

CSBG009T

ADVANCE PYTHON



DR. C.V. RAMAN UNIVERSITY

Kargi Road, Kota, Bilaspur, (C. G.),

Ph. 07753-253801, 07753-253872

E-mail: info@cvru.ac.in, Website: www.cvru.ac.in

CSBG009T
ADVANCE PYTHON

Credit- 2

Subject Expert Team

Course Edited By:

Unit Written By:

Warning: All rights reserved, No part of this publication may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the publisher.

Edition: 2026

Publish by: DR. C.V. RAMAN UNIVERSITY

Kargi Road, Kota, Bilaspur, (C. G.),

Ph. 07753-253801, 07753-253872

STRUCTURE OF PROGRAM

Title of the Program	: Bachelor of Computer Application
Program Type	: Undergraduate
Program Credits	: 120 Credits/160 Credits
Program Duration	: 3 Years / 4 Years
Eligibility Criteria	: 12th Pass

PROGRAMME OBJECTIVES

- To produce employable IT workforce, that will have sound knowledge of IT and business fundamentals that can be applied to develop and customize solutions for Small and Medium Enterprises (SME).
- To develop skilled manpower in the various areas of information technology like. Data base management, Software Development, Computer-Languages, Software engineering, Web based applications etc.
- To Work professionally with positive attitude as an individual or in multidisciplinary teams and communicate effectively.

PROGRAMME OUTCOMES

- PO 1.** Develop competence in basic technical subjects in computer applications like Programming Languages, Data Structures, Databases, Operating Systems, Software Engineering.
- PO 2.** Identify, analyze, formulate and develop computer applications.
- PO 3.** Map real life scenarios to various theoretical optimal solutions.
- PO 4.** Provide simplest automated solutions to various legacy systems.
- PO 5.** Use modern computing tools and techniques with confidence.

PROGRAMME SPECIFIC OUTCOME

Upon completion of program in Bachelor of Computer application, students will be able:

- PSO 1.** Able to implement computer skill in the areas related to multimedia and website design, Data base, Hardware and networking.
- PSO 2.** Able to demonstrate basic knowledge in the areas such as, Software Engineering, Data communication and Networking, Data base management, Web Technology and Operating Systems for building IT applications.
- PSO 3.** Students will be able to understand, analyze and develop computer programs in the areas related to algorithm, system software, web design and networking for efficient design of computer-based system.
- PSO 4.** Apply standard software engineering practices and strategies in software project development using open-source programming environment to deliver a quality of product for business success.
- PSO 5.** Student will be able to know various issues, latest trends in technology development and thereby innovate new ideas and solutions to existing problems.

STRUCTURE OF COURSE	
Course Title	: Advance Python
Course Type	: Generic Elective Course
Course Code	: CSBG009T
Course Credits	: 04
Semester	: 02

COURSE LEARNING OBJECTIVES:

- To develop foundational proficiency in NumPy for efficient array creation, indexing, and manipulation.
- To explore and implement statistical distributions using NumPy to understand their types and applications in data analysis.
- To examine and apply NumPy's Universal Functions (ufuncs) for performing fast, element-wise operations on arrays.
- To train students in using Pandas for data ingestion, cleaning, transformation, and aggregation in real-world datasets.
- To enable learners to create compelling visualizations using Matplotlib for data interpretation and analysis.

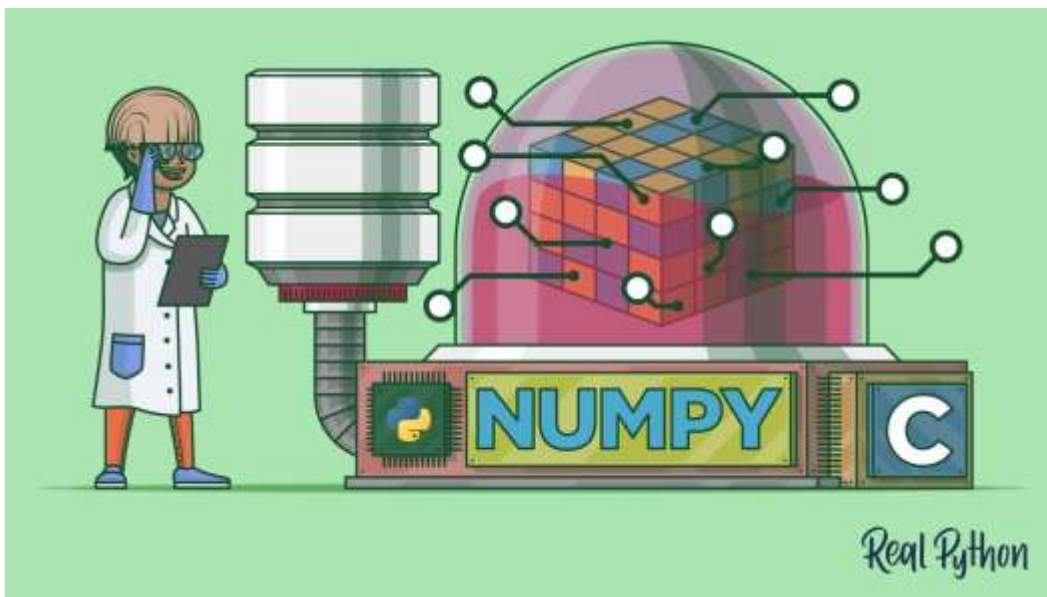
COURSE LEARNING OUTCOMES:

After completion of the course, the student would be able to

- CO-1** Elaborate the basics of Numpy in python.
- CO-2** Experiment with distribution and its types using NumPy in python.
- CO-3** Analyze the NumPy Universal Function in python.
- CO-4** Perceive the data manipulation with Pandas in python.
- CO-5** Analyze the Matplotlib in python.

Course Content		
Block 1: Introduction to NumPy		01-50
Unit 1	Basics of NumPy	03
Unit 2	Understanding NumPy	15
Unit 3	Data Manipulation in NumPy	31
Block 2: Numpy Random		51-88
Unit 4	Introduction to Data Distribution	53
Unit 5	Exploring Probability Distributions	65
Unit 6	Advanced Probability Distributions	77
Block 3: Numpy Universal Functions		89-122
Unit 7:	Understanding Numpy Ufuncs	91
Unit 8	Operations in NumPy	101
Unit 9	NumPy Mathematics	111
Block 4: Data Manipulation with Pandas		123-162
Unit 10	Introduction to Pandas	125
Unit 11	Data Cleaning in Pandas	139
Unit 12	Data Analysis with Pandas	151
Block 5: Python Matplotlib		163-186
Unit 13	Introduction to Matplotlib	165
Unit 14	Advanced Matplotlib	175

BLOCK I	INTRODUCTION TO NUMPY	01-50
Unit 1	Basics of NumPy	03
Unit 2	Understanding NumPy	15
Unit 3	Data Manipulation in NumPy	31



This block introduces students to NumPy, one of the most important Python libraries used for numerical computing, scientific calculations, and data analysis. Students will learn the basics of NumPy arrays, understand how NumPy stores and processes large datasets efficiently, and explore various functions that make numerical operations faster and easier compared to regular Python lists. The block also covers essential techniques for manipulating data, reshaping arrays, slicing, indexing, and performing vectorized operations. By the end of this block, learners will have a strong foundation in NumPy and will be ready to apply it to real-world analytical and computational tasks.

UNIT 1

BASICS OF NUMPY

NOTE

UNIT STRUCTURE:

- 1.1. Introduction
- 1.2. Objective
- 1.3. Basics of NumPy
- 1.4. Creating Arrays.
- 1.5. Indexing Arrays
- 1.6. Slicing Arrays
- 1.7. Summary
- 1.8. Glossary
- 1.9. Answers to check your progress
- 1.10. SUGGESTED READINGS – OER-OER
- 1.11. Reference
- 1.12. Exercise

1.1 INTRODUCTION

NumPy (Numerical Python) is a fundamental library for numerical computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these data structures efficiently. It is widely used in scientific computing, data analysis, machine learning, and engineering applications due to its speed and flexibility. Understanding the basics of NumPy is crucial for working with numerical data, as it enables efficient storage, manipulation, and computation.

This chapter provides an overview of NumPy, focusing on essential operations such as creating arrays, accessing elements through indexing, and extracting portions of arrays using slicing techniques. Mastering these fundamental concepts lays the groundwork for more advanced operations like broadcasting, vectorized computations, and linear algebraic manipulations.

1.2 OBJECTIVE

After completing this Unit we will be able to:

- Define NumPy and describe its features and advantages.
- Identify NumPy arrays and their types.
- Demonstrate the installation and basic usage of NumPy.
- Explain the differences between Python lists and NumPy arrays.

This chapter serves as a foundation for working with NumPy, providing the necessary skills to leverage its powerful capabilities in data manipulation and analysis.

NOTE

1.3 BASICS OF NUMPY

The array is a data structure that stores multiple values in a single variable, allowing efficient storage and retrieval of data. Unlike traditional Python lists, arrays provide better performance in numerical computations by utilizing contiguous memory locations and optimized mathematical operations.

In Python, arrays are commonly handled using the NumPy (Numerical Python) library, which offers a powerful ndarray (N-dimensional array) object for efficient operations on large datasets. Understanding the basics of arrays is essential for data science, scientific computing, machine learning, and numerical simulations.

1.3.1 Understanding Arrays

An array is a collection of elements of the same data type stored at contiguous memory locations. Arrays are useful for performing mathematical computations efficiently due to their structured storage.

Uses Arrays Instead of Lists

While Python lists are flexible, they are not optimized for numerical operations. NumPy arrays offer:

- **Better Performance:** NumPy arrays consume less memory and provide faster computation compared to Python lists.
- **Vectorized Operations:** NumPy allows element-wise operations without explicit loops, improving execution speed.
- **Support for Multi-dimensional Data:** Unlike lists, which are primarily one-dimensional, NumPy arrays can handle multi-dimensional data efficiently

1.4 CREATING ARRAYS

Arrays are fundamental data structures used to store and manipulate numerical data efficiently. In Python, the NumPy library provides the ndarray object, which is optimized for handling large, multi-dimensional arrays.

NumPy arrays offer faster computation, better memory management, and more powerful mathematical operations compared to Python lists.

In this note, we will explore different methods for creating arrays using NumPy, covering both basic and advanced techniques.

1.4.1. Creating NumPy Arrays from Python Lists and Tuples

A NumPy array can be created from a Python list or tuple using the `numpy.array()` function.

NOTE**Creating a 1D Array**

```
import numpy as np
# Creating a NumPy array from a list
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)
# Output: [1 2 3 4 5]
```

Creating a 2D Array (Matrix) from a Nested List

```
import numpy as np
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

Creating an Array from a Tuple

```
import numpy as np
arr3 = np.array((10, 20, 30))
print(arr3)
# Output: [10 20 30]
```

1.4.2. Creating Arrays Using NumPy Functions

NumPy provides several built-in functions to create arrays efficiently. These functions help to generate arrays of different shapes and values without manually specifying elements.

Creating Arrays with Zeros and Ones

These functions generate arrays filled with zeros or ones.

```
import numpy as np
# Creating an array of zeros
zeros_array = np.zeros((3, 4))
print(zeros_array)
# Output:
# [[0. 0. 0. 0.]
#  [0. 0. 0. 0.]
#  [0. 0. 0. 0.]]
```

NOTE

```
# Creating an array of ones
ones_array = np.ones((2, 5))
print(ones_array)
# Output:
# [[1. 1. 1. 1. 1.]
#  [1. 1. 1. 1. 1.]
```

Creating an Empty Array

The `empty()` function creates an array without initializing its values. The values will be arbitrary (leftover memory values).

```
import numpy as np
empty_array = np.empty((2, 3))
print(empty_array)
# Output: (Unpredictable values)
# [[4.665e-310 0.000e+000 4.665e-310]
#  [0.000e+000 0.000e+000 0.000e+000]]
```

Creating an Identity Matrix

The `eye()` function creates an identity matrix (square matrix with ones on the diagonal and zeros elsewhere).

```
import numpy as np
identity_matrix = np.eye(4)
print(identity_matrix)
# Output:
# [[1. 0. 0. 0.]
#  [0. 1. 0. 0.]
#  [0. 0. 1. 0.]
#  [0. 0. 0. 1.]
```

1.4.3. Creating Arrays with a Range of Values

NumPy provides functions to create arrays containing sequences of numbers.

Using `arange()` (Similar to Python's `range()`)

```
import numpy as np
range_array = np.arange(1, 10, 2) # Start=1, Stop=10 (exclusive), Step=2
```

NOTE

```
print(range_array)
# Output: [1 3 5 7 9]
```

Using linspace() for Evenly Spaced Values

`linspace()` generates values evenly spaced between a start and stop value.

```
import numpy as np
linspace_array = np.linspace(0, 10, 5) # 5 evenly spaced values from 0 to 10
print(linspace_array)
# Output: [ 0.  2.5  5.  7.5 10.]
```

1.4.4. Creating Random Arrays

NumPy provides the random module to generate arrays filled with random values.

Creating an Array of Random Numbers (Uniform Distribution)

```
import numpy as np
random_array = np.random.rand(3, 3) # 3x3 matrix with random values between 0 and 1
print(random_array)
```

Creating an Array of Random Integers

```
import numpy as np
random_int_array = np.random.randint(1, 100, (2, 4)) # Random integers between 1 and 100
print(random_int_array)
```

Creating a Normally Distributed Random Array

```
import numpy as np
normal_dist_array = np.random.randn(3, 3) # 3x3 matrix with normal distribution (mean=0, std=1)
print(normal_dist_array)
```

1.4.5. Creating Arrays with a Specific Data Type

NumPy allows setting a specific data type for arrays using the `dtype` parameter.

```
import numpy as np
int_array = np.array([1.5, 2.7, 3.9], dtype=int) # Converts float values to integers
```

NOTE

```
print(int_array)
# Output: [1 2 3]

float_array = np.array([1, 2, 3], dtype=float)
print(float_array)
# Output: [1. 2. 3.]
```

1.4.6. Reshaping Arrays

After creating an array, its shape can be modified using `reshape()`.

```
import numpy as np
arr = np.arange(1, 10)
reshaped_array = arr.reshape(3, 3)
print(reshaped_array)
# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]
```

1.4.7. Copying and Cloning Arrays**Creating a View (Shallow Copy)**

A **view** shares the same data as the original array. Modifications affect both.

```
import numpy as np
arr = np.array([10, 20, 30])
view_arr = arr.view()
view_arr[0] = 100
print(arr) # Output: [100 20 30] (Changes reflect in original array)
```

Creating a Deep Copy

A **deep copy** creates an independent duplicate.

```
import numpy as np
arr = np.array([10, 20, 30])
copy_arr = arr.copy()
copy_arr[0] = 100
print(arr) # Output: [10 20 30] (Original array remains unchanged)
```

NOTE

This unit serves as a foundation for understanding the essential concepts and techniques in programming, providing both theoretical knowledge and practical insights into effective programming practices.

**Activity 1.1**

Write a Python program to create a 1D NumPy array of 10 random integers between 1 and 50.

Display the array and print its data type, size, and shape.

.....

.....

.....

☑ Check Your Progress 1.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. How can you create a 2D NumPy array with values ranging from 1 to 9, and access the element in the second row and third column?

.....

.....

.....

Q.2 Explain the difference between indexing and slicing in NumPy arrays with suitable examples.

.....

.....

.....

1.5 SUMMARY

NumPy is a powerful Python library used for numerical computing, and a strong grasp of its array operations is essential for effective data analysis. The basics include creating arrays using functions like `np.array()`, `np.zeros()`, `np.ones()`, and `np.arange()` to generate arrays with specific shapes or values. Indexing allows access to individual elements or subsets of arrays using zero-based positions, while slicing enables the extraction of subarrays through a `start:stop:step` syntax. These operations make data manipulation efficient and form the foundation for more advanced numerical and scientific computations in Python.

1.6 GLOSSARY

- **NumPy:** A Python library used for numerical computations, particularly efficient for working with large, multi-dimensional arrays and matrices.

NOTE

- **Array:** A grid of values (all of the same type) indexed by a tuple of nonnegative integers; NumPy's core data structure.
- **ndarray:** The primary object in NumPy, representing an N-dimensional array.
- **np.array():** A NumPy function used to create an array from Python lists or tuples.
- **np.zeros():** Creates a new array of a specified shape filled with zeros.
- **np.ones():** Creates a new array of a specified shape filled with ones.
- **np.arange():** Generates an array with evenly spaced values within a given range.
- **Indexing:** The process of accessing specific elements within a NumPy array using their position (e.g., `array[2]` or `array[1,3]`).
- **Slicing:** A technique to extract a portion of an array using a range of indices (e.g., `array[1:5]`), often in the format `start:stop:step`.
- **Shape:** A tuple that defines the dimensions of a NumPy array (e.g., a 3x4 array has a shape of (3, 4)).
- **Dimension (Axis):** Refers to the direction along which data is arranged in an array; 1D arrays have one axis, 2D arrays have two, and so on.
- **Broadcasting:** A method used by NumPy to perform operations on arrays of different shapes by automatically expanding them to a common shape when possible.
- **Flattening:** Converting a multi-dimensional array into a one-dimensional array using methods like `.flatten()` or `.ravel()`.
- **Boolean Indexing:** Selecting elements in an array using Boolean values or conditions (e.g., `array[array > 5]`).
- **Multidimensional Array:** An array with more than one axis (e.g., 2D matrices or 3D tensors), used for organizing complex data.

1.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1.1**Answer 1**

We can create a 2D NumPy array using `np.arange()` and `reshape()` functions.

Step-by-step Explanation:

- **Step 1: Import NumPy**

Before working with NumPy, you need to import the library:

```
import numpy as np
```

- **Step 2: Create an array with values from 1 to 9**

Use `np.arange(1, 10)` to generate a 1D array from 1 to 9 (note that the stop value 10 is not included). Then, use `.reshape(3, 3)` to convert it into a 2D array of shape 3 rows × 3 columns:

```
array = np.arange(1, 10).reshape(3, 3)
```

Now array looks like this:

NOTE

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

- **Step 3: Access the element in the second row and third column**

In NumPy, indexing starts from **0**, so:

- Second row → index 1
- Third column → index 2

```
element = array[1, 2]
print(element) # Output: 6
```

So, the value at row 1, column 2 is 6.

Answer 2**Indexing:**

- **Definition:** Indexing refers to accessing a single element from a NumPy array using its position (index).
- **Syntax:** For a 2D array, use `array[row_index, column_index]`.

Example:

```
import numpy as np

array = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])

# Access element at 2nd row, 3rd column

print(array[1, 2]) # Output: 60
```

Here, `[1, 2]` accesses the element at row index 1 (second row) and column index 2 (third column).

Slicing:

- **Definition:** Slicing is used to extract a subarray (a portion of the array) using a range of indices.
- **Syntax:** `array[start:stop:step]`. For 2D arrays, it becomes `array[row_start:row_stop, col_start:col_stop]`.

Example:

```
# Extract subarray: rows 0 and 1, columns 1 and 2
```

```
subarray = array[0:2, 1:]
print(subarray)
```

NOTE

Output:

[[20 30]

[50 60]]

Explanation:

- 0:2 → rows from index 0 to 1 (excluding 2)
- 1: → columns from index 1 to the end

Table 1.1: Summary of Differences

Feature	Indexing	Slicing
Purpose	Access a single element	Access a range of elements (subarray)
Syntax	array[i, j]	array[start:stop, start:stop]
Example	array[2, 1] → gets one value	array[0:2, 1:] → gets subarray
Output Shape	Scalar (number)	Subarray (array object)

1.8 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/UnitUrls?unitId=1262	

1.9 REFERENCE

- <https://www.oreilly.com/library/view/python-for-data/9781491957653/>
- <https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/>

1.10 EXERCISE**Part 1****A. Multiple choice question:****1. Which of the following statements about NumPy is correct?**

- A) NumPy is a Python library used for numerical computations.
- B) NumPy arrays are slower than Python lists.
- C) NumPy does not support multidimensional arrays.

NOTE

D) NumPy arrays cannot store homogeneous data types.

2. Which function in NumPy is used to create an array from a Python list?

- A) np.array()
- B) np.asarray()
- C) Both a and b
- D) np.list()

3. Consider the array:

```
import numpy as np
```

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

What is the output of arr[2]?

- A) 10
- B) 20
- C) 30
- D) 40

4. Given the array:

```
arr = np.array([5, 10, 15, 20, 25, 30])
```

What is the output of arr[1:4]?

- A) [5, 10, 15]
- B) [10, 15, 20]
- C) [15, 20, 25]
- D) [20, 25, 30]

5. Given the 2D array:

