadtxt(filename, delimiter=",")

Example:-1

```
import numpy as np
url = "https://raw.githubusercontent.com/uiuc-cse/data-fa14/gh-pages/data/iris.csv"
data = np.genfromtxt(
   url,
   delimiter=",",
   skip_header=1,
   dtype=None,
   encoding="utf-8"
)
print(data)
```

❖ API (Application Programming Interface)

- API lets **one application talk to another**.
- Used to **send and receive data** over the internet.
- Mostly used as **REST APIs** in real projects.
- Works with **HTTP methods**: GET, POST, PUT, DELETE.
- APIs usually **return data in JSON format**.
- Used in **web apps, mobile apps, dashboards, data science**.
- In Python, APIs are accessed using the **requests library**.
- Helps in getting **real-time data** (weather, stock, users, etc.).

❖ JSON (JavaScript Object Notation) – Key Points

- JSON is a **lightweight data format**.
- Easy to **read, write, and understand**.
- Data stored as **key–value pairs**.
- Structure is similar to **Python dict and list**.
- JSON is the **most common API response format**.
- Supports **nested data**.
- Works with **all programming languages**.
- Best for **data exchange over the network**.

❖ API + JSON (Training View)

- **API provides data, JSON formats data**.
- Python → API → JSON → NumPy/Pandas → analysis.
- Used in **real-world applications**,

Reading from URL (CSV) using NumPy

```
import numpy as np
url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv"
# Load CSV data from URL
data = np.loadtxt(url, delimiter=",", skiprows=1)
print(data)
```

- Works directly
- Used in **real data science projects**
- skiprows=1 skips header row

Reading from API (JSON) → NumPy array

Real Public API (JSONPlaceholder)

>> Step 2: Install requests

>> pip install requests

```
import requests
import numpy as np
url = "https://jsonplaceholder.typicode.com/posts"
response = requests.get(url)
json_data = response.json()   # JSON list of dictionaries
# Convert JSON to NumPy array
data = np.array(json_data)
print(data)
print("Shape:", data.shape)
```

❖ Reading Multiple CSV files from a folder

Folder structure

```
data/
  ├── file1.csv
  ├── file2.csv
  └── file3.csv
```

Code

```
import numpy as np
import glob
files = glob.glob("data/*.csv")
all_data = []
for filename in files:
   arr = np.loadtxt(filename, delimiter=",", skiprows=1)
   all_data.append(arr)
# Combine all files into one array
final_data = np.vstack(all_data)
print(final_data)
```

Note: vstack() is used to merge multiple NumPy arrays row-wise into a single array, and * is used to unpack list elements when passing multiple arrays as arguments.

Weather App

weatherapp.py > ...

```python
import requests
import pandas as pd
import numpy as np
# 1 Live Weather API (No API Key)
url = (
    "https://api.open-meteo.com/v1/forecast"
    "?latitude=28.61&longitude=77.20"
    "&hourly=temperature_2m"
)
response = requests.get(url)
# 2 API → JSON
json_data = response.json()
# Extract useful part
time = json_data["hourly"]["time"]
temperature = json_data["hourly"]["temperature_2m"]
# 3 JSON → Pandas
df = pd.DataFrame({
    "Time": time,
    "Temperature": temperature
})
print("Pandas DataFrame:")
print(df.head())
# 4 Pandas → NumPy
numpy_array = df.to_numpy()
print("\nNumPy Array:")
print(numpy_array)
```


```python
import requests
import pandas as pd
import numpy as np
# 1 Live Weather API (No API Key)
url = (
   "https://api.open-meteo.com/v1/forecast"
   "?latitude=28.61&longitude=77.20"
   "&hourly=temperature_2m"
)
response = requests.get(url)
# 2 API → JSON
json_data = response.json()
# Extract useful part
time = json_data["hourly"]["time"]
temperature = json_data["hourly"]["temperature_2m"]
# 3 JSON → Pandas
df = pd.DataFrame({
   "Time": time,
   "Temperature": temperature
})
print("Pandas DataFrame:")
print(df.head())
# 4 Pandas → NumPy
numpy_array = df.to_numpy()
print("\nNumPy Array:")
print(numpy_array)
```

3. Data Conversion between Lists and Arrays

- Convert **list → array**: np.array(list)
- Convert **array → list**: array.tolist()

1. How to convert list to array

Example:

```
# List to array
lst = [1, 2, 3, 4]
arr = np.array(lst)
print(arr)
# Array to list
arr2 = np.array([10, 20, 30])
lst2 = arr2.tolist()
print(lst2)
```