```
arr = np.array([[1, 2, 3],
```

```
[4, 5, 6],
```

```
[7, 8, 9]])
```

What will arr[0:2, 1:3] return?

- A) [[1, 2], [4, 5]]
- B) [[2, 3], [5, 6]]
- C) [[4, 5], [7, 8]]
- D) [[3, 6], [9, 12]]

Answer with Explanations:

1. A) NumPy is a Python library used for numerical computations.

Explanation: NumPy is designed specifically for fast numerical computations in Python using optimized C-based operations.

2. C) Both a and b

Explanation: Both functions convert Python lists into NumPy arrays, so either one can create an array.

NOTE**3. C)** 30**Explanation:** Index 2 refers to the third element of the array, which is 30.**4. B)** [10, 15, 20]**Explanation:** `arr[1:4]` slices elements from index 1 to 3, giving 10, 15, and 20.**5. B)** [[2, 3], [5, 6]]**Explanation:** The slice selects rows 0–1 and columns 1–2, returning these sub-matrix values.**B. Short answer type question:**

1. What is NumPy, and why is it preferred over Python lists for numerical operations?
2. Write a Python statement using NumPy to create a 1D array containing the elements [2, 4, 6, 8, 10].
3. Given the array:

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

What is the output of `arr[-2]`?**Part 2****C. Long answer type question: (200 words)**

1. Explain the concept of array slicing in NumPy with suitable examples. Write code snippets to show how to:
 - Extract a subarray from a 1D array.
 - Extract a block of elements from a 2D array.
2. Differentiate between array indexing and array slicing in NumPy. Write Python examples for each case using both 1D and 2D arrays.

Part 3**D. Long Answer Type Questions: (300 words)**

1. Explain the importance of NumPy in scientific computing. Discuss how NumPy arrays differ from Python lists, emphasizing memory efficiency, speed, and functionality.
2. Describe the structure and features of a NumPy ndarray. Explain its attributes such as shape, size, ndim, dtype, and itemsize, with examples.
3. Discuss the process of installing NumPy, creating basic arrays, and performing simple array operations. Explain how these operations form the foundation for advanced numerical computing.

UNIT 2

UNDERSTANDING NUMPY

NOTE

UNIT STRUCTURE:

- 2.1 Introduction
- 2.2 Objective
- 2.3 Understanding NumPy
- 2.4 NumPy Data Types
- 2.5 NumPy Views
- 2.6 NumPy Shape Manipulation
- 2.7 NumPy Iteration
- 2.8 Summary
- 2.9 Glossary
- 2.10 Answers to Check Your Progress.
- 2.11 SUGGESTED READINGS – OER -OER
- 2.12 Reference
- 2.13 Exercise

2.1 INTRODUCTION

NumPy (Numerical Python) is a fundamental library for scientific computing in Python. It provides a powerful n-dimensional array object called ndarray, which enables fast, efficient operations on large datasets. NumPy plays a critical role in data science, machine learning, image processing, and numerical simulations due to its speed and versatility. Understanding its core concepts is essential for writing high-performance, scalable code.

This unit focuses on four foundational aspects of NumPy: data types, views, shape manipulation, and iteration. Learners will explore how NumPy handles different data types for optimized memory usage, how views differ from copies for efficient data access, how to reshape and transform arrays to fit computation needs, and how to iterate over arrays effectively. Mastery of these concepts will provide the skills needed to process and manipulate large datasets with ease.

2.2 OBJECTIVE

After completing this Unit we will be able to:

- Explain array attributes such as shape, size, and data types.
- Apply indexing and slicing techniques on NumPy arrays.
- Differentiate between one-dimensional and multi-dimensional arrays.
- Interpret array operations and broadcasting concepts

NOTE

2.3 UNDERSTANDING NUMPY

NumPy is a powerful Python library for numerical and scientific computing. It provides an efficient array structure called `ndarray` for handling large datasets. Unlike Python lists, NumPy arrays are faster and consume less memory. They support element-wise operations and broadcasting. NumPy enables vectorized operations, eliminating slow Python loops. It is widely used in data analysis, machine learning, and simulations. NumPy supports various data types for optimized performance. It allows reshaping, slicing, and transposing arrays easily. Views and copies help manage memory efficiently. Iteration tools like `nditer` enable fast element access.

NumPy is the foundation of scientific computing in Python and is used extensively in:

- 1. Data Analysis:** Data analysis is the process of inspecting, cleaning, transforming, and modeling data to discover useful information, patterns, and insights. Provides fast operations on large arrays of data (millions of rows). Enables statistical calculations like mean, median, standard deviation. Forms the core of Pandas, a popular data analysis library built on top of NumPy. Offers tools for data cleaning and transformation (filtering, masking, reshaping). Example: Calculating average temperature readings from a large dataset using `np.mean()`.
- 2. Machine Learning:** Machine learning is the use of algorithms to automatically learn patterns from data and make decisions or predictions. Stores feature data as matrices (datasets are typically in 2D array format). Supports fast matrix operations used in training algorithms (dot products, transposes, inverses). Used internally by ML libraries like Scikit-learn, TensorFlow, and PyTorch. Enables preprocessing tasks like normalization, reshaping, and vectorization. Example: Using NumPy to calculate the dot product between input features and weights in a neural network layer.
- 3. Image Processing:** Image processing involves performing operations on images to enhance them or extract useful information (like edges, shapes, colors). Images are represented as arrays (2D for grayscale, 3D for RGB). Enables pixel-level operations (filtering, transformation, masking). Works with libraries like OpenCV and Pillow, which convert images to NumPy arrays. Fast element-wise operations for tasks like brightness adjustment or thresholding. Example: Applying a grayscale filter or edge-detection mask to an image using NumPy array operations.
- 4. Statistical Computing:** Statistical computing focuses on applying statistical methods to analyze and interpret data. Offers built-in statistical functions (mean, median, std, var, `corrcoef`, etc.). Allows bootstrapping, sampling, and simulation techniques. Supports handling of large datasets with consistent precision. Works alongside SciPy for advanced statistical tests. Example: Calculating the correlation matrix of several financial indicators using `np.corrcoef()`.
- 5. Simulations and Numerical Modeling:** Simulations replicate real-world processes mathematically, while numerical modeling solves complex

NOTE

mathematical equations that cannot be solved analytically. Handles large-scale numerical computations efficiently. Supports differential equations, matrix algebra, and random number generation. Used with SciPy, Matplotlib, or custom code to model real systems (weather, traffic, finance). Allows Monte Carlo simulations using random sampling (np.random module). Example: Simulating the spread of a virus over a population using probabilistic models and NumPy arrays.

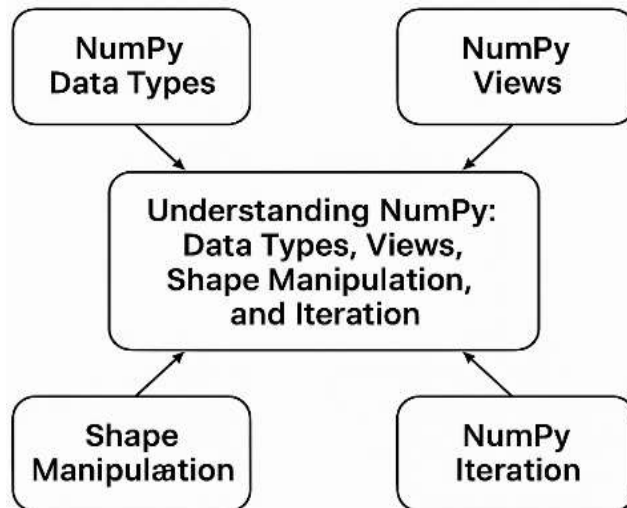


Figure 2.1: NumPy Description

2.4 NUMPY DATA TYPES

NumPy data types, known as dtypes, define the type of values stored in an array, such as integers, floats, booleans, or complex numbers. Unlike Python lists, NumPy arrays require all elements to be of the same data type, which enables faster and more efficient computation. Common data types include int32, float64, bool, and complex64, each serving different purposes. For example, float32 is often used in machine learning for balancing precision and memory usage. NumPy also supports unsigned integers (e.g., uint8) and flexible types like strings and objects. You can check an array's data type using `.dtype` and change it with `.astype()`. Using the right data type helps manage memory more efficiently, especially with large datasets. Understanding dtypes is essential for scientific computing, simulations, and high-performance applications. It also ensures compatibility when working with other libraries like Pandas or TensorFlow. Overall, mastering NumPy data types allows better control over numerical accuracy and resource usage.

Unlike Python lists, where each element can be of a different type,

1. **NumPy arrays are homogeneous** - NumPy arrays are homogeneous, meaning all elements in an array must be of the same data type. This uniformity allows NumPy to store data efficiently in memory and perform fast, vectorized operations. Unlike Python lists, NumPy does not allow mixed data types within a single array.

NOTE

```
import numpy as np

arr = np.array([1, 2, 3.5, 4])

print(arr)      # Output: [1.  2.  3.5  4. ]
print(arr.dtype) # Output: float64
```

Note: Even though 1 and 2 are integers, the entire array becomes float64 to maintain homogeneity.

- 2. Memory Efficiency:** Memory efficiency in NumPy comes from its use of fixed-size, homogeneous data types and contiguous memory allocation. This makes NumPy arrays significantly more compact and faster than Python lists, especially for large datasets. By specifying appropriate data types, we can further reduce memory usage and improve performance.

```
import numpy as np
import sys

list_data = [1, 2, 3, 4, 5]
array_data = np.array([1, 2, 3, 4, 5], dtype=np.int32)

print("List memory (bytes):", sys.getsizeof(list_data))    # e.g., 96 bytes
print("NumPy array memory (bytes):", array_data.nbytes)   # e.g., 20 bytes
```

Result: NumPy uses less memory due to its compact, type-specific storage.

- 3. Speed:** NumPy provides high-speed numerical operations by using optimized C-based backend and vectorized operations. It eliminates the need for Python loops by performing element-wise computations directly on entire arrays. This makes NumPy significantly faster than native Python lists for large-scale numerical tasks.

```
import numpy as np
import time

a = list(range(1000000))
b = list(range(1000000))
start = time.time()
result = [x + y for x, y in zip(a, b)]
print("List time:", time.time() - start)
```

NOTE

```
a_np = np.arange(1000000)
b_np = np.arange(1000000)
start = time.time()
result_np = a_np + b_np
print("NumPy time:", time.time() - start)
```

Result: NumPy performs the addition much faster than the Python list.

- 4. Precision Control:** Precision control in NumPy allows you to choose data types like float32, float64, or even float16 depending on the accuracy and memory trade-offs you need. This is especially important in scientific computing, where rounding errors can affect results. Using the correct precision helps balance speed, memory use, and numerical accuracy.

```
import numpy as np
a = np.array([1.123456789], dtype=np.float32)
b = np.array([1.123456789], dtype=np.float64)
print("Float32:", a) # Output: [1.1234568]
print("Float64:", b) # Output: [1.12345679]
```

Result: float64 retains more decimal precision than float32.

- 5. Compatibility:** Compatibility in NumPy refers to its ability to seamlessly interact with other scientific libraries like Pandas, TensorFlow, and SciPy. NumPy arrays are often the standard input and output format across these tools, ensuring smooth data exchange. This makes it easier to build end-to-end data pipelines using multiple libraries.

```
import numpy as np
import pandas as pd
arr = np.array([10, 20, 30])
df = pd.DataFrame(arr, columns=["Values"])
print(df)
```

Result: NumPy array is directly converted into a Pandas DataFrame, demonstrating compatibility.

Table 2.1: Common Numpy Data Types

Data Type	Description	Example Value
int8	Integer (-128 to 127)	np.int8(10)
int16	Integer (-32,768 to 32,767)	np.int16(200)

NOTE

int32	Integer (-2 billion to 2 billion)	np.int32(1000)
int64	Larger integer	np.int64(1e10)
float16	Half-precision float	np.float16(3.14)
float32	Single-precision float	np.float32(3.1415)
float64	Double-precision float (default)	np.float64(3.1415926)
bool	Boolean (True or False)	np.bool_(True)
complex64	Complex number (32-bit float real + imaginary)	np.complex64(1+2j)
str	Unicode string	np.str_('hello')

2.5 NUMPY VIEWS

In NumPy, a view is a new array object that looks at the same data as the original array. It does not create a copy of the data but simply provides a different perspective or shape of the same memory block. This makes views very memory-efficient and faster than copies. Any changes made to a view directly reflect in the original array, and vice versa. Views are created automatically during slicing or reshaping operations unless explicitly copied. Understanding views helps prevent unintended data modifications when manipulating arrays. They are especially useful when working with large datasets where memory conservation is critical. However, if independent data manipulation is required, a full copy should be made. The `.view()` method can also be used to create views manually. Overall, views are a powerful feature of NumPy that allow flexible data handling with minimal resource usage.

Key Aspects of Algorithms:

- 1. Shared Memory Handling:** NumPy views operate on the same memory block as the original array using pointer-based references, which makes them lightweight and efficient.
- 2. Stride Manipulation:** Views are formed by changing the strides of an array, which define how many bytes to step in memory to move to the next element in each dimension.
- 3. No Data Duplication:** Since views don't duplicate data, they enable low-overhead transformations, ideal for large-scale data without performance loss.
- 4. Automatic View Creation:** Basic operations like slicing, reshaping (when possible), and transposing often create views instead of copies to improve speed.

NOTE

5. **Safe vs. Unsafe Modifications:** Algorithms using views must manage side effects carefully, as modifying a view will reflect in the original array unless a `.copy()` is used.

Importance of Algorithms:

1. **Enhances Performance:** Algorithms that use views avoid unnecessary data copying, making operations faster and more memory-efficient—crucial for large-scale numerical tasks.
2. **Reduces Memory Usage:** Since views share memory with the original array, algorithms can manipulate large datasets without allocating extra memory, improving scalability.
3. **Supports In-Place Processing:** Views allow algorithms to modify data directly in memory, enabling real-time updates and reducing computational overhead.
4. **Improves Data Transformation Efficiency:** Tasks like slicing, reshaping, and broadcasting become lightweight when implemented through views, speeding up array transformations in algorithms.
5. **Foundation for Advanced Operations:** Many high-level NumPy operations, such as broadcasting or lazy evaluation in other libraries (like Dask), rely on the view concept to maintain performance.

Algorithms that utilize NumPy views are essential for writing high-performance, memory-efficient code. By operating directly on shared memory, these algorithms avoid unnecessary duplication, enabling faster computation and lower resource usage. This is especially beneficial when working with large arrays or in real-time data processing.

Views also allow flexible data manipulation through slicing, reshaping, and iteration, without copying the data. Efficient algorithms built on this concept improve the speed of numerical operations and serve as the foundation for advanced techniques in scientific computing and machine learning.

2.6 NUMPY SHAPE MANIPULATION

NumPy shape manipulation involves changing the dimensions or layout of an array without modifying its data. With functions like `reshape()`, users can convert a 1D array into a 2D matrix or even a 3D structure, as long as the total number of elements remains the same. This is particularly helpful in machine learning, where input data often needs to be reshaped to match model requirements.

Additional tools like `flatten()`, `ravel()`, and `transpose()` allow further control over array structure. `flatten()` creates a copy in 1D, while `ravel()` returns a view when possible. `transpose()` changes the axes order of an array, useful in matrix operations. Shape manipulation improves data alignment, processing efficiency, and code readability. Since many of these operations return views, they also optimize memory usage.

NOTE

1. **Reshaping Arrays:** NumPy's `reshape()` allows you to change the shape of an array without changing its data. It's useful for converting data between dimensions, like 1D to 2D or 2D to 3D. The total number of elements must remain the same.
2. **Flattening Arrays:** Functions like `flatten()` and `ravel()` convert multi-dimensional arrays into 1D arrays. `flatten()` returns a copy, while `ravel()` returns a view when possible. This is helpful when preparing data for linear operations.
3. **Transposing Arrays:** Using `transpose()` or `.T` changes the order of axes in an array. It's commonly used in matrix operations like switching rows and columns. Transposing does not modify the original data, only its view.
4. **Adding or Removing Dimensions:** With `np.expand_dims()` or `np.newaxis`, you can add new axes to an array. This is useful in broadcasting or shaping data for models. Similarly, `squeeze()` removes single-dimensional entries.
5. **Memory Efficiency with Views:** Many shape manipulations return views, not copies. This allows fast reshaping without consuming extra memory. It boosts performance in large-scale numerical operations.

2.7 NUMPY ITERATION

NumPy iteration refers to the process of accessing each element of an array, typically using loops. Although Python's for loop can iterate through NumPy arrays, it's often less efficient than vectorized operations. However, when element-wise processing is required, NumPy provides tools like `nditer()` for flexible, efficient iteration.

The `nditer()` object allows iteration across multi-dimensional arrays in a consistent manner. It supports options like read-write access, buffering, and broadcasting during iteration. Additionally, you can iterate with index using `ndenumerate()` or over specific axes with `np.apply_along_axis()`. These tools help write cleaner, faster code for element-wise operations. Efficient iteration is key when performance matters and full vectorization isn't feasible.

1. **Reshape an Array:** You can reshape a 1D array into 2D using **`reshape()`**, keeping the total number of elements the same. This is useful when preparing data for matrix operations.

```
import numpy as np
a = np.array([1, 2, 3, 4, 5, 6])
b = a.reshape(2, 3) # 2 rows, 3 columns
```

2. **Flatten an Array:** Use **`flatten()`** to convert a multi-dimensional array into a 1D array (returns a copy). It's useful for linear models or storing output.

NOTE

```
arr = np.array([[1, 2], [3, 4]])
flat = arr.flatten()
```

3. Transpose an Array: The **transpose()** function (or **.T**) swaps rows and columns.

This is commonly used in linear algebra and image processing.

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
transposed = matrix.T
```

4. Add a New Axis: **np.newaxis** or **expand_dims()** adds a new dimension to an array.

Useful for changing shape for broadcasting or model input.

```
vec = np.array([10, 20, 30])
column_vec = vec[:, np.newaxis] # Shape becomes (3, 1)
```

**Activity 2.1**

TITLE OF ACTIVITY: Operations on Two-Dimensional NumPy Arrays

OBJECTIVE: To develop practical proficiency in creating multi-dimensional arrays, executing array slicing, and utilizing statistical reduction functions within the NumPy library.

MATERIAL NEEDED: A computer or smartphone equipped with Python and the NumPy library installed (using an environment like Jupyter Notebook, VS Code, or Google Colab), and a digital notebook.

ACTIVITY: Create a 3×3 two-dimensional NumPy array containing numeric values of your choice. Write a Python script to execute and display the outputs for the following tasks:

1. Extract and print the entire first row of your matrix.
2. Isolate and print the very last column of your matrix.
3. Determine and print both the absolute maximum and minimum values present across the entire array.

EXPECTATION: You should write a clean, error-free Python code snippet using correct NumPy slicing syntax (such as `array[0, :]` for the first row and `array[:, -1]` for the last column) along with standard aggregation methods like `.max()` and `.min()`. Your final output must show the code alongside its executed results, labeled clearly with descriptive print statements.

NOTE

REFLECTION: Think about how slicing a two-dimensional NumPy array compares to indexing a standard Python nested list. Why do you think this direct, multi-dimensional coordinate slicing mechanism makes NumPy vastly superior and more efficient when working with larger business datasets or mathematical matrices?

☑ Check Your Progress 2.1

Notes:

- a) Space is given below for your answer.
- b) Check your answer with the one given at the end of this unit.

Q1. What are Mastering arrays is the first step toward mastering data?

.....

Q2. In NumPy, algorithms define how data flows through operations, while notations like slicing, indexing, and reshaping act as the language that makes array manipulation precise, readable, and efficient?

.....

2.8 SUMMARY

Understanding NumPy involves mastering its core components—data types, views, shape manipulation, and iteration—which are essential for efficient numerical computation in Python. These concepts allow developers and data scientists to work with large datasets seamlessly, ensuring performance, memory optimization, and code readability.

- 1. Data Types:** NumPy supports a wide range of data types (integers, floats, complex, etc.), enabling precision and efficiency. Using the correct data type reduces memory usage and speeds up computation.
- 2. Views:** Views are lightweight references to the original array data, not copies. They allow memory-efficient operations by avoiding duplication.
- 3. Shape Manipulation:** You can reshape, flatten, or transpose arrays using built-in functions. Shape manipulation helps align data to fit models or operations.
- 4. Iteration:** NumPy supports fast iteration using tools like `nditer()` and `ndenumerate()`. These help process each array element with control and flexibility.

NOTE

5. **Memory Optimization:** Shape manipulation and views work together to minimize memory usage. This is crucial for handling large-scale data without slowing performance.
6. **Practical Applications:** These concepts are foundational in data analysis, machine learning, and simulations. They provide a structured, efficient approach to working with array data.

Programming Aids: Programming aids refer to tools and techniques that assist in the software development process, making programming more efficient and manageable. These include:

1. **dtype Attribute:** Use `array.dtype` to check or set the data type of elements in a NumPy array. Helps you ensure precision and memory efficiency.
2. **view() and copy() Methods:** Use `.view()` to create a lightweight reference and `.copy()` to avoid shared memory. This helps distinguish between views and independent data blocks.
3. **reshape(), flatten(), transpose() Functions:** Built-in functions like `reshape()`, `flatten()`, and `transpose()` simplify shape manipulation. They are essential for preparing data for machine learning models or matrix operations.
4. **nditer() and ndenumerate() for Iteration:** These tools offer structured and efficient iteration over multi-dimensional arrays. They allow element-wise access with or without index control.
5. **np.newaxis and np.expand_dims():** Use these for adding dimensions to arrays, crucial for broadcasting and model compatibility. They aid in reshaping data without affecting its content.

Algorithms and Notations: Algorithms are step-by-step procedures or formulas for solving problems, and notations are the methods used to represent them. Key aspects include:

1. **Data Type Handling Algorithm (astype() Notation):** Algorithmically convert array elements to a specific type for precision or memory optimization.
Notation: `array.astype(np.float32)` – converts all elements to 32-bit float.
2. **View vs Copy Detection (view() and copy() Use):** Design algorithms to operate on views for speed or copies for data safety.
Notation: `b = a.view()` (no data copied), `b = a.copy()` (data copied).
3. **Shape Transformation Algorithm (reshape() Notation):** Used to adapt data shape before feeding into models or matrix operations.
Notation: `array.reshape(rows, columns)` – must match total elements.
4. **Flattening and Transposing (flatten(), transpose()):** Flatten for linearization, transpose for axis switching in matrix logic.
Notation: `array.flatten()` or `array.T` (shortcut for transpose).
5. **Efficient Iteration Algorithm (nditer() Notation):** Create element-wise processing loops that are fast and memory-aware.

NOTE

Notation: for x in np.nditer(array): – accesses each element sequentially.

6. **Axis-wise Operation (apply_along_axis()):** Apply functions along specified axes for structured data analysis.

Notation: np.apply_along_axis(func, axis, array) – apply func row-wise or column-wise.

2.9 GLOSSARY

- **Array Indexing** – The method of accessing individual elements in a NumPy array using their positions.
- **Array Slicing** – The technique of extracting a portion of an array using a specified range of indices.
- **Reshaping** – Changing the dimensions of an array without altering its data.
- **Flattening** – Converting a multi-dimensional array into a one-dimensional array.
- **Transpose** – An operation that swaps the rows and columns of a two-dimensional array.
- **Iteration** – The process of accessing array elements one at a time using loops or iterators.
- **Copy** – A duplicate array that occupies a separate memory location from the original array.
- **View** – A new array object that shares the same memory as the original array.
- **Array Attributes** – Properties such as shape, size, dimensions, and data type that describe an array.
- **Memory Efficiency** – The optimized use of memory resources while storing and processing arrays.

2.10 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 2.1

Answer 1:

Mastering arrays is essential for anyone working with data, as arrays form the backbone of numerical computing in Python, particularly with NumPy. Arrays allow for the efficient storage, manipulation, and processing of large volumes of structured data. By understanding how to create, reshape, slice, and iterate over arrays, you gain the ability to perform complex operations with simple, readable code. This skill is fundamental not only for data analysis and machine learning but also for tasks like simulations, image processing, and statistical modeling. In essence, mastering arrays empowers you to work with data more effectively, accurately, and efficiently. First Step Toward Mastering Data Points:

- **Structured Data Representation:** Arrays provide a clean and organized format for storing tabular, numerical, or multi-dimensional data.

NOTE

- **Efficient Data Processing:** Arrays allow for fast, vectorized operations — essential for handling large datasets with high performance.
- **Foundation for Data Manipulation:** Tasks like filtering, slicing, reshaping, and aggregating are easily performed using array techniques.
- **Seamless Integration with Tools:** Arrays are the standard input/output format for libraries like Pandas, SciPy, scikit-learn, and TensorFlow.
- **Crucial for Model Readiness:** Machine learning and AI models require data in specific array shapes; mastering arrays ensures compatibility and correctness.

Answer 2:

In NumPy, algorithms refer to the step-by-step logic used to process and manipulate data — like sorting, transforming, or calculating over arrays. These algorithms operate on arrays using efficient backend implementations that handle large-scale data quickly.

Notations such as slicing (`array[1:5]`), indexing (`array[0, 2]`), and reshaping (`array.reshape(2, 3)`) provide a concise and readable syntax for performing these operations. They act as a specialized language that allows you to control the flow and structure of data within NumPy arrays without writing complex code.

Together, algorithms define what happens, and notations define how it's expressed — making NumPy both powerful and user-friendly for scientific and numerical computing.

1. **Slicing for Subset Selection:** Slicing is a notation that helps extract portions of data from arrays — clean and readable.

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
print(arr[1:4]) # Output: [20 30 40]
```

2. **Indexing for Direct Access:** Indexing allows precise access to elements in multidimensional arrays.

```
matrix = np.array([[1, 2], [3, 4]])
print(matrix[1, 0]) # Output: 3
```

3. **Reshaping for Structural Transformation:** `reshape()` is an essential notation for changing array shape without changing data.

```
a = np.array([1, 2, 3, 4, 5, 6])
reshaped = a.reshape(2, 3)
print(reshaped)
# Output:
```

NOTE

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

- 4. Iteration Using `nditer()`:** **`nditer()`** enables element-wise traversal of arrays regardless of their dimension.

```
b = np.array([[1, 2], [3, 4]])
for x in np.nditer(b):
    print(x, end=' ') # Output: 1 2 3 4
```

- 5. Type Conversion with `astype()`:** Algorithms often require type uniformity. **`astype()`** provides an easy, readable conversion method.

```
float_arr = np.array([1.2, 3.5, 6.7])
int_arr = float_arr.astype(int)
print(int_arr) # Output: [1 3 6]
```

These examples show how **notation** in NumPy acts like a compact, readable syntax for executing powerful **algorithms** underneath — all while keeping your code clean and efficient.

2.11 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/Use/UnitUrls?unitId=1263	

2.12 REFERENCE

- [https://www.oreilly.com/library/view/python-for data/9781491957653/](https://www.oreilly.com/library/view/python-for-data/9781491957653/)
- <https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/>

2.13 EXERCISE**Part 1****A. Multiple choice question:**

- 1. What is the primary advantage of NumPy arrays over Python lists?**

A) They store text data more efficiently

NOTE

- B) They allow variable-sized elements
 C) They are more memory- and computation-efficient
 D) They support only single-dimensional data
- 2. Which NumPy function is used to change the shape of an array without changing its data?**
- A) reshape()
 B) resize()
 C) reshape_into()
 D) reform()
- 3. What is a NumPy "view"?**
- A) A copy of the original array with reversed elements
 B) A different array stored separately in memory
 C) A new array that references the same data as the original
 D) An array that can only be read, not modified
- 4. Which function is commonly used for iterating through NumPy arrays efficiently?**
- A) for_each()
 B) nditer()
 C) array_loop()
 D) loop_array()
- 5. What does the data type dtype=int32 specify in NumPy?**
- A) The array stores 32-character strings
 B) The array stores 32-bit integer values
 C) The array is 32 dimensions
 D) It limits the array to 32 values

Answer with Explanations:

- 1. C)** They are more memory- and computation-efficient

Explanation: NumPy arrays are faster and use less memory because they store data in contiguous blocks.

- 2. A)** reshape()

Explanation: reshape() changes the structure of an array without altering its stored data.

- 3. C)** A new array that references the same data as the original

Explanation: A view refers to the same underlying data, so modifying it also affects the original array.

NOTE**4. B)** `nditer()`

Explanation: `nditer()` efficiently iterates through NumPy arrays, even multidimensional ones.

5. B) The array stores 32-bit integer values

Explanation: `dtype=int32` means each element is stored as a 32-bit (4-byte) integer.

B. Short answer type question:

1. What is the difference between a NumPy view and a copy?
2. How can you change the shape of a NumPy array?
3. Why is iteration in NumPy more efficient using `nditer()` than a regular Python loop?

Part 2**C. Long answer type question: (200 words)**

1. Explain the role and importance of data types in NumPy arrays. How do they affect performance and memory?
2. What are NumPy views? How are they created, and why are they important in data manipulation?
3. Describe NumPy shape manipulation operations and their significance in scientific computing. Provide examples.

Part 3**D. Long Answer Type Questions: (300 words)**

1. Explain array indexing and slicing in NumPy. Discuss different indexing techniques such as basic indexing, boolean indexing, and fancy indexing, with suitable examples.
2. Describe NumPy's broadcasting mechanism. Explain how broadcasting rules work and analyze situations where broadcasting simplifies array operations.
3. Discuss the concept of vectorization in NumPy. Compare vectorized operations with traditional Python loops and explain why NumPy provides significant performance improvements.

UNIT-3

DATA MANIPULATION IN NUMPY

NOTE

UNIT STRUCTURE

- 3.1 Introduction**
- 3.2 Objective**
- 3.3 Data Manipulation in NumPy**
- 3.4 Array Joining Techniques in NumPy**
- 3.5 Splitting Arrays: Breaking Data into Parts**
- 3.6 Efficient Searching in NumPy Arrays**
- 3.7 Sorting Arrays: Organizing Data Efficiently**
- 3.8 Filtering Data with Boolean Indexing**
- 3.9 Summary**
- 3.10 Glossary**
- 3.11 Answers to Check Your Progress**
- 3.12 SUGGESTED READINGS – OER-OER**
- 3.13 Reference**
- 3.14 Exercise**

3.1 INTRODUCTION

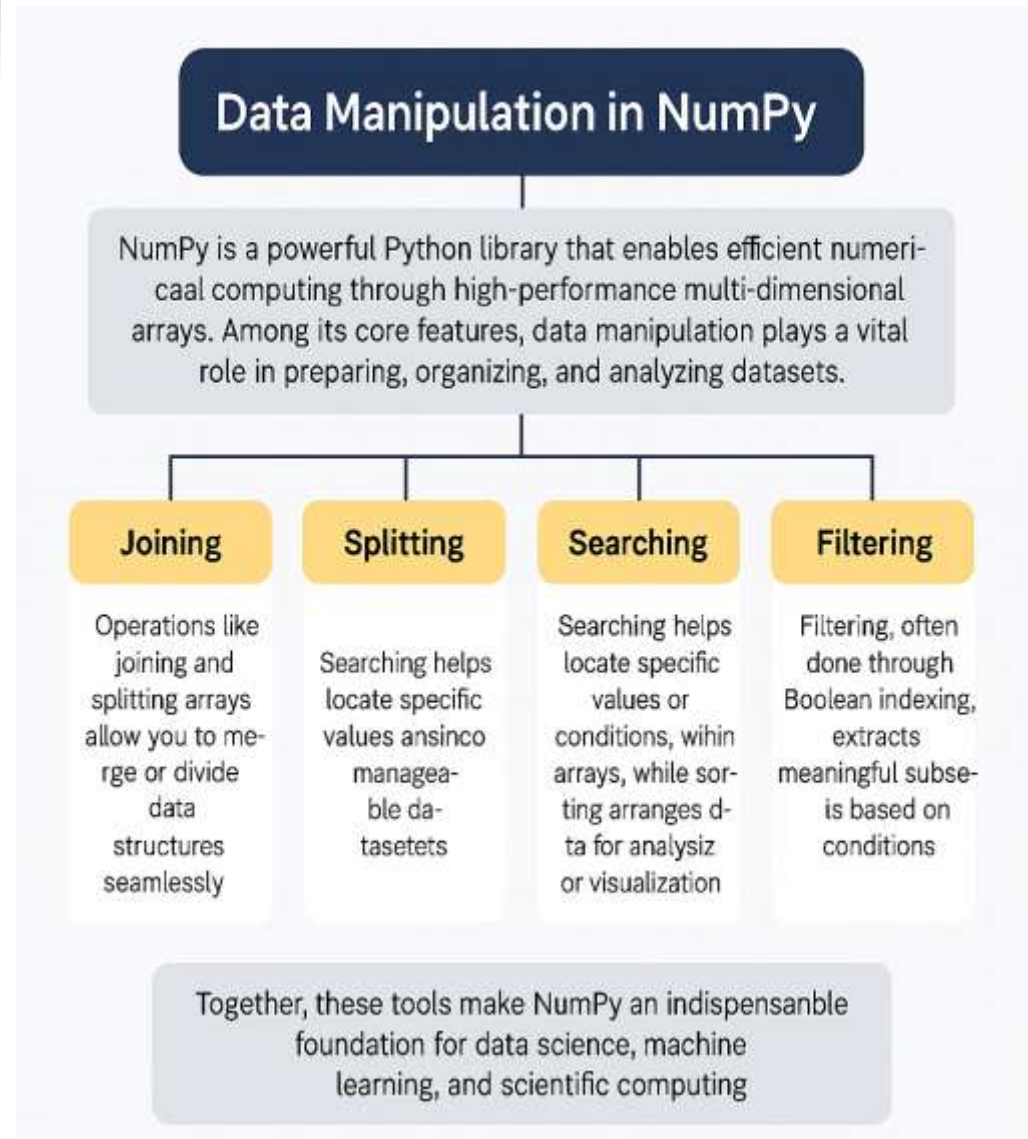
NumPy is a powerful Python library that enables efficient numerical computing through high-performance multi-dimensional arrays. Among its core features, data manipulation plays a vital role in preparing, organizing, and analyzing datasets. Operations like joining and splitting arrays allow you to merge or divide data structures seamlessly. These are essential in tasks such as reshaping datasets or combining results from different sources.

In addition to structural operations, NumPy provides robust functions for searching, sorting, and filtering data. Searching helps locate specific values or conditions within arrays, while sorting arranges data for analysis or visualization. Filtering, often done through Boolean indexing, extracts meaningful subsets based on conditions. Together, these tools make NumPy an indispensable foundation for data science, machine learning, and scientific computing.

3.2 OBJECTIVE

After completing this Unit we will be able to:

- Apply reshaping, resizing, and flattening of arrays.
- Manipulate array elements using insertion, deletion, and concatenation.
- Analyze data using sorting and filtering techniques.
- Evaluate efficient data handling methods using NumPy functions.

NOTE*Figure 3.1: NumPy Data Manipulation***3.3 DATA MANIPULATION IN NUMPY**

Data Manipulation in NumPy refers to the process of transforming, organizing, and extracting useful information from numerical data using NumPy's powerful array-handling functions. NumPy provides a suite of efficient tools to reshape, combine, filter, and analyze multi-dimensional arrays, which are the foundation of most scientific and data science computations.

NumPy is a foundational Python library designed for efficient numerical computing. At its core, it supports powerful N-dimensional array objects, which are optimized for performance and memory usage. One of the most important features of NumPy is its ability to manipulate data seamlessly. Whether you're cleaning datasets, transforming them for analysis, or preparing them for machine learning models, NumPy provides the tools to reshape, merge, slice, and operate on arrays with ease.

NOTE

The data manipulation capabilities of NumPy include array joining and splitting, which allow users to combine multiple arrays into a single structure or divide large arrays into manageable sections.

This is particularly useful in handling tabular or matrix-based data. NumPy also supports searching operations using functions like `np.where()` and `np.searchsorted()` that help locate values based on conditions or indices. These are essential when filtering records, identifying outliers, or selecting values that meet specific criteria.

Beyond structure, NumPy enables value-based manipulation through sorting and filtering. You can sort arrays using `np.sort()` or get the indices of sorted data using `np.argsort()`—helpful for ranking or ordering data. Boolean indexing lets you filter elements based on conditions, such as extracting all values above a threshold. These tools, when combined, form a powerful data manipulation pipeline that supports everything from exploratory data analysis to machine learning preprocessing. NumPy's efficiency and simplicity make it a key component in any data-driven Python application.

Key Aspects of Data Manipulation in NumPy:

1. Joining Arrays

NumPy allows you to combine multiple arrays using functions like `np.concatenate()`, `np.hstack()`, and `np.vstack()` for horizontal or vertical stacking of data.

```
import numpy as np

# Creating sample arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

# 1. Concatenate along axis 0 (vertical stacking)
vertical_join = np.concatenate((arr1, arr2), axis=0)
print("Vertical Join:\n", vertical_join)

# 2. Concatenate along axis 1 (horizontal stacking)
horizontal_join = np.concatenate((arr1, arr2), axis=1)
print("\nHorizontal Join:\n", horizontal_join)

# 3. Using hstack() – horizontal stack
hstacked = np.hstack((arr1, arr2))
print("\nHorizontal Stack using hstack():\n", hstacked)

# 4. Using vstack() – vertical stack
vstacked = np.vstack((arr1, arr2))
print("\nVertical Stack using vstack():\n", vstacked)
```

NOTE**Output:**

Vertical Join:

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

Horizontal Join:

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

Horizontal Stack using hstack():

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

Vertical Stack using vstack():

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

2. Splitting Arrays

Large arrays can be divided into smaller parts using `np.split()`, `np.array_split()`, or their horizontal/vertical variants to manage subsets of data.

```
import numpy as np

# Creating a sample array
arr = np.array([[10, 20, 30, 40],
                [50, 60, 70, 80]])

# 1. Split into 2 arrays along columns (axis=1)
split_columns = np.hsplit(arr, 2)
print("Horizontal Split (Columns):")
for part in split_columns:
    print(part)

# 2. Split into 2 arrays along rows (axis=0)
split_rows = np.vsplit(arr, 2)
```

NOTE

```

print("\nVertical Split (Rows):")
for part in split_rows:
    print(part)
# 3. General split using np.split (splits into equal parts)
arr_1D = np.array([1, 2, 3, 4, 5, 6])
equal_split = np.split(arr_1D, 3)
print("\nEqual Split of 1D Array:")
for part in equal_split:
    print(part)
# 4. Using np.array_split for unequal split
unequal_split = np.array_split(arr_1D, 4)
print("\nUnequal Split of 1D Array:")
for part in unequal_split:
    print(part)

```

Output:

```

Horizontal Split (Columns):
[[10 20]
 [50 60]]
[[30 40]
 [70 80]]
Vertical Split (Rows):
[[10 20 30 40]]
[[50 60 70 80]]
Equal Split of 1D Array:
[1 2]
[3 4]
[5 6]
Unequal Split of 1D Array:
[1 2]
[3 4]
[5]
[6]

```

Summary:

NOTE

- `np.hsplit()` splits an array horizontally (by columns).
- `np.vsplit()` splits an array vertically (by rows).
- `np.split()` requires the array to be evenly divisible.
- `np.array_split()` works even when the array isn't divisible evenly.

3. Searching Within Arrays

Functions like `np.where()` and `np.searchsorted()` help locate specific values or positions based on conditions, enabling targeted data access.

```
import numpy as np

# Sample array
arr = np.array([10, 20, 30, 40, 50])

# 1. Find indices where condition is true
indices = np.where(arr > 25)
print("Indices where arr > 25:", indices)

# 2. Find elements equal to 30
equal_to_30 = np.where(arr == 30)
print("Index where arr == 30:", equal_to_30)

# 3. Using searchsorted to find insertion point
sorted_arr = np.array([10, 20, 30, 40, 50])
insert_pos = np.searchsorted(sorted_arr, 35)
print("Insertion position for 35 in sorted array:", insert_pos)
```

Output:

```
Indices where arr > 25: (array([2, 3, 4]),)
Index where arr == 30: (array([2]),)
Insertion position for 35 in sorted array: 3
```

4. Sorting Data

You can sort arrays in ascending or descending order using `np.sort()` or obtain indices of sorted elements using `np.argsort()` for advanced data arrangements.

```
import numpy as np

# Sample 1D array
arr1D = np.array([40, 10, 50, 20, 30])

# 1. Sort the array
sorted_arr = np.sort(arr1D)

print("Sorted 1D array:", sorted_arr)
```

NOTE

```
# 2. Get the indices that would sort the array
sorting_indices = np.argsort(arr1D)
print("Indices for sorting:", sorting_indices)

# Sample 2D array
arr2D = np.array([[30, 10],
                  [20, 40]])

# 3. Sort each row of 2D array
sorted_rows = np.sort(arr2D, axis=1)
print("Row-wise sorted 2D array:\n", sorted_rows)

# 4. Sort each column of 2D array
sorted_columns = np.sort(arr2D, axis=0)
print("Column-wise sorted 2D array:\n", sorted_columns)
```

Output

```
Sorted 1D array: [10 20 30 40 50]
Indices for sorting: [1 3 4 0 2]
Row-wise sorted 2D array:
[[10 30]
 [20 40]]
Column-wise sorted 2D array:
[[20 10]
 [30 40]]
```

5. Filtering Data

Boolean indexing enables you to filter arrays by applying logical conditions (e.g., `array[array > 10]`), allowing the selection of meaningful subsets.