Output: [1 2 3 4]
[10, 20, 30]

❖ Practical Example: Save & Load NumPy Array

```
import numpy as np
# Create a random array
data = np.random.randint(1, 100, (3, 3))
print("Original Array:")
print(data)
# Save array to file
np.save("random_data.npy", data)
np.savetxt("random_data.csv", data, delimiter=',')
# Load arrays back
loaded_bin = np.load("random_data.npy")
loaded_txt = np.loadtxt("random_data.csv", delimiter=',')
print("Loaded from .npy file:")
print(loaded_bin)
print("Loaded from .csv file:")
print(loaded_txt)
```

Output:

```
Original Array:
[[23 45 67]
 [12 78 34]
 [56 89 90]]
Loaded from .npy file:
[[23 45 67]
 [12 78 34]
 [56 89 90]]
Loaded from .csv file:
[[23. 45. 67.]
 [12. 78. 34.]
 [56. 89. 90.]]
```

Real-life example:

You run a **machine learning experiment** and generate a large dataset of features. Instead of recalculating it every time, you **save it** and **load it quickly** for testing models.

```
import numpy as np
# Simulate features of 100 students
features = np.random.randint(0, 101, (100, 5))  # 5 features per student
# Save features
np.save("student_features.npy", features)
# Load features
loaded_features = np.load("student_features.npy")
print(loaded_features.shape)
```

Output: (100, 5) **Note:-** Real-life use: Save **large datasets** for ML or data analysis.

1.2 Using .txt or .csv

Real-life example:

You have **sales data** from a store and want to share it with Excel or colleagues.

```
sales = np.array([[100, 200, 150], [120, 220, 180], [90, 210, 160]])
# Save as CSV
np.savetxt("sales_data.csv", sales, delimiter=',')
# Load CSV
loaded_sales = np.loadtxt("sales_data.csv", delimiter=',')
print(loaded_sales)
```

Output:

```
[[100. 200. 150.]
 [120. 220. 180.]
 [ 90. 210. 160.]]
```

Note:- Real-life use: Save **numeric data** to share in **Excel, Google Sheets, or other programs**.

2. Convert NumPy Array to Python List → Convert **array → list**: array.tolist()

Example:

```
import numpy as np
# Create a NumPy array
arr = np.array([5, 10, 15, 20])
# Convert array to list
lst = arr.tolist()
print("NumPy Array:")
print(arr)
print("Python List:")
print(lst)
```

Output:

```
NumPy Array:
[ 5 10 15 20 ]
Python List:
[5, 10, 15, 20]
```

Example:

```
import numpy as np
# Create a random NumPy array
data = np.random.randint(1, 100, (3, 3))
print("Original NumPy Array:")
print(data)
# Convert NumPy array to Python list
data_list = data.tolist()
print("Converted Python List:")
print(data_list)
```

Output:

```
Original NumPy Array:
[[12 45 67]
 [23 89 10]
 [56 34 78]]
Converted Python List:
[[12, 45, 67], [23, 89, 10], [56, 34, 78]]
```

Mini Project: Student Data Generator + Save/Load + Analysis

This project generates **random student data**, saves it to a file, loads it again, and performs analysis.

Step 1: Generate Random Student Dataset

```
import numpy as np
np.random.seed(10)                    # repeatable output
n = 20                                # Number of students
roll = np.arange(1, n+1)              # 1. Roll numbers
math = np.random.randint(0, 101, n)   # 2. Random Marks (0–100)
science = np.random.randint(0, 101, n)
english = np.random.randint(0, 101, n)
# Create full dataset (20 rows × 3 subjects)
marks = np.column_stack((math, science, english))
print("Generated Marks:")
print(marks)
```

Step 2: Save Data to a .npy File (Fast Loading)

```
np.save("student_marks.npy", marks)
```

Step 3: Load Data from .npy File

```
loaded_marks = np.load("student_marks.npy")
print("Loaded Marks:")
print(loaded_marks)
```

Step 4: Save Data to CSV (to share with Teacher)

```
np.savetxt("student_marks.csv", marks, delimiter=",", fmt="%d")
```

Note: CSV can be opened in **Excel**, **Google Sheets**, etc.