```
import numpy as np

# Sample array
arr = np.array([10, 25, 30, 45, 50, 60])

# 1. Filter elements greater than 30
filtered = arr[arr > 30]
print("Elements greater than 30:", filtered)

# 2. Filter even numbers
even_numbers = arr[arr % 2 == 0]
```

NOTE

```
print("Even numbers:", even_numbers)

# 3. Filter elements between 20 and 50
range_filter = arr[(arr >= 20) & (arr <= 50)]
print("Elements between 20 and 50:", range_filter)
```

Output:

```
Elements greater than 30: [45 50 60]
Even numbers: [10 30 50 60]
Elements between 20 and 50: [25 30 45 50]
```

These operations make NumPy indispensable for data preprocessing, cleaning, analysis, and feeding structured data into machine learning models or statistical algorithms.

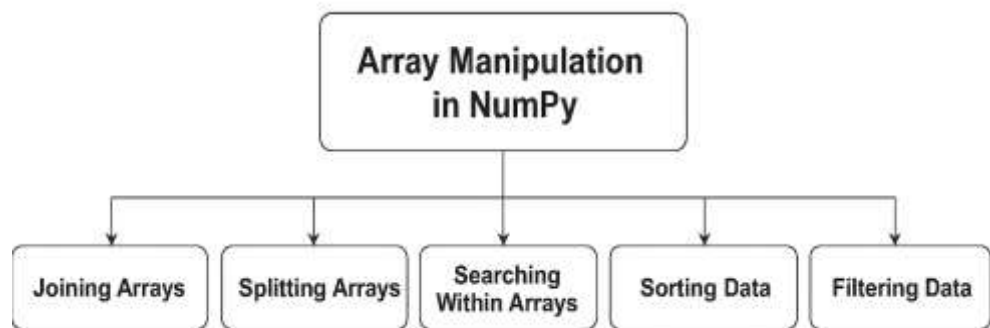


Figure 3.2: Numpy Array Manipulation

3.4 ARRAY JOINING TECHNIQUES IN NUMPY

In NumPy, array joining refers to combining two or more arrays into a single unified array. This operation is essential when consolidating data from multiple sources or merging intermediate results.

The `np.concatenate()` function is the most versatile, allowing arrays to be joined along a specified axis. To use it effectively, the arrays must have matching shapes along the axis not being joined.

Horizontal stacking is performed using `np.hstack()`, which appends arrays side-by-side (along `axis=1`). This method is particularly useful when dealing with column-based data or when adding features to datasets. On the other hand, `np.vstack()` stacks arrays vertically (along `axis=0`), effectively appending rows. These stacking functions internally utilize `concatenate` with the `axis` parameter adjusted for layout.

For more advanced operations, `np.stack()` allows you to join arrays along a new dimension, which is helpful for generating higher-dimensional arrays from 1D or 2D inputs. It differs from `concatenate` as it creates an additional axis, which must be accounted for when indexing.

These techniques are foundational in data science workflows for preprocessing, feature combination, or time-series structuring.

NOTE

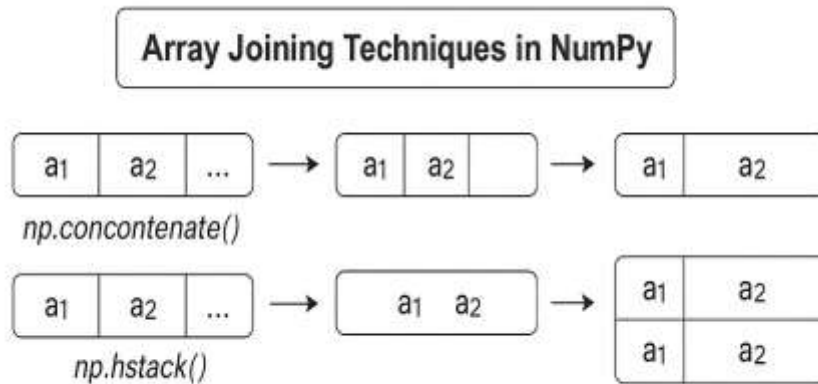


Figure 3.3: Numpy Array Joining Techniques

3.5 SPLITTING ARRAYS: BREAKING DATA INTO PARTS

In NumPy, splitting arrays is an essential operation for dividing large datasets into smaller, manageable chunks. This is useful for tasks such as cross-validation, batch processing, or analyzing data segments independently. The most basic function for this purpose is `np.split()`, which divides an array into equal-sized parts along a specified axis. The `axis` parameter determines whether you're splitting rows, columns, or another dimension.

For more flexible splits, NumPy provides `np.array_split()`, which can divide arrays into unequal parts if the number of elements doesn't divide evenly. This makes it highly useful when working with datasets of arbitrary size. Unlike `np.split()`, it doesn't raise an error if the division isn't perfect. Instead, it handles the remainder by distributing it across the output arrays.

Horizontal and vertical variants of `split`, namely `np.hsplit()` and `np.vsplit()`, offer simplified syntax when working with 2D arrays. These are especially handy in image processing or when separating features and labels in tabular datasets. They internally call `np.split()` with `axis` values set appropriately, providing readability and convenience.

These functions empower users to modularize data workflows and operate on segments independently, enhancing clarity and computational efficiency. Whether in preprocessing, analytics, or modeling, array splitting helps keep code modular and scalable.

```
import numpy as np

# Create an array
arr = np.arange(12)
```

NOTE

```
# Split into 3 equal parts
split_arr = np.split(arr, 3)

print("Original Array:", arr)
print("Split Arrays:", split_arr)
```

This splits the array [0, 1, ..., 11] into 3 sub-arrays of 4 elements each.

3.6 EFFICIENT SEARCHING IN NUMPY ARRAYS

Searching is a core operation in data analysis, and NumPy provides highly efficient tools to locate values within arrays. The `np.where()` function is one of the most commonly used, allowing users to find elements based on conditions. It returns indices where the condition holds true, making it ideal for filtering or data validation. This approach ensures that only relevant data is extracted for further processing.

Another powerful function is `np.searchsorted()`, used to find the index where a new value should be inserted to maintain the sorted order. This is particularly useful in time series data or any sorted datasets where fast binary search performance is needed. It returns the insertion point, not the value, helping in ranking and positioning operations.

For more complex lookups, `np.nonzero()` can be used to return indices of non-zero elements, often paired with condition-based filtering. Additionally, `np.argmax()` and `np.argmin()` help identify the positions of maximum or minimum values in arrays. These functions help analyze trends, detect peaks, and evaluate dataset extremes.

NumPy's searching capabilities are optimized for speed and scalability, even when handling large datasets. With logical expressions and vectorized operations, users can write cleaner, faster, and more efficient code. These tools are crucial in data preprocessing, condition checking, and real-time analytics.

```
import numpy as np

arr = np.array([5, 12, 7, 20, 25])

# Use np.where to find values greater than 10
indices = np.where(arr > 10)

# Use np.searchsorted to find insert position for 15
position = np.searchsorted(arr, 15)

print("Indices of elements > 10:", indices)
```

```
print("Insert position for 15:", position)
```

NOTE

This code shows how to locate specific values and identify where a new value fits in a sorted array, ensuring efficient data management.

3.7 SORTING ARRAYS: ORGANIZING DATA EFFICIENTLY

Sorting is a fundamental operation when preparing data for analysis, and NumPy offers efficient tools to organize arrays in ascending or descending order. The `np.sort()` function returns a sorted copy of the array without altering the original data. This is especially useful for preserving the initial dataset while still analyzing ordered values for trends or thresholds.

If you need to determine the positions that would sort an array, `np.argsort()` comes into play. It returns the indices that would sort the array, allowing for indirect sorting or rearranging associated data based on those positions. This is crucial in applications like ranking systems or matching sorted keys with their original values.

For multi-dimensional arrays, NumPy allows sorting along a specific axis. By specifying the axis parameter, users can sort rows or columns independently, making it ideal for table-like datasets. These flexible, vectorized functions make sorting operations fast and suitable for large-scale numerical computing.

```
import numpy as np

arr = np.array([15, 3, 9, 1, 20])

# Sort array in ascending order
sorted_arr = np.sort(arr)

# Get indices that would sort the array
sorted_indices = np.argsort(arr)

print("Sorted Array:", sorted_arr)
print("Sorting Indices:", sorted_indices)
```

This example shows how to organize data efficiently while preserving or analyzing original order information using NumPy's sorting tools.

3.8 FILTERING DATA WITH BOOLEAN INDEXING

Boolean indexing is a powerful feature in NumPy that allows you to extract elements from an array that meet a specified condition. Instead of looping through elements, NumPy uses logical expressions to generate Boolean arrays,

NOTE

where True represents values to keep. This approach is efficient and readable, making it widely used in data science workflows.

One of the key advantages of Boolean indexing is its ability to apply complex conditions in a single step. For example, you can filter values greater than a certain threshold and less than another simultaneously using logical operators. These expressions are concise and run faster than traditional Python loops.

Boolean arrays can also be used to assign new values to elements that meet a condition. This is helpful in data cleaning and transformation tasks, such as replacing outliers or missing values. By combining filtering and assignment, you can perform bulk operations on arrays efficiently.

Filtering can also be combined with multiple conditions using & (and), | (or), and ~ (not), offering great flexibility. These techniques allow users to extract meaningful subsets of data for analysis, modeling, or visualization—all without modifying the original dataset structure.

```
import numpy as np

arr = np.array([10, 25, 5, 30, 15])

# Create a Boolean mask for values greater than 15
mask = arr > 15

# Use the mask to filter values
filtered_arr = arr[mask]

print("Filtered Array (values > 15):", filtered_arr)
```

This code demonstrates how Boolean indexing selects elements that meet a condition—fast and without loops.

**Activity 3.1**

Create an array of numbers from 1 to 20.

Reshape it into a 4×5 array

Slice the second and third rows

.....

.....

.....

☑ Check Your Progress 3.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

NOTE

Q1. What makes NumPy indispensable is its ability to reshape, merge, and split data effortlessly—turning complex operations into clear, logical steps that drive modern data science.

Q2. How you organize, search, and filter data determines the story your analysis tells—NumPy gives you the tools to write that story with accuracy and speed.

3.9 SUMMARY

Data manipulation in NumPy is a core competency in scientific computing and data analysis. NumPy's array-based structure allows for efficient handling of large datasets, with operations that are both concise and high-performing. Joining arrays lets you merge multiple datasets into a single structure using functions like `np.concatenate()`, `np.hstack()`, and `np.vstack()`, making it easy to combine data from various sources. On the other hand, splitting arrays with tools like `np.split()` or `np.array_split()` enables you to divide large datasets into smaller, manageable parts for focused analysis. These techniques are essential when dealing with batch processing, cross-validation, or separating training and test datasets in machine learning workflows.

Searching, sorting, and filtering enhance the analytical capability of NumPy by allowing you to explore and refine datasets. Searching functions like `np.where()` and `np.searchsorted()` allow for quick identification of element positions or matching conditions. Sorting, through `np.sort()` and `np.argsort()`, is crucial for organizing data for visualization, reporting, or preparation for further processing. Filtering, often done via Boolean indexing, enables selective extraction of data based on logical conditions, ensuring you work only with relevant information. Together, these techniques provide a powerful framework for structuring, transforming, and analyzing numerical data, making NumPy a foundational tool in the data science ecosystem.

3.10 GLOSSARY

- **Concatenation** – The process of joining two or more arrays along a specified axis.
- **Splitting** – Dividing a single array into multiple smaller arrays.
- **Sorting** – Arranging array elements in ascending or descending order.
- **Filtering** – Selecting array elements based on specified logical conditions.
- **Boolean Masking** – A technique for filtering array elements using Boolean values.

NOTE

- **Aggregation** – The process of computing summary statistics such as sum, mean, or maximum.
- **Stacking** – Combining multiple arrays vertically or horizontally into a larger array.
- **Appending** – Adding new elements or arrays to an existing array.
- **Deletion** – Removing selected elements, rows, or columns from an array.
- **Mathematical Functions** – Built-in NumPy functions used to perform arithmetic, statistical, and trigonometric operations on arrays.

3.11 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 3.1**Answer 1:**

- **Array Manipulation Made Easy:** NumPy provides functions like `reshape()`, `concatenate()`, and `split()` to handle data efficiently and intuitively.
- **Axis-Based Operations:** It allows operations along specific axes, enabling complex transformations with simple syntax.
- **High Performance:** Operations are performed at high speed using optimized C-based backends.
- **Memory Efficiency:** Views and broadcasting minimize data copying, making operations lightweight.
- **Foundation for Data Science:** These features enable consistent, reproducible workflows essential for real-world data applications.

Answer 2:

In data analysis, the way you organize and access data fundamentally shapes your insights and outcomes. NumPy, a powerful numerical computing library in Python, provides the core tools for:

1. Organizing Data:

- NumPy's `ndarray` (N-dimensional array) allows structured storage of large datasets.
- Efficient indexing, slicing, reshaping, and broadcasting enable you to rearrange data without copying it unnecessarily.

2. Searching Data:

- Functions like `np.where()`, `np.nonzero()`, and boolean indexing help **locate elements** based on conditions.
- Logical operations (e.g., `a > 5`) return masks that make conditional filtering seamless and highly efficient.

3. Filtering Data:

- With boolean arrays and conditions, you can **filter datasets dynamically** (e.g., selecting all values greater than a threshold or excluding NaNs).
- Combined with logical operators (`&`, `|`, `~`), you can form complex filters to isolate meaningful patterns.

NOTE**4. Impact on Data Storytelling:**

- Clean and logically structured data leads to clearer, more accurate visualizations and interpretations.
- NumPy's performance-optimized operations allow real-time data transformations, even on large datasets, enabling agile exploration and hypothesis testing.
- Ultimately, the clarity, speed, and accuracy of NumPy functions ensure your analysis reflects the true narrative hidden in the data — free of noise, redundancy, or bias.

3.12 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/Use r/UnitUrls?unitId=1264	

3.13 REFERENCE

- **NumPy Developers.** (2024). *NumPy Documentation*.
- **VanderPlas, J.** (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media.

3.14 EXERCISE**Part 1****A. Multiple choice question:**

- Which NumPy function is used to join two arrays along an existing axis?**
 - np.split()
 - np.concatenate()
 - np.where()
 - np.argsort()
- What does the np.array_split() function allow that np.split() does not?**
 - Splitting arrays vertically
 - Joining arrays

NOTE

- C) Uneven splitting of arrays
- D) Sorting arrays

3. Which function helps in locating the positions of elements that meet a condition?

- A) np.argsort()
- B) np.where()
- C) np.sort()
- D) np.vstack()

4. To filter elements in an array greater than 10, which of the following is correct?

- A) array < 10
- B) array = 10
- C) array[array > 10]
- D) np.filter(array, 10)

5. What does np.argsort() return when used on a NumPy array?

- A) Sorted array
- B) Indexes that would sort the array
- C) Maximum value
- D) Boolean mask

Answer with Explanations:

1. B) np.concatenate()

Explanation: Concatenate() joins arrays along an existing axis.

2. C) Uneven splitting of arrays

Explanation: array_split() can split an array into unequal parts, unlike split() which requires equal sections.

3. B) np.where()

Explanation: np.where() returns indices of elements that satisfy a given condition.

4. C) array[array > 10]

Explanation: This applies a boolean condition to filter values greater than 10.

5. B) Indexes that would sort the array

Explanation: argsort() returns positions that would arrange the array in sorted order.

B. Short answer type question

1. What is the purpose of joining arrays in NumPy, and which functions are used for this task?

2. How does Boolean indexing help in filtering data in NumPy arrays?
3. What is the difference between `np.sort()` and `np.argsort()` in sorting arrays?

NOTE**Part 2****C. Long answer type question (200 words)**

1. What are the core techniques for joining and splitting arrays in NumPy, and why are they important in data preprocessing?
2. How does NumPy enable efficient searching, sorting, and filtering, and what advantages do these functions offer in data analysis?
3. Why is data manipulation a foundational step in NumPy-based workflows, and how does it support advanced data science applications?

Part 3**D. Long Answer Type Questions (300 words)**

1. Explain reshaping, resizing, and transposing arrays in NumPy. Discuss when and why these operations are used in data analysis.
2. Describe how NumPy handles concatenation, splitting, stacking, and tile operations. Provide examples showing practical applications.
3. Discuss common data manipulation techniques such as sorting, filtering, condition-based selection, and replacing values in arrays.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | UCI | Member of: APU | Joint Committee: UGC | UER | AICTE | Recognized by: UGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

Block II	NUMPY RANDOM	51-88
Unit 4	Introduction to Data Distribution	53
Unit 5	Exploring Probability Distributions	65
Unit 6	Advanced Probability Distributions	77



This block focuses on the NumPy Random module, which is widely used in statistics, machine learning, simulations, and data science. Students will learn how data distributions work and how random numbers are generated in NumPy. The block covers essential probability distributions such as uniform, normal, binomial, and more advanced ones used for modeling and statistical analysis. By understanding how random sampling and probability distributions operate, students will be able to create simulations, analyze statistical patterns, and develop data-driven applications with greater confidence.

UNIT 4

INTRODUCTION TO DATA DISTRIBUTION

NOTE

UNIT STRUCTURE

- 4.1 Introduction
- 4.2 Objective
- 4.3 Introduction to Data Distribution.
- 4.4 Random Permutation
- 4.5 Seaborn Basics, Seaborn Histograms
- 4.6 Permutation Visualization
- 4.7 Summary
- 4.8 Glossary
- 4.9 Answers to Check Your Progress
- 4.10 SUGGESTED READINGS – OER-OER
- 4.11 Reference
- 4.12 Exercise

4.1 INTRODUCTION

Understanding data distribution is essential for analyzing patterns and making statistical inferences. Data can be spread in various ways—normally, uniformly, or skewed—and recognizing this helps in selecting the right models. Random permutation, often used in simulations and testing, rearranges data to study variability without changing its content. Together, they form the base for robust, repeatable data analysis.

Seaborn is a powerful visualization library in Python that simplifies complex data insights through beautiful, informative plots. It works seamlessly with pandas and NumPy, allowing easy plotting of distributions, categories, and relationships. From histograms to KDE plots, Seaborn helps explore data structure visually. Combining data distribution, permutation, and visualization gives analysts a complete toolkit for exploratory data analysis (EDA).

Table 4.1: Key Concepts Overview

Concept	Description	Example
Data Distribution	Patterns of data spread (e.g., normal, uniform)	np.random.normal()
Random Permutation	Shuffling data to test robustness or randomness	np.random.permutation()

NOTE

Seaborn Visualization	Visual representation of data distributions	sns.histplot(), sns.kdeplot()
Probability Density	Function showing likelihood of data values in range	sns.kdeplot()
Categorical Visualization	Comparison of categories using barplots or boxplots	sns.barplot(), sns.boxplot()

4.2 OBJECTIVE

After completing this Unit we will be able to:

- Define random numbers and data distribution concepts.
- Explain uniform and non-uniform distributions.
- Generate random data using NumPy random functions.
- Interpret the role of randomness in data analysis.

4.3 INTRODUCTION TO DATA DISTRIBUTION

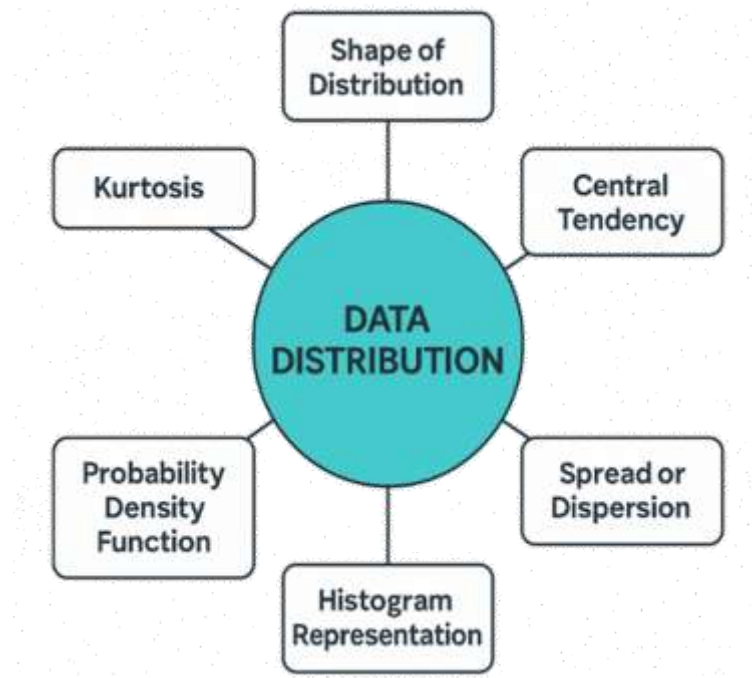


Figure 4.1: Data Distribution

“Data distribution refers to the way data values are spread or arranged across a dataset. Understanding the distribution helps identify patterns such as central tendency, variability, and skewness. Common types include normal, uniform, and binomial distributions, each with unique characteristics. Analyzing data distribution is crucial for selecting the right statistical methods and models.”

NOTE**Key Features of Data Distribution:**

1. **Shape of Distribution:** Describes the pattern of data spread — common shapes include normal (bell-shaped), skewed (left or right), and uniform (evenly spread).
2. **Central Tendency:** Measures like mean, median, and mode represent the center or typical value in a dataset, helping to summarize distribution.
3. **Spread or Dispersion:** Variability in data is indicated by range, variance, or standard deviation, showing how far values deviate from the center.
4. **Skewness:** Indicates the asymmetry in data distribution — positive skew has a longer right tail, and negative skew has a longer left tail.
5. **Kurtosis:** Measures the "tailedness" or peak of the distribution, helping to understand extreme values and outliers in the data.
6. **Probability Density Function (PDF):** In continuous distributions, PDF defines the likelihood of a value occurring in a particular range, especially in normal distribution.
7. **Histogram Representation:** Visual tool used to represent distribution of data, allowing quick insight into frequency, spread, and shape of the dataset.

4.4 RANDOM PERMUTATION

Random permutation is the process of rearranging the elements of an array or dataset in a random order without changing the actual data values. In NumPy, this is typically achieved using `np.random.permutation()`, which helps in creating randomized sequences. It is commonly used in tasks like shuffling data before training machine learning models to avoid bias. This technique ensures that data is treated fairly and uniformly during sampling and experimentation.

Random Permutation in NumPy, each illustrated with a code example:

1. Shuffling a 1D Array

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
shuffled = np.random.permutation(arr)
print("Shuffled 1D array:", shuffled)
```

Randomly rearranges elements in a 1D array without modifying the original.

2. Shuffling Rows in a 2D Array

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

NOTE

```
shuffled_rows = np.random.permutation(matrix)
print("Shuffled rows:\n", shuffled_rows)
```

Shuffles only the rows of a 2D array while keeping row data intact.

3. Preserving the Original Array

```
original = np.array([10, 20, 30])
shuffled = np.random.permutation(original)
print("Original:", original)
print("Shuffled:", shuffled)
```

`np.random.permutation()` returns a new array, keeping the original unchanged.

4. Creating a Permutation of Indexes

```
indexes = np.random.permutation(5)
print("Random indexes:", indexes)
```

Generates a random sequence of integers from 0 to n-1 — useful for shuffling data indirectly.

5. Reordering Arrays with Permuted Indexes

```
data = np.array(['a', 'b', 'c', 'd'])
indices = np.random.permutation(len(data))
print("Shuffled data:", data[indices])
```

Use permuted indexes to shuffle complex or linked datasets consistently.

4.5 SEABORN BASICS, SEABORN HISTOGRAMS

Seaborn is a Python visualization library built on top of Matplotlib, designed to make statistical plotting easier and more visually appealing. One of its core strengths is the ability to create informative graphics with minimal code. Histograms in Seaborn are used to display the distribution of a single variable, helping understand frequency and spread. The `sns.histplot()` function is commonly used for this purpose with built-in support for styling and bins control.

Five Key Points for Seaborn Basics & Histograms with Code Examples:

1. Importing Seaborn and Basic Histogram

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.histplot([1, 2, 2, 3, 4, 4, 5])
plt.show()
```

Simple histogram showing frequency distribution.

2. Using a Dataset with Seaborn

```
tips = sns.load_dataset("tips")
sns.histplot(data=tips, x="total_bill")
plt.show()
```

Uses built-in dataset to plot histograms of specific columns.

3. Adjusting Number of Bins

```
sns.histplot(data=tips, x="total_bill", bins=20)
plt.show()
```

Customizes how detailed the histogram is with *bins* parameter.

4. Adding Hue for Categories

```
sns.histplot(data=tips, x="total_bill", hue="sex", multiple="stack")
plt.show()
```

Color codes histogram based on categories like gender.

5. Styling the Histogram

```
sns.set(style="darkgrid")
sns.histplot(data=tips, x="tip", kde=True, color="purple")
plt.show()
```

4.6 PERMUTATION VISUALIZATION

Permutation visualization is a graphical method used to represent the rearrangement of elements within a sequence. In data science and algorithm analysis, permutations are crucial for understanding the outcomes of shuffling, sorting, or reordering data. Visualizing permutations helps to identify the structure, randomness, or patterns resulting from these operations. It allows analysts to inspect whether the reshuffling meets the criteria for randomness or controlled transformation.

A common way to visualize permutations is through scatter plots, where the original indices of an array are plotted against their permuted positions. If the permutation is random, the scatter plot will appear scattered without any discernible pattern.

NOTE

Conversely, structured permutations such as sorting will form a straight or curved line. These plots are valuable for testing algorithms like random number generators or for debugging reshuffling logic.

In Python, permutation visualizations can be efficiently created using libraries such as NumPy for permutation and Matplotlib or Seaborn for plotting. For example, using `np.random.permutation()` to shuffle an array and then plotting the result against the original order can visually demonstrate how the data has been altered. This kind of visual inspection is critical when working with randomized models or stochastic simulations.

Permutation visualization also plays a role in machine learning, especially in tasks like cross-validation where dataset shuffling affects training and testing splits. By visualizing permutations before modeling, one can ensure data integrity and fairness in sampling. Overall, permutation visualization not only supports technical accuracy but also enhances the interpretability of randomized data processes.



Activity 4.1

TITLE OF ACTIVITY: Simulating and Visualizing a Uniform Distribution Using NumPy and Matplotlib

OBJECTIVE: To gain practical experience in generating pseudo-random synthetic datasets and visually analyzing continuous data distributions using histograms.

MATERIAL NEEDED: A computer or smartphone equipped with a Python programming environment (such as Google Colab, Jupyter Notebook, or VS Code) with the `numpy` and `matplotlib` libraries installed, and a digital learning journal.

ACTIVITY: Use the NumPy random module (`numpy.random`) to generate an array of 100 floating-point values drawn from a uniform distribution between the range of 0 and 1. After successfully generating this dataset, utilize Matplotlib's plotting functions to construct a histogram that visualizes these values. Ensure you enhance your plot by adding a descriptive title, an X-axis label, and a Y-axis label to maintain professional documentation standards.

EXPECTATION: You should produce a clean, error-free Python script that effectively imports the required libraries, generates the correct multi-value array, and renders a properly labeled graphical histogram. Your final submission must display both the source code and the resulting visual plot side-by-side in your notebook.

REFLECTION: Examine the shape of the bars in your generated histogram. Since a uniform distribution theoretically means every single value has an

NOTE

equal mathematical probability of being selected, why do the bars in your 100-value plot not look perfectly flat or uniform? Reflect on how increasing your sample size from 100 values to 10,000 values would change the shape of the histogram, and what this teaches us about the Law of Large Numbers in data analytics.

☑ Check Your Progress 4.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. What shapes your insights is how you understand data distribution knowing whether data clusters, spreads, or skews is the first step toward accurate analysis.

.....

.....

.....

Q2. How you visualize randomness and structure in data determines what patterns emerge tools like random permutation and Seaborn turn raw numbers into readable stories.

.....

.....

.....

4.7 SUMMARY

Understanding how data is distributed is a foundational step in data analysis. It allows us to explore whether values are clustered, evenly spread, skewed, or follow specific patterns like normal distribution. Recognizing these characteristics helps in choosing the right statistical models and drawing meaningful conclusions. Data distribution gives insight into the central tendency, variability, and overall shape of the data, which directly influences analysis outcomes. Random permutation is a technique used to rearrange elements in a dataset randomly. This method is widely used in hypothesis testing and simulations, particularly to create sampling distributions or test the significance of patterns. It ensures that the randomness does not alter the statistical properties of the data, making it a powerful tool in controlled experiments and bootstrapping methods. Seaborn, a high-level Python visualization library based on Matplotlib, simplifies the process of making complex plots. It provides built-in themes and color palettes, making it easier to create attractive and informative visualizations. With functions like `sns.histplot()`, `sns.boxplot()`, and `sns.scatterplot()`, analysts can represent distributions, relationships, and variations in data with minimal code.

NOTE

4.8 GLOSSARY

- **Data Distribution** – The pattern in which data values are spread across a dataset.
- **Frequency Distribution** – A tabular or graphical representation showing how often each value or class occurs.
- **Histogram** – A graph that displays the frequency distribution of continuous data using adjacent bars.
- **Normal Distribution** – A symmetrical, bell-shaped probability distribution where most values cluster around the mean.
- **Skewness** – A measure of the asymmetry of a data distribution.
- **Kurtosis** – A measure that describes the peakedness or flatness of a data distribution.
- **Mean** – The arithmetic average of all values in a dataset.
- **Median** – The middle value in an ordered dataset.
- **Mode** – The value that occurs most frequently in a dataset.
- **Standard Deviation** – A statistical measure that indicates the dispersion of data values from the mean.

4.9 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 4.1

Answer 1:

Data distribution is the foundation upon which meaningful insights in data analysis are built. It refers to the way data points are spread across the range of values in a dataset. By examining this distribution, we uncover important characteristics such as central tendency (mean, median), variability (standard deviation, range), and the presence of patterns like symmetry or skewness.

When data **clusters** (i.e., many data points fall around a central value), it suggests consistency and may point toward normal behavior or trends. A **spread** indicates how varied or dispersed the data is; wide spreads might imply greater uncertainty or variability. **Skewness**, on the other hand, shows whether the data leans toward one side — left or right — and this influences which statistical methods are appropriate (e.g., mean may not be reliable in skewed distributions). Failing to understand these aspects may lead to incorrect assumptions, poor model performance, or misinterpreted results. For instance, applying a model that assumes normal distribution on heavily skewed data can result in significant errors. Therefore, visual tools like histograms, boxplots, and probability plots are essential to observe and interpret distribution before deeper analysis begins.

Ultimately, the **way you perceive distribution determines the quality of your analysis**, making it not just a technical requirement, but a critical thinking skill in data science, machine learning, and statistics.

Answer 2:

In data analysis, randomness and structure are two sides of the same coin. **Randomness** represents the uncertainty or variability in data, while **structure**

refers to consistent patterns, relationships, or trends that might exist within it. The ability to distinguish between the two is crucial for making sense of any dataset.

Random permutation is a technique used to shuffle data without changing its distribution, helping analysts test hypotheses, validate models, or check for biases. For example, when testing the significance of an observed effect, randomly rearranging data values allows us to simulate what outcomes might look like by chance alone. This helps us understand whether our findings are due to real patterns or random variation.

Seaborn, on the other hand, is a powerful Python visualization library that makes it easy to uncover structure in data. With just a few lines of code, it can generate histograms, scatter plots, and box plots that reveal trends, clusters, outliers, and correlations — patterns that are invisible in raw numerical data.

By using tools like random permutation to probe randomness and Seaborn to visualize structure, analysts can turn complex data into clear visual narratives. This storytelling with data is what drives informed decisions, accurate predictions, and real-world understanding. Without visualization, data remains a collection of numbers — with it, data becomes insight.

NOTE

4.10 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/Use/UnitUrls?unitId=1265	

4.11 REFERENCE

- McKinney, W. (2018). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (2nd ed.). O'Reilly Media.
- VanderPlas, J. (2016). Python Data Science Handbook: Essential Tools for Working with Data. O'Reilly Media.
- Downey, A. B. (2014). Think Stats: Exploratory Data Analysis in Python (2nd ed.). Green Tea Press.

4.11 EXERCISE

Part 1

A. Multiple choice question

1. What does a data distribution primarily help you understand?

- A. The syntax of a programming language

NOTE

- B. The logical flow of a function
 - C. The frequency and spread of data values
 - D. The structure of a for-loop
- 2. Which NumPy function is used to randomly permute elements in an array?**
- A. `np.random.shuffle()`
 - B. `np.random.permutation()`
 - C. `np.random.seed()`
 - D. `np.random.choice()`
- 3. What type of plot is typically used in Seaborn to show data distribution?**
- A. Line plot
 - B. Scatter plot
 - C. Histogram
 - D. Pie chart
- 4. In Seaborn, which function is used to create a histogram?**
- A. `sns.barplot()`
 - B. `sns.scatterplot()`
 - C. `sns.histplot()`
 - D. `sns.boxplot()`
- 5. What is the benefit of using random permutation in data analysis?**
- A. It helps remove order bias during experiments or training.
 - B. It sorts the data in ascending order.
 - C. It groups identical values together.
 - D. It deletes duplicate values.

Answer with Explanations:

- 1. C)** The frequency and spread of data values
Explanation: A data distribution shows how values are spread and how frequently they occur.
- 2. B)** `np.random.permutation()`
Explanation: `permutation()` returns a shuffled copy of the array.
- 3. C)** Histogram
Explanation: Histograms are commonly used to visualize data distribution.
- 4. C)** `sns.histplot()`
Explanation: `histplot()` is Seaborn's function for creating histograms.

NOTE

5. **A)** It helps remove order bias during experiments or training.

Explanation: Random permutation helps eliminate order bias during analysis or model training.

B. Short answer type question

1. What is data distribution, and why is it important in data analysis?
2. What does `np.random.permutation()` do in NumPy?
3. How does Seaborn's `histplot()` function help in visualizing data?

Part 2

C. Long answer type question (200 words)

1. What is data distribution, and how does it impact data analysis and visualization?
2. Explain the concept of random permutation in NumPy and its use cases in data science.
3. How does Seaborn enhance data visualization, particularly for distributions and randomness?

Part 3

D. Long Answer Type Questions (300 words)

1. Explain what data distribution means in statistics. Discuss the role of NumPy Random in generating synthetic datasets for analysis.
2. Describe uniform, normal, binomial, and Poisson distributions. Explain how NumPy generates each distribution and where they are used.
3. Discuss the relevance of random sampling in real-world applications. Explain various NumPy sampling functions with examples.

UNIT 5

EXPLORING PROBABILITY DISTRIBUTIONS

NOTE

UNIT STRUCTURE

- 5.1 Introductions
- 5.2 Objective
- 5.3 Understanding the Normal Distribution Curve
- 5.4 Modeling Binary Outcomes with Binomial Distribution
- 5.5 Poisson Distribution for Rare Events
- 5.6 Uniform Distribution: Equal Probability Explained
- 5.7 Logistic Distribution in Statistical Modeling
- 5.8 Multinomial Distribution for Multi-Category Trials
- 5.9 Summary
- 5.10 Glossary
- 5.11 Answers to check your progress
- 5.12 SUGGESTED READINGS – OER - OER
- 5.13 Reference
- 5.14 Exercise

5.1 INTRODUCTION

Understanding probability distributions is fundamental to statistics and data science. These distributions describe how values of a random variable are likely to occur. Each type—Normal, Binomial, Poisson, Uniform, Logistic, and Multinomial—models specific types of random events. They help in making predictions, identifying patterns, and analyzing uncertainty. Choosing the right distribution depends on the nature of the data and the problem at hand.

The Normal distribution, often called the bell curve, represents data that clusters around a mean. Binomial distribution models the probability of successes in a fixed number of trials with binary outcomes. Poisson distribution is used for modeling the number of times an event occurs within a given interval. Uniform distribution assumes all outcomes are equally likely across an interval. Logistic distribution looks like the normal but is used for classification and growth processes.

The Multinomial distribution extends the binomial to scenarios with more than two categories. These distributions form the backbone of many statistical tests, machine learning models, and real-world applications. Whether estimating probabilities, running simulations, or analyzing risk, understanding these tools is essential. By mastering them, analysts can better represent complex phenomena. In turn, this strengthens the accuracy and depth of data-driven decision-making.

NOTE

5.2 OBJECTIVE

After completing this Unit we will be able to:

- Explain common probability distributions (normal, binomial, etc.).
- Apply NumPy functions to simulate probability distributions.
- Analyze the characteristics of different distributions.
- Compare distributions based on their properties.

5.3 UNDERSTANDING THE NORMAL DISTRIBUTION CURVE

The Normal Distribution Curve, often referred to as the bell curve, is one of the most important concepts in statistics and data analysis. It represents how data tends to be distributed around a central value (mean) with a specific spread (standard deviation). The curve is symmetric, with most values clustering around the mean and fewer values appearing as you move further from the center.

Key characteristics of the normal distribution include:

- Mean = Median = Mode (centered perfectly).
- The curve follows the 68-95-99.7 rule:
 - About 68% of data falls within 1 standard deviation of the mean.
 - 95% within 2 standard deviations.
 - 99.7% within 3 standard deviations.
- It is used widely in natural sciences, finance, and machine learning because many real-world variables (like heights, test scores, or measurement errors) follow this pattern.

Understanding this curve helps you assess probability, identify outliers, and perform hypothesis testing. It also underpins many statistical tools like z-scores, confidence intervals, and normality tests.

5.4 MODELING BINARY OUTCOMES WITH BINOMIAL DISTRIBUTION

Modeling Binary Outcomes with Binomial Distribution

The Binomial Distribution is used to model binary (yes/no or success/failure) outcomes across a fixed number of independent trials. Each trial has two possible results: success with probability p , or failure with probability $1-p$.

Use Cases:

- Tossing a coin (heads/tails)
- Pass/fail exam outcomes
- Product defects (defective or not)

NOTE**Key Parameters:**

- n : number of trials
- p : probability of success in a single trial
- k : number of successes (random variable)

Python Code Example Using NumPy and SciPy

```
import numpy as np
from scipy.stats import binom
import matplotlib.pyplot as plt
# Parameters
n = 10      # number of trials
p = 0.5     # probability of success (e.g., heads)
k = np.arange(0, n+1) # possible number of successes

# PMF: Probability Mass Function
pmf = binom.pmf(k, n, p)
# Plotting
plt.bar(k, pmf, color='skyblue', edgecolor='black')
plt.title('Binomial Distribution: n=10, p=0.5')
plt.xlabel('Number of Successes')
plt.ylabel('Probability')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

Explanation:

- `binom.pmf(k, n, p)` computes the probability of getting exactly k successes out of n trials.
- This example models a fair coin tossed 10 times and visualizes the probabilities of getting 0 to 10 heads.

5.5 POISSON DISTRIBUTION FOR RARE EVENTS

The Poisson distribution is used to model the number of times an event occurs in a fixed interval of time or space, assuming the events occur independently and rarely. It is ideal when you're counting occurrences—like the number of customer arrivals per minute or system failures per month.

Example: Modeling Rare Events

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import poisson

# Average rate of event ( $\lambda$ )
lambda_rate = 4 # e.g., 4 customers arriving per hour
```

NOTE

```
# Possible number of occurrences
x = np.arange(0, 15)

# Calculate Poisson PMF
pmf = poisson.pmf(x, mu=lambda_rate)

# Plotting
plt.bar(x, pmf, color='lightcoral', edgecolor='black')
plt.title('Poisson Distribution ( $\lambda = 4$ )')
plt.xlabel('Number of Events')
plt.ylabel('Probability')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

5.6 UNIFORM DISTRIBUTION: EQUAL PROBABILITY EXPLAINED

The Uniform distribution is a type of probability distribution in which all outcomes are equally likely within a specific range. It's one of the simplest distributions and is useful when each value within the interval has the same chance of occurring.

Example: Visualizing Uniform Distribution

```
import numpy as np
import matplotlib.pyplot as plt

# Generate 1000 random numbers between 10 and 50
data = np.random.uniform(low=10, high=50, size=1000)

# Plot histogram
plt.hist(data, bins=20, color='skyblue', edgecolor='black', density=True)
plt.title('Uniform Distribution (10 to 50)')
plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

5.7 LOGISTIC DISTRIBUTION IN STATISTICAL MODELING

NOTE

The Logistic distribution is a continuous probability distribution used in statistical modeling, particularly for problems involving binary outcomes and logistic regression. It has a bell-shaped curve similar to the normal distribution but features heavier tails, making it more robust to outliers. Defined by a location (mean) and scale parameter, the logistic distribution is commonly used to model sigmoidal (S-shaped) growth, such as in population dynamics or probability predictions in classification tasks. Its mathematical properties and interpretability make it a key tool in machine learning, social sciences, and economics.

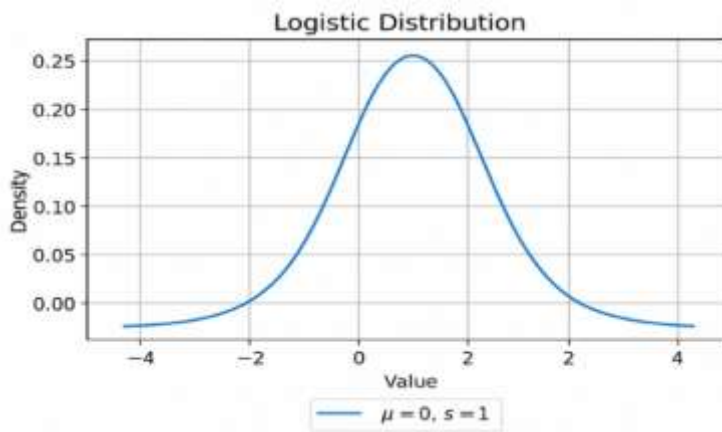


Figure 5.1: Logistic Distribution

5.8 MULTINOMIAL DISTRIBUTION FOR MULTI-CATEGORY TRIALS

The Multinomial distribution is a generalization of the Binomial distribution that models the outcomes of multi-category experiments. While the Binomial distribution deals with only two outcomes (like success or failure), the Multinomial handles more than two.

Each trial in a Multinomial experiment is independent, and each outcome falls into one of k categories, with the probability of each category remaining constant throughout. The sum of probabilities across all categories must equal 1. The result is a vector showing the count of occurrences in each category after n total trials.

In practical terms, this distribution is useful in classification tasks, natural language processing, and market research, where outcomes are split among several groups or labels. Tools like `numpy.random.multinomial()` allow simulation of such experiments easily in Python, making it ideal for statistical modeling and simulation-based learning.

Multinomial Distribution for Multi-Category Trials – Code Example:

```
import numpy as np
# Number of trials
```

NOTE

```
n = 10
# Probabilities for each category (must sum to 1)
pvals = [0.2, 0.5, 0.3]
# Simulate outcomes
samples = np.random.multinomial(n, pvals, size=5)
print("Multinomial Distribution Samples:")
print(samples)
```



Activity 5.1

Activity 5.1

TITLE OF ACTIVITY: Statistical Analysis of Synthetic Normal Distributions

OBJECTIVE: To build practical proficiency in generating continuous random datasets using a Gaussian (normal) distribution and applying statistical functions to verify central tendency and data spread.

MATERIAL NEEDED: A computer or smartphone with a Python programming environment installed (such as Google Colab, Jupyter Notebook, or VS Code), the numpy library, and a digital learning journal.

ACTIVITY: Use the NumPy random module to generate a synthetic dataset containing 1,000 values drawn from a standard normal distribution (where the theoretical mean is 0 and the standard deviation is 1). Write a concise Python script to calculate and print out the following four empirical properties of your generated sample:

1. The sample mean
2. The sample standard deviation
3. The absolute maximum value
4. The absolute minimum value

Ensure your code prints these metrics with clear, descriptive labels so that your output is easy to interpret.

EXPECTATION: You should produce a clean, executable, and error-free Python script utilizing appropriate NumPy operations (such as `np.mean()`, `np.std()`, `np.max()`, and `np.min()`). Your executed results must show a sample mean very close to 0 and a sample standard deviation very close to 1, demonstrating your ability to capture theoretical parameters through empirical coding.

REFLECTION: Look closely at your final printed sample mean and standard deviation. Why are they not *exactly* 0.0 and 1.0, even though those were the precise parameters you passed into the random generation function? Reflect on how finite sample sizes create slight deviations from theoretical

expectations and how increasing your dataset from 1,000 to 100,000 values would impact this precision.

NOTE**☑ Check Your Progress 5.1**

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. What defines the nature of your data is the distribution it follows knowing whether it's normal, binomial, or multinomial shapes how you model, test, and predict outcomes?

.....

Q2. What makes probability distributions powerful is their ability to capture real-world randomness from rare events to binary choices guiding accurate analysis and decision-making.

.....

5.9 SUMMARY

Exploring probability distributions provides a foundational understanding of how data behaves under different random conditions. The Normal distribution models continuous data with a symmetrical bell curve, widely used in natural and social sciences. Binomial distribution is essential for modeling binary outcomes like success/failure in repeated trials. Poisson distribution captures rare events over time or space, such as calls per minute or defects per unit. Uniform distribution assumes equal probability across a defined range, useful in simulations and basic probability models. Logistic distribution is often applied in classification tasks and statistical modeling due to its S-shaped curve. Multinomial distribution extends binomial logic to multi-category outcomes, such as predicting the result of a die roll. Each distribution serves as a tool to analyze, predict, and simulate real-world phenomena. Understanding these helps in selecting appropriate models and conducting meaningful statistical analyses. These distributions together empower data scientists and researchers to make accurate, informed decisions based on the nature of the data.

5.10 GLOSSARY

- **Probability Distribution** – A mathematical function that describes the likelihood of different outcomes of a random variable.
- **Random Variable** – A variable whose value depends on the outcome of a random experiment.

NOTE

- **Discrete Distribution** – A probability distribution for random variables that take distinct, countable values.
- **Continuous Distribution** – A probability distribution for random variables that can assume any value within a given range.
- **Probability Mass Function (PMF)** – A function that gives the probability of each possible value of a discrete random variable.
- **Probability Density Function (PDF)** – A function that describes the probability distribution of a continuous random variable.
- **Cumulative Distribution Function (CDF)** – A function that gives the probability that a random variable is less than or equal to a specified value.
- **Expected Value** – The long-run average value of a random variable based on its probability distribution.
- **Variance** – A measure of the spread of a probability distribution around its expected value.
- **Binomial Distribution** – A discrete probability distribution that models the number of successes in a fixed number of independent trials.

5.11 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 5.1

Answer 1:

What defines the nature of your data is the distribution it follows

This means that the underlying statistical distribution of a dataset—such as **Normal**, **Binomial**, or **Multinomial**—dictates the behavior and structure of that data. Understanding this distribution helps you decide which tools and models are suitable for your analysis.

Knowing whether it's normal, binomial, or multinomial

Each distribution has specific characteristics:

1. **Normal Distribution:** Symmetrical bell curve, common in natural measurements (e.g., height, test scores).
2. **Binomial Distribution:** Used for binary outcomes (success/failure), often in trials or experiments.
3. **Multinomial Distribution:** Extends binomial to outcomes with more than two categories.

Shapes how you model, test, and predict outcomes

Once the distribution is identified:

- You can **choose the right statistical tests** (e.g., t-tests, chi-square).

NOTE

- You can **build appropriate predictive models** (e.g., logistic regression for binomial data).
- You can **interpret results accurately**, avoiding invalid assumptions or errors.

Answer 2:

In real-world scenarios, outcomes are rarely predictable. Whether it's customer behavior, medical test results, or traffic flow, **probability distributions model this uncertainty**, giving structure to randomness.

Different types of distributions specialize in modeling various kinds of events:

- **Poisson Distribution** captures *rare events*, such as system failures or incoming calls at a help desk.
- **Binomial Distribution** handles *binary outcomes*, like pass/fail, yes/no, or success/failure scenarios.
- **Multinomial and Normal Distributions** extend this coverage to multi-category trials and continuous data, respectively.

"Guiding accurate analysis and decision-making"

By selecting the right distribution, analysts can:

- Perform valid statistical tests.
- Make more precise predictions.
- Reduce uncertainty in decision-making.

For example, using a Poisson model for hospital admissions helps in **resource planning**, while a binomial model supports **marketing success predictions**.

5.12 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/Use/UnitUrls?unitId=1266	

5.13 REFERENCE

- <https://numpy.org/doc/stable/reference/random/generated/numpy.random.normal.html>
- <https://scipy.github.io/devdocs/reference/stats.html>
- <https://www.statology.org/probability-distributions/>

NOTE

5.14 EXERCISE

Part 1

A. Multiple choice question:

1. What shape does the Normal distribution typically follow?

- A) Bell-shaped curve
- B) Rectangular shape
- C) Skewed right
- D) Step function

2. Which distribution is most appropriate for modeling binary outcomes, such as success or failure?

- A) Poisson Distribution
- B) Binomial Distribution
- C) Normal Distribution
- D) Multinomial Distribution

3. The Poisson distribution is best suited for modeling:

- A) Continuous variables
- B) Outcomes with equal probability
- C) Rare events over a fixed interval
- D) Probability of success in multiple trials

4. Which of the following distributions assumes each outcome has an equal probability?

- A) Normal Distribution
- B) Logistic Distribution
- C) Binomial Distribution
- D) Uniform Distribution

5. The Multinomial distribution is an extension of which other distribution?

- A) Normal
- B) Logistic
- C) Binomial
- D) Poisson

Answer with Explanations:

1. A) Bell-shaped curve

Explanation: The normal distribution forms a symmetric bell-shaped curve.

NOTE

2. **B) Binomial Distribution**

Explanation: Binomial distribution models outcomes with two possibilities like success/failure.

3. **C) Rare events over a fixed interval**

Explanation: Poisson distribution models rare, count-based events over a fixed interval

4. **D) Uniform Distribution**

Explanation: Uniform distribution assumes all outcomes have equal probability.

5. **C) Binomial**

Explanation: The multinomial distribution generalizes the binomial to more than two categories.

B. Short answer type question

1. What is the key characteristic of a Normal distribution, and where is it commonly applied?
2. How does the Binomial distribution differ from the Multinomial distribution in terms of outcomes?
3. When is it appropriate to use a Poisson distribution instead of a Binomial distribution?

Part 2

C. Long answer type question (200 words)

1. What is the significance of the Normal distribution in statistics, and how does it relate to real-world data analysis?
2. Compare and contrast Binomial, Poisson, and Multinomial distributions.
3. How are Uniform and Logistic distributions applied in statistical modeling, and what makes them useful in specific scenarios?

Part 3

D. Long Answer Type Questions (300 words)

1. Explain the concept of probability distributions. Compare continuous and discrete distributions with examples generated using NumPy.
2. Describe the normal (Gaussian) distribution in detail. Explain its characteristics, importance, and how it is used in data science and

NOTE

UNIT 6

ADVANCED PROBABILITY DISTRIBUTIONS

UNIT STRUCTURE

- 6.1 Introductions
- 6.2 Objective
- 6.3 Understanding Time-Based Events with Exponential Distribution
- 6.4 Modeling Statistical Deviations Using Chi-Square and Rayleigh Distributions
- 6.5 Heavy-Tailed Patterns in Pareto and Zipf Distributions
- 6.6 Summary
- 6.7 Glossary
- 6.8 Answers to check your progress
- 6.9 SUGGESTED READINGS – OER-OER
- 6.10 Reference
- 6.11 Exercise

6.1 INTRODUCTION

Advanced probability distributions offer deeper insight into complex data behaviors that go beyond common distributions like normal or binomial. The Exponential distribution is widely used to model the time between events in a Poisson process—ideal for scenarios like waiting times or service systems. Chi-Square distribution, on the other hand, plays a key role in hypothesis testing and statistical inference, especially for variance analysis and goodness-of-fit tests.

The Rayleigh distribution finds its place in signal processing and environmental modeling, particularly where magnitudes of vectors (like wind speed or noise amplitude) are involved. Pareto distribution captures the "80/20 rule" seen in economics and social sciences, where a small proportion accounts for the majority of the effect—making it ideal for modeling wealth distribution or file sizes. Lastly, Zipf's distribution is useful for ranking data such as word frequency in texts or city populations, where rank is inversely proportional to frequency. These distributions allow analysts to understand rare, extreme, or skewed patterns in data more effectively. By mastering them, one gains powerful tools for predictive modeling, decision-making under uncertainty, and real-world phenomena modeling in fields like physics, finance, linguistics, and beyond.

6.2 OBJECTIVE

After completing this Unit we will be able to:

- Describe advanced distributions such as Poisson and exponential.

- Apply advanced distribution functions in NumPy.
- Analyze real-world scenarios using probability distributions.
- Evaluate the suitability of distributions for different datasets.

NOTE

6.3 UNDERSTANDING TIME-BASED EVENTS WITH EXPONENTIAL DISTRIBUTION

The exponential distribution models the time between independent events occurring at a constant average rate, such as the time until a light bulb burns out.

1. **Key Parameter (λ or scale):** It is defined by a single parameter: the rate (λ) or its inverse (scale = $1/\lambda$), which controls how quickly events occur.
2. **Memoryless Property:** The exponential distribution is memoryless—past events do not affect the probability of future occurrences. This makes it ideal for modeling systems like queueing or failure rates.
3. **Applications:** Common in reliability engineering, queueing theory, and survival analysis—for example, modeling customer service wait times or component lifetimes.
4. **Visualization and Simulation:** Python libraries like NumPy and Matplotlib help generate and visualize exponential distributions for analysis.

Example:

```
import numpy as np
import matplotlib.pyplot as plt

# Generate exponential data (scale = 2 means  $\lambda = 0.5$ )
data = np.random.exponential(scale=2.0, size=1000)

# Plot histogram
plt.hist(data, bins=50, color='skyblue', edgecolor='black', density=True)
plt.title("Exponential Distribution (scale=2.0)")
plt.xlabel("Time")
plt.ylabel("Probability Density")
plt.grid(True)
plt.show()
```

6.4 MODELING STATISTICAL DEVIATIONS USING CHI-SQUARE AND RAYLEIGH DISTRIBUTIONS

1. **Chi-Square Distribution:** The Chi-Square distribution is used in hypothesis testing and confidence interval estimation for variance. It arises from the sum of the squares of independent standard normal

NOTE

variables. This makes it ideal for assessing statistical deviations and goodness-of-fit tests.

2. **Degrees of Freedom (df):** The shape of the Chi-Square distribution depends on its degrees of freedom (df). As df increases, the distribution becomes more symmetric and resembles the normal distribution. It's crucial in determining the test statistic in variance-based analysis.
3. **Rayleigh Distribution:** Rayleigh distribution models the magnitude of a vector with normally distributed components. It's often used in signal processing, especially for modeling noise and fading in wireless communications. It represents statistical deviations from the mean in two-dimensional spaces.
4. **Parameter Control:** Chi-Square's only parameter is degrees of freedom, while Rayleigh is defined by its scale (σ). Scale affects the spread or width of the Rayleigh distribution. Both are sensitive to changes in parameters, reflecting different real-world behaviors.
5. **Applications in Real World:** Chi-Square is widely used in bio-statistics, machine learning (feature independence), and experimental analysis. Rayleigh appears in radar systems, oceanography, and speech recognition. Their flexibility helps in modeling different types of statistical deviation.
6. **Data Visualization:** Python allows us to generate and compare these distributions easily. Histograms and density plots offer insight into their shapes. Below is an example comparing Chi-Square and Rayleigh distributions visually.

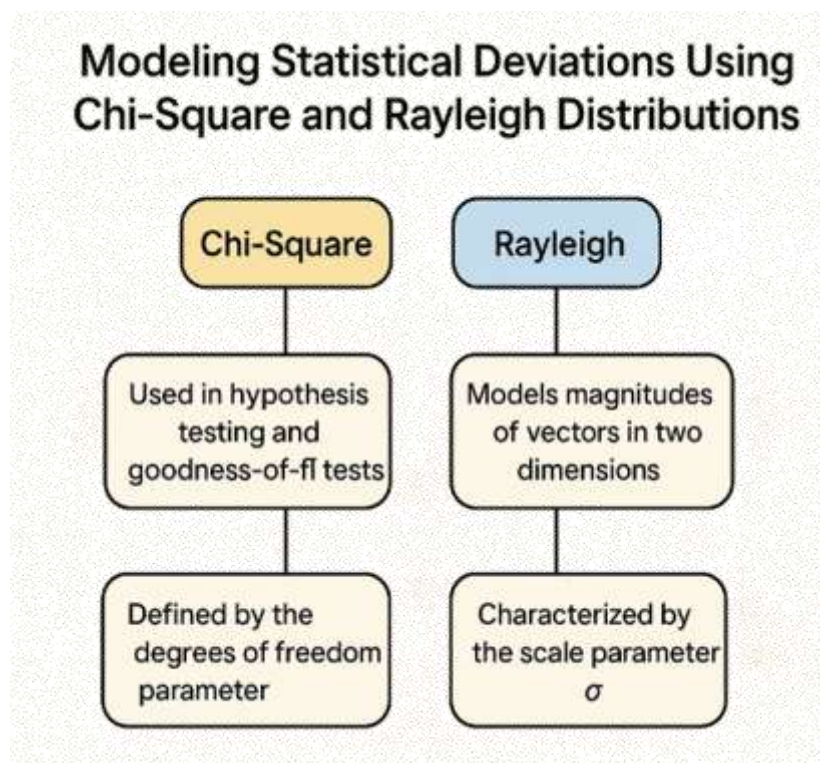


Figure 6.1: Statistical Deviation Modelling

Example:**NOTE**

```

import numpy as np
import matplotlib.pyplot as plt

# Generate data
chi_data = np.random.chisquare(df=4, size=1000)
rayleigh_data = np.random.rayleigh(scale=2.0, size=1000)

# Plot both distributions
plt.hist(chi_data, bins=40, alpha=0.6, label='Chi-Square (df=4)',
         color='salmon', density=True)
plt.hist(rayleigh_data, bins=40, alpha=0.6, label='Rayleigh (scale=2.0)',
         color='skyblue', density=True)

plt.title("Chi-Square vs Rayleigh Distribution")
plt.xlabel("Value")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.show()

```

6.5 HEAVY-TAILED PATTERNS IN PARETO AND ZIPF DISTRIBUTIONS

Heavy-tailed distributions like Pareto and Zipf highlight patterns where a small number of events account for most outcomes. These models are useful in economics, linguistics, and network theory, capturing power-law behaviors. Pareto distribution models wealth inequality, while Zipf's law is often used for word frequencies in natural language.

1. Pareto Distribution assumes that a few large values dominate—ideal for modeling wealth or resource distribution.
2. Zipf Distribution explains ranked data where frequency inversely correlates with rank—common in web traffic or language.
3. Both distributions emphasize *long-tail* behavior, where rare events still have significant influence.
4. These models challenge assumptions of symmetry and normality, highlighting real-world skewness.

NOTE

```
import numpy as np
import matplotlib.pyplot as plt

# Pareto distribution
pareto = (np.random.pareto(a=2.0, size=1000) + 1) * 1.5

# Zipf distribution
zipf = np.random.zipf(a=2.0, size=1000)

# Plot
plt.figure(figsize=(10,5))
plt.hist(pareto, bins=50, alpha=0.6, label='Pareto')
plt.hist(zipf, bins=50, alpha=0.6, label='Zipf')
plt.legend()
plt.title("Pareto vs Zipf Distribution")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```



Activity 6.1

Generate random samples using the following distributions:

Binomial distribution (n=10, p=0.5, size=50)

Poisson distribution ($\lambda = 3$, size=50)

Compare both distributions by printing their first 10 values.

.....

.....

.....

☒ Check Your Progress 6.1

Notes:

- Space is given below for your answer.
- Check your answer with the one given at the end of this unit.

Q1. What defines advanced probability distributions is their ability to model complexity whether it's time between events, extreme values, or rare outcomes, each distribution offers a lens to understand unique data behaviors.

.....

NOTE

Q2. What makes distributions like Exponential, Chi-Square, and Zipf powerful is how they explain the unpredictable from sudden failures to power-law phenomena giving analysts the tools to forecast rare yet impactful events?

6.6 SUMMARY

Advanced probability distributions provide powerful tools for modeling diverse real-world scenarios. The Exponential distribution is key to analyzing the time between events, making it useful in survival analysis and queuing systems. Chi-Square distribution aids in statistical hypothesis testing and assessing goodness-of-fit in datasets. Rayleigh distribution is valuable in engineering and physics, especially in modeling wave patterns and noise signals. Pareto distribution highlights wealth or resource inequality, capturing situations where a few items account for most outcomes. Zipf distribution, with its power-law behavior, describes phenomena like word usage and web traffic. Each distribution caters to specific data patterns and tail behaviors. They offer insights into rare events, asymmetry, or long-tail effects often missed by basic models. These tools are vital in fields ranging from economics and linguistics to reliability engineering. By understanding these advanced distributions, analysts can model, interpret, and predict complex probabilistic behaviors with greater precision.

6.7 GLOSSARY

- **Poisson Distribution** – A discrete probability distribution that models the number of events occurring within a fixed interval.
- **Exponential Distribution** – A continuous probability distribution used to model the time between independent events.
- **Uniform Distribution** – A probability distribution in which all outcomes within a specified range are equally likely.
- **Gaussian Distribution** – Another name for the normal distribution, characterized by its bell-shaped curve.
- **Sampling Distribution** – The probability distribution of a statistic obtained from repeated random samples.
- **Central Limit Theorem** – A statistical principle stating that the sampling distribution of the sample mean approaches normality as the sample size increases.

NOTE

- **Degrees of Freedom** – The number of independent values that are free to vary in a statistical calculation.
- **Z-Distribution** – The standard normal distribution with a mean of 0 and a standard deviation of 1.
- **T-Distribution** – A probability distribution used in hypothesis testing when the sample size is small and the population standard deviation is unknown.
- **Chi-Square Distribution** – A probability distribution commonly used in hypothesis testing and tests of independence.

6.8 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 6.1

Answer 1:

Advanced probability distributions are designed to handle situations where basic models like normal or binomial are not sufficient.

1. The **Exponential distribution** models time between independent events (e.g., system failures).
2. The **Chi-Square distribution** is used to test hypotheses and understand variance from expected outcomes.
3. The **Rayleigh distribution** applies to magnitude-related phenomena, such as signal noise.
4. The **Pareto distribution** captures extreme wealth distributions and power laws.
5. The **Zipf distribution** is common in natural language processing and web traffic, where a few items dominate.

Each distribution gives us tools to describe complex systems and uncover patterns that would be invisible with simpler models.

Answer 2 :

This statement reflects the importance of specialized distributions in real-world analytics:

1. **Exponential distribution** helps predict the timing of unexpected events (e.g., machine breakdowns, server crashes).
2. **Chi-Square distribution** is used to test statistical significance, especially in cases where normality cannot be assumed.
3. **Zipf distribution** models highly skewed data, like word frequency in language or views on viral videos, where a few elements dominate.

These distributions help analysts not only understand but also anticipate **rare and high-impact outcomes**, which are often the most critical to model accurately in domains such as risk analysis, reliability engineering, finance, and data science.

NOTE

6.9 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/Use/UnitUrls?unitId=1267	

6.10 REFERENCE

- <https://docs.scipy.org/doc/scipy/reference/stats.html>
- https://en.wikipedia.org/wiki/Probability_distribution
- <https://numpy.org/doc/stable/reference/random/generated/numpy.random.zipf.html>

6.11 EXERCISE

Part 1

A. Multiple choice question

1. What is the key characteristic of the Exponential distribution?

- A) Symmetry around the mean
- B) Models time between independent events
- C) Distribution of categorical outcomes
- D) Uniform probability across range

2. The Chi-Square distribution is primarily used for:

- A) Predicting rare word frequency
- B) Modeling equal probability events
- C) Statistical hypothesis testing and variance analysis
- D) Analyzing logistic regression outcomes

3. Which distribution is often used in modeling the magnitude of a vector with Gaussian components?

- A) Pareto

NOTE

- B) Rayleigh
- C) Zipf
- D) Binomial

4. The Pareto distribution is most appropriate when:

- A) Modeling uniform data
- B) Events follow the 80/20 rule
- C) All outcomes are equally likely
- D) Analyzing normal errors

5. The Zipf distribution is commonly observed in:

- A) Physical measurement errors
- B) Uniform sampling
- C) Word frequency in natural language
- D) Radioactive decay

Answer with Explanations:

1. **B)** Models time between independent events

Explanation: The Exponential distribution models the time between independent events occurring continuously.

2. **C)** Statistical hypothesis testing and variance analysis

Explanation: The Chi-Square distribution is mainly used for hypothesis testing and analyzing variance.

3. **B)** Rayleigh

Explanation: The Rayleigh distribution models the magnitude of a vector whose components follow Gaussian distribution.

4. **B)** Events follow the 80/20 rule

Explanation: Pareto distribution fits scenarios following the 80/20 rule where a small portion accounts for most effects.

5. **C)** Word frequency in natural language

Explanation: Zipf distribution commonly appears in word frequency patterns in natural language.

B. Short answer type question

1. What is the main application of the Exponential distribution in real-world scenarios?
2. How is the Pareto distribution useful in economics or business?
3. What kind of data does the Zipf distribution best represent?