Step 5: Load Data from CSV

```
csv_marks = np.loadtxt("student_marks.csv", delimiter=",")
print("Marks Loaded from CSV:")
print(csv_marks)
```

Step 6: Convert Array → List and Back

```
marks_list = marks.tolist()
print("Converted to List:", marks_list[:3])  # show first 3
marks_array = np.array(marks_list)
print("Back to NumPy Array:")
print(marks_array[:3])
```

Step 7: Calculate Total, Percentage & Grade

```
total = marks_array.sum(axis=1)       # Total marks per student
percentage = total / 3                # Percentage
grades = []                           # Grade system
for p in percentage:
    if p >= 90: grades.append("A+")
    elif p >= 80: grades.append("A")
    elif p >= 70: grades.append("B")
    elif p >= 60: grades.append("C")
    else: grades.append("F")
print("Percentages:", percentage)
print("Grades:", grades)
```

Output:

```
Generated Marks:
[[ 9  15  89]
 [ 64  28  88]
 [ 39  13  87]
 ...
]

Loaded Marks:
[[ 9  15  89]
 [ 64  28  88]
 ...
]
Percentages:
[37.67 60.00 46.33 ...]
Grades:
        ['F', 'C', 'F', ...]
```

Topic-8: Handling Missing Data in NumPy

1. Representing Missing Data (np.nan)

- np.nan means **Not a Number**.
- It is used to represent **missing or undefined values** in NumPy arrays.

Example:

```
import numpy as np
arr = np.array([10, 20, np.nan, 40])
```

2. Checking Missing Data (np.isnan())

- np.isnan() checks which values are NaN.
- It returns **True for NaN** and **False for numbers**.

Example: np.isnan(arr)

3. Replacing NaN Values (np.nan_to_num())

- np.nan_to_num() replaces NaN with **0** (by default).
- Useful to clean data before calculations.

Example: clean_arr = np.nan_to_num(arr)

4. Functions Ignoring NaN

- np.nanmean() → Calculates mean **ignoring NaN**
- np.nansum() → Calculates sum **ignoring NaN**

Example:

```
np.nanmean(arr)
np.nansum(arr)
```

Practical Example

```
import numpy as np
data = np.array([5, 10, np.nan, 15, np.nan])
# Replace NaN with 0
new_data = np.nan_to_num(data)
# Mean ignoring NaN
mean_value = np.nanmean(data)
print("After replacing NaN:", new_data)
print("Mean ignoring NaN:", mean_value)
```

Note:

- np.nan → missing value
- np.isnan() → find missing values
- np.nan_to_num() → replace missing values
- np.nanmean() → average without missing values

What is RID Organization (RID संस्था क्या

- **RID Organization** यानि **Research, Innovation and Discovery Organization एक** संस्था हैं जो TWF (TWKSAA WELFARE FOUNDATION) **NGO** द्वारा RUN किया जाता है | जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही **इस RID Organization** की स्थपना 30.09.2023 किया गया है | जो TWF द्वारा संचालित किया जाता है |
- TWF (TWKSAA WELFARE FOUNDATION) NGO की स्थपना 26-10-2020 में बिहार की पावन धरती सासाराम में Er. RAJESH PRASAD एवं Er. SUNIL KUMAR द्वारा किया गया था जो की भारत सरकार द्वारा मान्यता प्राप्त संस्था हैं |
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों**,** शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

MISSION, VISSION & MOTIVE OF "RID ORGANIZATION"	
मिशन	हर एक ONE भारत के संग
विजन	TALENT WORLD KA SHRESHTM AB AAYEGA भारत में और भारत का TALENT भारत में
मकसद	NEW (RID, PMS, TLR)

MOTIVE OF RID ORGANIZATION NEW (RID, PMS, TLR)		
NEW (RID)		
R	I	D
Research	Innovation	Discovery
NEW (TLR)		
T	L	R
Technology, Theory, Technique	Law	Rule
NEW (PMS)		
P	M	S
Product, Project, Production	Machine	Service

RID रीड संस्था की मिशन, विजन एवं मकसद को सार्थक हमें बनाना हैं |
भारत के वर्चस्व को हर कोने में फैलना हैं |
कर के नया कार्य एक बदलाव समाज में लाना हैं |
रीड संस्था की कार्य-सिध्दांतों से ही, हमें अपनी पहचान बनाना हैं |

Er. Rajesh Prasad (B.E, M.E)
Founder:
TWF & RID Organization

Advance Python के इस **E-Book** में अगर मिलती त्रुटी मिलती है तो कृपया हमें सूचित करें | **WhatsApp's No: 9202707903 or**
Email Id: ridorg.in@gmail.com

।। धन्यवाद ।।