Part 2**C. Long answer type question (200 words)**

1. Explain how the Exponential distribution is used to model time-based events and its significance in reliability engineering.
2. Discuss the characteristics and applications of the Chi-Square distribution in statistical testing.
3. How do Pareto and Zipf distributions help model heavy-tailed data in economics, linguistics, and natural phenomena?

Part 3**D. Long Answer Type Questions : (300 words)**

1. Explain the binomial distribution and how NumPy simulates real-world experiments using it. Provide practical examples.
2. Describe the Poisson distribution. Discuss its applications in modeling rare events and how NumPy is used to generate Poisson samples.
3. Compare multiple advanced distributions (geometric, exponential, chi-square, etc.) and discuss their uses in analytics.

NOTE



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | UCI | Member of: APU | Joint Committee: UGC | UER | AICTE | Recognized by: UGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

BLOCK III	NUMPY UNIVERSAL FUNCTIONS	89-122
Unit 7	Understanding NumPy Ufuncs	91
Unit 8	Operations in NumPy	101
Unit 9	NumPy Mathematics	111



This block explains NumPy’s Universal Functions (ufuncs), which allow fast and efficient mathematical operations on arrays. Students will learn how ufuncs work internally and why they make NumPy much faster than regular Python loops. The block covers common operations such as arithmetic, logical, trigonometric, and statistical functions. It also explores how NumPy handles broadcasting, element-wise operations, and mathematical computations on multi-dimensional arrays. By mastering ufuncs, students will be able to perform high-speed calculations required in data science, engineering, and scientific computing.

UNIT 7

UNDERSTANDING NUMPY UFUNCS

NOTE

UNIT STRUCTURE

- 7.1 Introductions
- 7.2 Objective
- 7.3 Mastering NumPy Ufuncs: From Creation to Computation
- 7.4 Efficient Math with NumPy: Ufuncs, Arithmetic, and Rounding
- 7.5 NumPy Ufuncs Explained: Create, Compute, and Round
- 7.6 Summary
- 7.7 Glossary
- 7.8 Answers to check your progress
- 7.9 SUGGESTED READINGS – OER-OER
- 7.10 Reference
- 7.11 Exercise

7.1 INTRODUCTION

Understanding NumPy's universal functions (ufuncs) is essential for efficient numerical computation in Python. Ufuncs are optimized functions that perform element-wise operations on arrays, enabling high-speed mathematical processing. These functions are designed to be flexible, broadcasting automatically across compatible shapes and supporting multiple input/output arguments. NumPy provides a wide range of ufuncs for arithmetic, trigonometric, exponential, and logical operations. This makes complex mathematical manipulation faster and more concise. Whether you're building machine learning models or running statistical simulations, ufuncs are the foundation of high-performance data handling.

The creation and application of ufuncs not only simplify operations but also reduce the need for explicit loops. Arithmetic ufuncs like add, subtract, multiply, and divide handle array-wise calculations with ease and clarity. Rounding ufuncs such as floor, ceil, and rint help in controlling numerical precision—crucial in financial and scientific data. Ufuncs also support type casting, out parameters, and condition-based computations. Their integration into NumPy's core allows for memory-efficient operations even on large datasets. Learning how to create and apply these functions unlocks a powerful way to process arrays quickly and accurately.

7.2 OBJECTIVE

After completing this Unit we will be able to:

- Define universal functions (ufuncs) in NumPy.
- Explain how ufuncs improve computational efficiency.
- Apply built-in ufuncs for array operations.
- Differentiate between custom and built-in ufuncs.

NOTE

7.3 MASTERING NUMPY UFUNCS: FROM CREATION TO COMPUTATION

NumPy's universal functions (ufuncs) are at the core of fast numerical operations in Python. They allow you to apply operations element-wise on arrays without using explicit loops, making your code cleaner and dramatically faster. Whether you're doing addition, exponentiation, or trigonometric calculations, ufuncs process data efficiently across entire arrays.

Mastering ufuncs starts with understanding built-in functions like `np.add`, `np.subtract`, `np.sqrt`, or `np.sin`, and how they differ from Python loops in terms of performance. These ufuncs support broadcasting, type casting, and error handling, which makes them powerful for advanced computations. You can even create custom ufuncs using `np.frompyfunc()` when NumPy's defaults don't meet specific needs.

In computation-heavy scenarios, knowing how to chain or combine ufuncs improves both clarity and speed. For instance, computing normalized values or applying conditional logic using `np.where` can be seamlessly integrated with ufuncs. With proper use of ufuncs, not only do you reduce runtime, but you also write code that's scalable and easier to maintain in data science and machine learning workflows.

7.4 EFFICIENT MATH WITH NUMPY: UFUNCS, ARITHMETIC, AND ROUNDING

NumPy provides universal functions (ufuncs) that are optimized for element-wise operations on arrays, making mathematical computations both fast and memory-efficient. These ufuncs include basic arithmetic like addition, subtraction, multiplication, division, and more complex operations such as exponentiation and trigonometric functions. Unlike Python loops, ufuncs operate over whole arrays without explicit iteration, which is key to performance.

Alongside ufuncs, NumPy supports broadcasting, allowing arithmetic operations between arrays of different shapes when possible. This eliminates the need to manually reshape data for compatibility. Arithmetic operations (+, -, *, /) work seamlessly with arrays, making it easy to perform calculations across datasets in just one line of code.

Another powerful feature is rounding functions, including `np.round()`, `np.floor()`, `np.ceil()`, and `np.trunc()`. These are used to control the precision of results or prepare data for reporting and display. Together, ufuncs, arithmetic, and rounding streamline numerical analysis and scientific computing.

Code Example:

```
import numpy as np

# Creating arrays
a = np.array([1.5, 2.3, 3.9])
```

NOTE

```
b = np.array([4.2, 1.7, 0.6])
# Using ufuncs for arithmetic
sum_ab = np.add(a, b)
product_ab = np.multiply(a, b)
diff_ab = np.subtract(a, b)
# Rounding operations
rounded = np.round(sum_ab, 1)
floored = np.floor(product_ab)
ceiled = np.ceil(diff_ab)

# Display results
print("Sum:", sum_ab)
print("Product:", product_ab)
print("Rounded Sum:", rounded)
print("Floored Product:", floored)
print("Ceiled Difference:", ceiled)
```

This example showcases:

1. Use of `np.add`, `np.multiply`, and `np.subtract` for element-wise math
2. Application of rounding methods for result formatting

7.5 NUMPY UFUNCS EXPLAINED: CREATE, COMPUTE, AND ROUND

NumPy Universal Functions (ufuncs) are highly efficient functions designed to operate element-wise on arrays. They are built into NumPy and provide fast, vectorized operations without the need for explicit loops. These include arithmetic (add, subtract, multiply), comparison (greater, less_equal), trigonometric, and exponential functions.

The creation of ufuncs begins with initializing arrays using `np.array()`, after which you can perform mathematical operations. For example, `np.add(a, b)` adds corresponding elements from arrays `a` and `b`. These ufuncs support broadcasting, making it possible to apply operations to arrays of different shapes intelligently.

After computation, rounding functions help format or control numeric precision. NumPy offers `np.round()` to round values to a specific number of decimals, `np.floor()` to round down, and `np.ceil()` to round up. These are particularly useful in data presentation and statistical processing.

```
import numpy as np

# Create arrays
x = np.array([2.5, 3.7, 1.2])
y = np.array([1.5, 2.3, 0.8])

# Compute with ufuncs
sum_result = np.add(x, y)
diff_result = np.subtract(x, y)
```

NOTE

```

mult_result = np.multiply(x, y)

# Rounding operations
rounded_result = np.round(mult_result, 2)
floor_result = np.floor(sum_result)
ceil_result = np.ceil(diff_result)

# Output results
print("Sum:", sum_result)
print("Difference:", diff_result)
print("Product:", mult_result)
print("Rounded Product:", rounded_result)
print("Floor of Sum:", floor_result)
print("Ceil of Difference:", ceil_result)

```

This example covers:

1. **Creating** NumPy arrays
2. **Computing** with ufuncs (addition, subtraction, multiplication)
3. **Rounding** results using round, floor, and ceil



Activity 7.1

Create an array of values from 0 to 9.

Apply the following ufuncs and display outputs:

```

`np.square()`
`np.sqrt()`
`np.exp()`

```

.....

.....

.....

☒ Check Your Progress 7.1

Notes:

- a) Space is given below for your answer.
- b) Check your answer with the one given at the end of this unit.

Q1. What defines NumPy ufuncs is their ability to perform fast, element-wise computations turning array math into efficient, readable, and scalable code.

.....

.....

.....

NOTE

Q2. To explain NumPy's power is to highlight its ufuncs: from creation to rounding, they streamline data operations and eliminate the need for complex loops.

.....

.....

.....

7.6 SUMMARY

NumPy universal functions (ufuncs) are optimized, vectorized operations that perform element-wise computations across arrays. They eliminate the need for explicit loops, which enhances code readability and execution speed. Ufuncs support broadcasting, allowing operations between arrays of different shapes. Their creation helps manage data transformations effectively in numerical computing.

Arithmetic operations like addition, subtraction, multiplication, and division are efficiently handled by ufuncs. These operations can be performed across entire arrays in a single call, making large-scale computation seamless. Ufuncs also offer features like output control, reducing temporary memory usage. They are foundational for mathematical and statistical modeling in Python.

Rounding operations in NumPy, such as `np.floor()`, `np.ceil()`, and `np.round()`, provide precise control over numerical formatting. These are particularly useful in financial, scientific, and graphical applications where precision matters. Ufuncs help round entire datasets efficiently without loops. Their utility in cleaning and formatting data adds another layer of flexibility to array handling.

7.6 GLOSSARY

- **Universal Function (Ufunc)** – A NumPy function that performs element-wise operations efficiently on arrays.
- **Element-wise Operation** – An operation applied individually to each element of an array.
- **Unary Ufunc** – A universal function that operates on a single input array.
- **Binary Ufunc** – A universal function that performs operations on two input arrays.
- **Broadcasting** – A mechanism that enables operations on arrays of different shapes by automatically expanding dimensions.
- **Vectorized Computation** – Performing operations on entire arrays without using explicit loops.
- **Reduction Operation** – A process of reducing an array to a single value, such as sum or maximum.

NOTE

- **Accumulation** – A sequential operation that stores intermediate results, such as cumulative sums.
- **Output Array** – The array in which the results of a ufunc operation are stored.
- **Mathematical Ufunc** – A universal function used to perform mathematical operations such as square root, exponential, or logarithm.

7.8 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 7.1

Answer 1:

NumPy ufuncs (universal functions) are core tools in NumPy that allow for **element-wise operations** on arrays, enabling the application of mathematical functions over entire datasets without writing explicit loops. Their power lies in **vectorization**, which translates Python operations into optimized low-level C code for speed.

These functions—like `np.add()`, `np.subtract()`, `np.sqrt()`, and many others—are not only **fast**, but also **readable**, making code cleaner and easier to maintain. They support **broadcasting**, meaning they can operate on arrays of different shapes when certain conditions are met, making the code highly **scalable** for larger data.

```
import numpy as np

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

result = np.add(a, b) # Output: array([5, 7, 9])
```

Here, `np.add()` performs a fast, element-wise addition on both arrays—no loops required. This showcases exactly what the quotation highlights: fast computation, efficient syntax, and scalable data processing using ufuncs.

Answer 2:

NumPy's power comes largely from **ufuncs (universal functions)**, which simplify mathematical operations by applying them **element-wise** over entire arrays. Whether it's **creating ufuncs**, performing **arithmetic**, or **rounding** values, these functions replace traditional, verbose for loops with concise, high-performance code.

From basic arithmetic like `np.multiply()` to more advanced operations like `np.floor()` or `np.modf()`, ufuncs allow developers to write **fast, clean, and**

scalable code. These functions are **vectorized**, meaning they process array elements in parallel, which leads to massive performance boosts.

Example:

```
import numpy as np
arr = np.array([3.14, 2.71, 1.41])
rounded = np.round(arr) # Output: array([3., 3., 1.])
```

This single line of code rounds all values in the array—no loop required. Such capabilities make NumPy a powerful tool for scientific computing, data analysis, and machine learning, by abstracting low-level operations and focusing on efficient logic.

NOTE

7.9 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/Use/UnitUrls?unitId=1268	

7.10 REFERENCE

- <https://numpy.org/doc/stable/reference/ufuncs.html>
- <https://realpython.com/numpy-ufuncs-guide/>
- <https://towardsdatascience.com/numpy-ufuncs-explained>

7.11 EXERCISE

Part 1

A. Multiple choice question

1. What is the main purpose of NumPy ufuncs?

- A) To create 3D plots
- B) To handle string operations
- C) To perform element-wise operations on arrays
- D) To convert arrays to lists

2. Which of the following is a ufunc used for arithmetic operations in NumPy?

- A) np.concatenate()

NOTE

- B) np.add()
- C) np.linspace()
- D) np.array_split()

3. What does the np.floor() function do?

- A) Rounds values up to the nearest integer
- B) Rounds values down to the nearest integer
- C) Returns only the decimal part
- D) Calculates the factorial

4. Which function is used to apply a universal function to two arrays simultaneously?

- A) np.combine()
- B) np.apply_ufunc()
- C) np.vectorize()
- D) np.add()

5. What does broadcasting in NumPy ufuncs allow?

- A) Sending arrays over the internet
- B) Executing a function on only one element
- C) Applying operations on arrays of different shapes
- D) Sorting arrays in place

Answer with Explanations :

1. C) To perform element-wise operations on arrays

Explanation: Ufuncs perform fast element-wise operations on NumPy arrays.

2. B) np.add()

Explanation: np.add() is an arithmetic ufunc that adds elements element-wise

3. B) Rounds values down to the nearest integer

Explanation: np.floor() rounds values down to the nearest integer.

4. D) np.add()

Explanation: Ufuncs like np.add() operate on two arrays simultaneously in an element-wise manner.

5. C) Applying operations on arrays of different shapes

Explanation: Broadcasting lets NumPy apply operations to arrays of different shapes when compatible.

B. Short answer type question

1. What is a NumPy ufunc and why is it used?

2. Give an example of a ufunc used for arithmetic operations.
3. Which ufuncs are commonly used for rounding values in NumPy?

NOTE**Part 2****C. Long answer type question (200 words)**

1. What are NumPy ufuncs and how do they differ from traditional Python functions in terms of performance and usage?
2. Explain how arithmetic operations are performed using NumPy ufuncs with examples. Why are these preferred over loops?
3. Describe the rounding functions provided by NumPy ufuncs. How are they useful in data analysis and numerical computation?

Part 3**D. Long Answer Type Questions (300 words)**

1. Explain the concept, structure, and advantages of NumPy Ufuncs. Discuss how Ufuncs help in fast element-wise operations.
2. Describe the difference between unary and binary Ufuncs, with examples. Explain why Ufuncs improve computational performance.
3. Discuss Ufunc attributes such as reduce, accumulate, and outer. Explain how they enhance mathematical operations.

UNIT 8

OPERATIONS IN NUMPY

NOTE

UNIT STRUCTURE

- 8.1 Introductions
- 8.2 Objective
- 8.3 Optimizing Array Computations with NumPy Ufuncs
- 8.4 Mastering NumPy: From Ufunc Creation to Precise Rounding
- 8.5 Summary
- 8.6 Glossary
- 8.7 Answers to check your progress
- 8.8 Suggested readings-OER
- 8.9 Reference
- 8.10 Exercise

8.1 INTRODUCTION

Mathematical operations are fundamental in any data analysis or scientific computation task, and NumPy provides highly efficient tools for such operations. NumPy's rich suite includes logarithmic functions, summations, products, differences, and number-theoretic operations like LCM and GCD. These operations are executed element-wise across arrays, making them scalable for large datasets. Whether you're performing financial modeling, signal processing, or statistical computation, NumPy simplifies the workflow. With built-in broadcasting and vectorization, you can handle complex numerical logic without loops. This efficiency allows data scientists and engineers to focus more on logic than on manual computation.

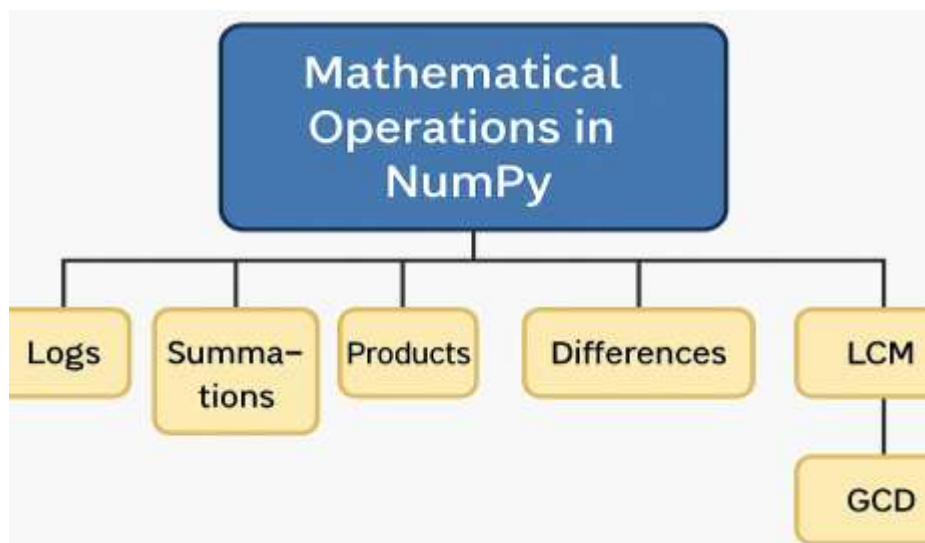


Figure 8.1: Numpy Mathematical Operations

NOTE

From calculating log transformations for normalization to finding the greatest common divisor in arrays of integers, NumPy ensures accuracy and speed. Summation and product functions work seamlessly across any axis, enabling multi-dimensional array processing with ease. The difference function reveals how values change over intervals—a crucial metric in time-series and trend analysis. Additionally, LCM and GCD operations help solve problems in number theory, cryptography, and system design. Together, these mathematical tools form a critical backbone for real-world applications. With just a few lines of code, users can unleash the full computational power NumPy offers.

8.2 OBJECTIVE

After completing this Unit we will be able to:

- Perform arithmetic and logical operations on arrays.
- Apply aggregation functions such as sum, mean, and min/max.
- Analyze array operations for performance optimization.
- Demonstrate element-wise and matrix operations.

8.3 OPTIMIZING ARRAY COMPUTATIONS WITH NUMPY UFUNCS

NumPy Universal Functions (Ufuncs) are at the core of fast and efficient numerical computations in Python. These functions are designed to perform element-wise operations on entire arrays without the need for explicit loops, which makes them not only concise but also computationally optimized using C under the hood.

Ufuncs cover a wide range of mathematical operations—such as addition, subtraction, multiplication, division, trigonometric functions, exponentials, and more. What makes them powerful is vectorization—a process that replaces slow Python loops with fast, low-level C operations. This reduces execution time and improves code readability.

Another key benefit of Ufuncs is broadcasting, which allows operations between arrays of different shapes under certain conditions. This allows for flexible computations without reshaping arrays manually. Ufuncs also support features like output control, accumulation, reduction, and type casting.

For example:

```
import numpy as np

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

# Use a ufunc for addition
result = np.add(a, b)
print(result) # Output: [5 7 9]
```

NumPy Ufuncs help you optimize performance, reduce complexity, and write cleaner, more expressive numerical code—an essential tool for data science, machine learning, and scientific computing.

NOTE

8.4 MASTERING NUMPY: FROM UFUNC CREATION TO PRECISE ROUNDING

Mastering NumPy: From Ufunc Creation to Precise Rounding means gaining control over one of the most powerful features in NumPy—universal functions (ufuncs). These functions operate element-wise on ndarray objects and are optimized for performance, removing the need for explicit loops. Whether you're generating new ufuncs using NumPy's `frompyfunc()` or applying existing ones for arithmetic, comparisons, or logical operations, they provide a consistent, vectorized way to manipulate arrays efficiently.

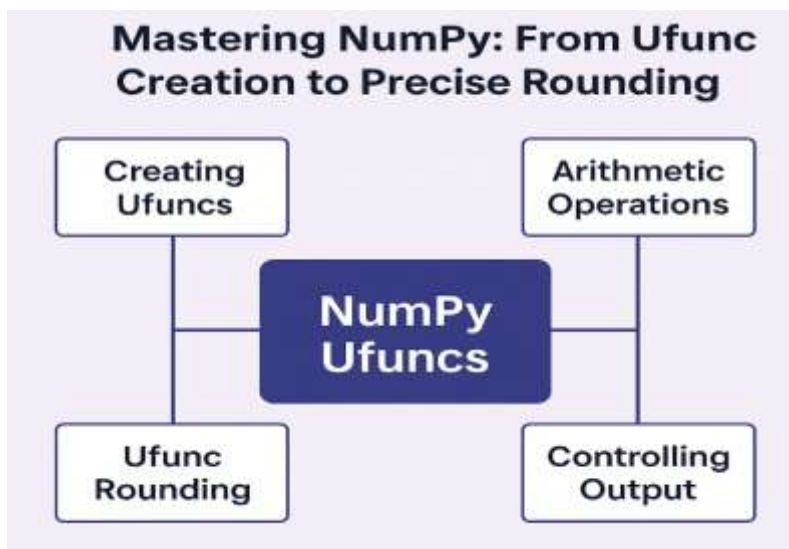


Figure 8.2: NumPy Ufuncs

The journey starts with creating ufuncs, where custom operations are converted into functions that can be broadcasted across arrays. Then, arithmetic operations such as addition, subtraction, multiplication, and division are handled effortlessly and faster than traditional Python loops. NumPy ensures that these computations are memory-efficient and scalable, especially with large datasets.

Finally, rounding operations like `np.floor()`, `np.ceil()`, and `np.round()` help refine numerical data with precision. These functions are essential for aligning results in statistical summaries, graphical presentations, or machine learning pipelines. By mastering this full flow—from creation to rounding—you unlock the true computational power of NumPy, enabling fast, readable, and maintainable numerical code for a wide range of scientific and engineering tasks.

complete code example that walks you through the concept of "Mastering NumPy: From Ufunc Creation to Precise Rounding" using:

NUMPY UNIVERSAL FUNCTIONS

NOTE

- Custom ufunc creation
- Basic arithmetic operations
- Rounding functions

```
import numpy as np

# Step 1: Creating a custom ufunc (universal function)
# This function adds 10 to each element
def add_ten(x):
    return x + 10

# Convert Python function to a NumPy ufunc
add_ten_ufunc = np.frompyfunc(add_ten, 1, 1)

# Sample array
arr = np.array([1.2, 2.5, 3.8, 4.6])

# Step 2: Apply custom ufunc
result_add_ten = add_ten_ufunc(arr)

# Step 3: Arithmetic operations using built-in ufuncs
added = np.add(arr, 2)      # Adds 2 to each element
multiplied = np.multiply(arr, 3) # Multiplies each element by 3

# Step 4: Rounding operations
floored = np.floor(arr)     # Rounds down
ceiled = np.ceil(arr)       # Rounds up
rounded = np.round(arr, 1)  # Rounds to 1 decimal place

# Output
print("Original Array:", arr)
print("Custom Ufunc (add 10):", result_add_ten)
print("Added 2:", added)
print("Multiplied by 3:", multiplied)
print("Floor:", floored)
print("Ceil:", ceiled)
print("Rounded to 1 decimal:", rounded)
```

Highlights:

- `np.frompyfunc()` turns any Python function into a NumPy ufunc.

- Arithmetic operations like `np.add()` and `np.multiply()` are fast, vectorized.
- `np.floor()`, `np.ceil()`, and `np.round()` ensure rounding precision.

This example gives you both custom and built-in ufunc power for efficient array computations in NumPy.

NOTE



Activity 8.1

Create two arrays of size 5 and perform:

Element-wise addition

Element-wise multiplication

Element-wise comparison (which values are equal?)

.....

.....

.....

☑ Check Your Progress 8.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. What makes NumPy indispensable is its ability to reduce complex mathematical operations like logs, sums, and products into simple, efficient array functions that scale with your data.?

.....

.....

.....

Q2. What you can compute with NumPy isn't just numbers it's insight; from differences to GCD, each function reveals patterns that power meaningful analysis.

.....

.....

.....

8.5 SUMMARY

NumPy provides a powerful suite of mathematical functions that allow for efficient numerical analysis and computation on arrays. Functions like `np.log()` and `np.log10()` help compute natural and base-10 logarithms, crucial for data scaling, scientific calculations, and understanding exponential relationships. Summation (`np.sum`) and product (`np.prod`) operations quickly aggregate array values, enabling streamlined statistical and algebraic processing. These

NOTE

functions work efficiently on both one-dimensional and multi-dimensional arrays with axis-wise operations. Moreover, `np.diff()` helps compute successive differences between elements, which is essential in signal processing, time-series analysis, and numerical differentiation. Each function is optimized to perform element-wise operations without the need for explicit loops.

Additionally, NumPy includes functions like `np.lcm()` and `np.gcd()` to compute least common multiples and greatest common divisors, which are vital in discrete mathematics and number theory applications. These operations can be extended across arrays using `np.lcm.reduce()` and `np.gcd.reduce()` to compute results over entire data sets. Whether used in linear algebra, data transformation, or engineering analysis, these mathematical tools make NumPy a core part of the scientific Python ecosystem. Their ability to handle large data efficiently supports high-performance computation and lays the foundation for advanced analytics and modeling. As a result, mastering these tools enhances both clarity and speed in data-driven workflows.

8.6 GLOSSARY

- **Arithmetic Operations** – Mathematical operations such as addition, subtraction, multiplication, and division performed on arrays.
- **Comparison Operations** – Operations that compare array elements and return Boolean values.
- **Logical Operations** – Operations that evaluate logical conditions using Boolean arrays.
- **Array Broadcasting** – The automatic adjustment of array dimensions to perform compatible operations.
- **Matrix Multiplication** – The mathematical operation of multiplying two matrices according to matrix algebra rules.
- **Dot Product** – The sum of the products of corresponding elements of two vectors or arrays.
- **Aggregate Functions** – Functions that compute summary statistics such as sum, mean, minimum, and maximum.
- **Sorting Operation** – The process of arranging array elements in ascending or descending order.
- **Searching Operation** – The process of locating specific elements or conditions within an array.
- **Conditional Selection** – Selecting array elements based on specified logical conditions.

8.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 8.1

Answer 1:

NumPy is a core Python library designed for numerical computing. Its true power lies in **universal functions (ufuncs)** like `np.log`, `np.sum`, `np.prod`, and

NOTE

others, which allow users to perform advanced mathematical operations directly on arrays **without writing explicit loops**. This not only improves code **readability** but also drastically enhances **performance**.

These functions are **vectorized**, meaning they operate on entire arrays at once, leveraging C-level speed. Whether you're summing a million numbers, taking logarithms of large datasets, or calculating products in multidimensional matrices, NumPy handles it **fast and efficiently**.

Matters:

- Reduces code complexity.
- Enables big data scalability.
- Supports real-time analytics and modeling.
- Promotes clean and maintainable syntax.

Answer 2:

This quote highlights the **transformational power** of NumPy. While it may appear that NumPy only processes numbers, what it actually provides is the **foundation for extracting insights** from data. Each mathematical operation—like computing differences with `np.diff()`, or finding the Greatest Common Divisor using `np.gcd()`—helps reveal trends, anomalies, or hidden structures in datasets.

For instance:

- `np.diff()` helps identify **rate of change**, revealing how values evolve over time.
- `np.sum()` and `np.prod()` measure **aggregates** that summarize entire datasets.
- `np.gcd()` and `np.lcm()` can be used in **modular arithmetic**, cryptography, and signal processing to find harmony or alignment between number sets.

NumPy doesn't just simplify calculations; it enables **pattern discovery** that drives decisions in fields like data science, engineering, and machine learning.

8.8 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/Use/UnitUrls?unitId=1269	

8.9 REFERENCE

- <https://numpy.org/doc/stable/reference/routines.math.html>

NOTE

- <https://realpython.com/numpy-array-programming/>
- https://www.w3schools.com/python/numpy/numpy_math.asp

8.10 EXERCISE

Part 1

A. Multiple choice question

- Which NumPy function is used to compute the natural logarithm (base e) of elements in an array?**
 - np.log10()
 - np.exp()
 - np.log()
 - np.sqrt()
- What does np.sum(arr) return when arr = np.array([1, 2, 3, 4])?**
 - 24
 - 10
 - 9
 - 0
- Which NumPy function calculates the Least Common Multiple (LCM) of two or more integers?**
 - np.gcd()
 - np.lcm()
 - np.product()
 - np.min()
- What does np.diff([2, 5, 9, 14]) output?**
 - [3, 4, 5]
 - [7, 4, 2]
 - [2, 3, 4]
 - [3, 4, 5]
- Which NumPy function computes the product of all elements in an array?**
 - np.prod()
 - np.product()
 - np.multiply()
 - np.sum()

Answer with Explanations:

- C) np.log()**

Explanation: np.log() computes the natural logarithm (base e) of each array element.

NOTE

2. B) 10

Explanation: The sum of 1+2+3+4 equals 10.

3. B) np.lcm()

Explanation: np.lcm() calculates the least common multiple of integers.

4. A) [3, 4, 5]

Explanation: np.diff() returns consecutive differences: 5-2, 9-5, 14-9.

5. A) np.prod()

Explanation: np.prod() multiplies all elements of the array together.

B. Short answer type question

1. What is the purpose of the np.log() function in NumPy?
2. How do np.sum() and np.prod() differ in their operation on a NumPy array?
3. When would you use np.lcm.reduce() instead of np.lcm() in NumPy?

Part 2

C. Long answer type question (200 words)

1. Explain how logarithmic functions (np.log, np.log10, np.log2) in NumPy are used for data transformation and analysis. Provide use cases and advantages.
2. Discuss how summation, product, and difference functions in NumPy contribute to efficient array computation. Include examples and performance considerations.
3. Describe the role of np.lcm() and np.gcd() in mathematical modeling. How do they work with arrays and what are their applications?

Part 3

D. Long Answer Type Questions (300 words)

1. Explain arithmetic, logical, and comparison operations in NumPy. Compare them with Python's built-in operations.
2. Describe how NumPy handles broadcasting during operations. Explain why broadcasting eliminates the need for loops.
3. Discuss aggregate operations such as sum, mean, min, max, and cumulative functions.

UNIT 9

NUMPY MATHEMATICS

NOTE

UNIT STRUCTURE

- 9.1 Introductions
- 9.2 Objective
- 9.3 Exploring NumPy's Trigonometric Functions: From Angles to Applications.
- 9.4 Understanding Hyperbolic Functions in NumPy for Scientific Computing
- 9.5 Efficient Data Handling with NumPy Set Operations: Union, Intersection, and Difference
- 9.6 Summary
- 9.7 Glossary
- 9.8 Answers to check your progress
- 9.9 Suggested readings - OER
- 9.10 Reference
- 9.11 Exercise

9.1 INTRODUCTION

NumPy is a powerful Python library that simplifies complex mathematical operations, especially those involving arrays. Among its wide range of functions, trigonometric and hyperbolic operations play a vital role in scientific and engineering applications. Trigonometric functions like sine, cosine, and tangent help model periodic behavior, such as waves or oscillations. With NumPy, these operations can be applied element-wise to entire arrays, making calculations fast and efficient. It supports both angle-based input in radians and transformations for degrees. This enables users to simulate real-world scenarios like signal processing or circular motion. Moreover, NumPy ensures numerical stability and precision in computations. This consistency is critical when handling large datasets in computational tasks.

Hyperbolic functions, such as $\sinh$, $\cosh$, and $\tanh$, are essential in fields like physics, machine learning, and differential equations. NumPy provides built-in support for these functions and their inverses, allowing users to perform hyperbolic transformations quickly. These functions are used in neural network activation and modeling of natural growth phenomena. In addition to trigonometric and hyperbolic capabilities, NumPy also offers powerful set operations for array comparisons and data filtering. Functions like `union1d`, `intersect1d`, and `setdiff1d` help manage and analyze data relationships. These tools are especially useful in data preprocessing and feature selection tasks. Altogether, these features make NumPy a complete toolkit for mathematical computation and data analysis.

NOTE

9.2 OBJECTIVE

After completing this Unit we will be able to:

- Apply mathematical functions such as trigonometric and logarithmic operations.
- Compute statistical measures using NumPy.
- Analyze numerical data using mathematical transformations.
- Evaluate the accuracy and efficiency of computations.

9.3 EXPLORING NUMPY'S TRIGONOMETRIC FUNCTIONS: FROM ANGLES TO APPLICATIONS

NumPy's trigonometric functions provide a powerful toolkit for handling angle-related computations, especially when working with periodic data such as waves, circular motion, and signal processing. These functions operate element-wise on NumPy arrays, allowing for fast, vectorized computation of sine, cosine, tangent, and their inverses.

With functions like `np.sin()`, `np.cos()`, and `np.tan()`, one can model oscillations, phase shifts, and angular velocities efficiently. Additionally, inverse trigonometric functions such as `np.arcsin()` and `np.arctan()` help retrieve angles from values, making them essential for geometric and scientific applications. Conversion functions like `np.deg2rad()` and `np.rad2deg()` are also available to switch between degrees and radians easily. By integrating these functions into data pipelines, analysts can simulate cyclic patterns, calculate angular distances, or analyze frequency-based signals. Whether applied in physics simulations, robotics, or engineering, NumPy's trigonometric capabilities transform raw numeric input into meaningful, interpretable angular behavior.

A table summarizing NumPy Mathematics: Trigonometric, Hyperbolic, and Set Operations, along with their key functions and applications.

Table 9.1: NumPy Mathematical Operations

Category	Common NumPy Functions	Key Applications
Trigonometric	<code>np.sin()</code> , <code>np.cos()</code> , <code>np.tan()</code> , <code>np.arcsin()</code> , <code>np.arccos()</code> , <code>np.arctan()</code> , <code>np.deg2rad()</code> , <code>np.rad2deg()</code>	Wave analysis, angle conversions, circular motion, signal processing
Hyperbolic	<code>np.sinh()</code> , <code>np.cosh()</code> , <code>np.tanh()</code> , <code>np.arcsinh()</code> , <code>np.arccosh()</code> , <code>np.arctanh()</code>	Modeling growth/decay, engineering simulations, neural networks
Set Operations	<code>np.union1d()</code> , <code>np.intersect1d()</code> , <code>np.setdiff1d()</code> , <code>np.setxor1d()</code> , <code>np.unique()</code>	Finding unique elements, comparing datasets, logical array analysis

9.4 UNDERSTANDING HYPERBOLIC FUNCTIONS IN NUMPY FOR SCIENTIFIC COMPUTING

NOTE

Hyperbolic functions in NumPy are essential tools in scientific computing, especially when modeling phenomena involving growth, decay, and signal processing. Just as trigonometric functions relate to circular motion, hyperbolic functions are associated with hyperbolas and appear in contexts like special relativity, electrical engineering, and mathematical modeling. NumPy offers built-in universal functions (ufuncs) like `np.sinh()`, `np.cosh()`, and `np.tanh()` for computing the hyperbolic sine, cosine, and tangent of input arrays efficiently.

These functions are vectorized, meaning they operate element-wise on NumPy arrays, allowing fast computation over large datasets without the need for explicit loops. For example, `np.cosh(x)` computes the hyperbolic cosine for each element in `x`, whether it's a scalar, list, or multidimensional array.

Additionally, inverse functions such as `np.arcsinh()`, `np.arccosh()`, and `np.arctanh()` enable the solving of equations where the hyperbolic value is known, and the angle or input must be determined.

In scientific domains, hyperbolic functions model heat transfer, describe shape curves like cables or arches, and represent activation functions in neural networks. Their seamless integration in NumPy makes complex mathematical modeling more accessible, accurate, and performant across a variety of disciplines.

A simple and clear Python code example demonstrating how to use NumPy's hyperbolic functions for scientific computing:

```
import numpy as np
import matplotlib.pyplot as plt

# Generate a range of values
x = np.linspace(-4, 4, 100)

# Apply hyperbolic functions
sinh_values = np.sinh(x)
cosh_values = np.cosh(x)
tanh_values = np.tanh(x)

# Plotting the functions
plt.figure(figsize=(10, 6))
plt.plot(x, sinh_values, label='sinh(x)', color='blue')
```

NOTE

```
plt.plot(x, cosh_values, label='cosh(x)', color='green')
plt.plot(x, tanh_values, label='tanh(x)', color='red')
plt.title("Hyperbolic Functions using NumPy")
plt.xlabel("x values")
plt.ylabel("Function values")
plt.axhline(0, color='gray', linestyle='--', linewidth=0.5)
plt.axvline(0, color='gray', linestyle='--', linewidth=0.5)
plt.legend()
plt.grid(True)
plt.show()
```

This code does:

- Creates a range of x values from -4 to 4.
- Computes sinh, cosh, and tanh using NumPy.
- Plots all three functions on the same graph for easy comparison.

9.5 EFFICIENT DATA HANDLING WITH NUMPY SET OPERATIONS: UNION, INTERSECTION, AND DIFFERENCE

NumPy's set operations offer a powerful and efficient way to handle unique elements in arrays, especially when performing comparisons, filtering, or deduplication in data analysis tasks. These operations—union, intersection, and difference—mimic standard set theory concepts and are implemented using optimized functions such as `np.union1d()`, `np.intersect1d()`, and `np.setdiff1d()`.

1. **Union (`np.union1d`)** combines two arrays and returns the sorted, unique elements found in either of them. This is ideal when merging datasets and ensuring no duplication.
2. **Intersection (`np.intersect1d`)** returns only the elements present in both arrays. This is useful when identifying common records or shared data points between datasets.
3. **Difference (`np.setdiff1d`)** provides elements from one array that are not present in another, which is especially helpful for filtering or exclusion logic.

These operations are particularly efficient because they're vectorized and avoid the overhead of loops or manual comparisons. They help clean, compare, and reconcile datasets—making them essential tools in data preprocessing, machine learning pipelines, and data science workflows. Whether comparing user lists, filtering logs, or joining datasets, NumPy's set operations offer speed, clarity, and scalability.

NOTE

1. **Union (`np.union1d`) – Combine Unique Elements:** This function merges two arrays and returns a sorted array of unique elements from both. It's useful for combining datasets without duplication.

```
np.union1d([1, 2, 3], [3, 4, 5]) # Output: [1 2 3 4 5]
```

2. **Intersection (`np.intersect1d`) – Find Common Data:** It identifies the shared elements between two arrays. This is helpful when comparing datasets to find overlaps or common records.

```
np.intersect1d([1, 2, 3], [2, 3, 4]) # Output: [2 3]
```

3. **Difference (`np.setdiff1d`) – Exclude Unwanted Values:** It returns elements present in the first array but not in the second. Useful for filtering out specific data points or identifying what's missing.

```
np.setdiff1d([1, 2, 3, 4], [2, 4]) # Output: [1 3]
```

4. **Optimized Performance for Large Arrays:** All these functions are implemented efficiently using NumPy's internal C-based operations, making them ideal for large-scale data operations in real-time analysis or preprocessing.
5. **Enhances Clarity and Conciseness:** Set operations eliminate the need for loops and conditional logic, resulting in clean, readable, and maintainable code—especially important in collaborative and production environments.



Activity 9.1

Create a NumPy array of 20 random integers between 10 and 100.

Compute and print:

- **Mean**
- **Median**
- **Standard deviation**
- **Sum of all elements**

.....

.....

.....

Check Your Progress: 9.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

NOTE

Q1. What NumPy delivers in trigonometry is accurate angle computation at lightning speed.

.....

Q2. What makes NumPy powerful is its ability to unify trigonometric, hyperbolic, and set operations under one roof.

.....

9.6 SUMMARY

NumPy offers a wide range of mathematical functions that are essential for scientific and engineering tasks. Its trigonometric functions like `sin()`, `cos()`, and `tan()` allow precise angle-based calculations commonly used in signal processing, physics, and geometry. These functions accept angles in radians and can be converted easily using `deg2rad()` or `rad2deg()` utilities. NumPy's ability to handle arrays enables batch processing of multiple angles efficiently. Alongside direct trigonometric evaluation, inverse functions like `arcsin()` and `arccos()` are also supported. These features provide seamless integration with applications requiring complex angular transformations. The performance of these operations is highly optimized for large datasets.

NumPy also includes hyperbolic functions like `sinh()`, `cosh()`, and `tanh()` which are vital for solving equations in hyperbolic geometry and relativistic physics. Furthermore, NumPy's set operations such as `union1d()`, `intersect1d()`, and `setdiff1d()` are powerful tools for data comparison and filtering. These operations help in identifying unique values, finding overlaps, and computing differences across datasets. Their ability to handle large, unordered data arrays makes them essential in data science and machine learning pipelines. Together, trigonometric, hyperbolic, and set operations enhance NumPy's role as a core tool for mathematical computation. This unified functionality allows developers to build accurate and scalable analytical models with minimal code.

9.7 GLOSSARY

- **Mathematical Functions** – Built-in NumPy functions used for performing arithmetic and scientific calculations.
- **Trigonometric Functions** – Functions such as sine, cosine, and tangent used for angle-based calculations.
- **Exponential Function** – A function that raises Euler's number to a specified power.
- **Logarithmic Function** – A function that calculates the logarithm of array elements to a specified base.

NOTE

- **Rounding Functions** – Functions used to round numerical values to the nearest integer or decimal place.
- **Statistical Functions** – Functions that calculate statistical measures such as mean, median, variance, and standard deviation.
- **Absolute Value** – The non-negative magnitude of a number regardless of its sign.
- **Power Function** – A function that raises array elements to a specified exponent.
- **Square Root** – A mathematical function that computes the value whose square equals the given number.
- **Linear Algebra Functions** – NumPy functions used for matrix operations such as determinant, inverse, and eigenvalue calculations.

9.8 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 9.1

Answer 1:

This statement emphasizes **NumPy's trigonometric capabilities**, which are both **fast** and **precise**, thanks to its underlying C-based implementation and vectorized operations.

NumPy achieves this:

1. **Efficient Functions:** NumPy provides fast, built-in trigonometric functions like `np.sin()`, `np.cos()`, and `np.tan()` that operate on **arrays**, not just single values.
2. **Vectorized Computation:** These functions work element-wise on entire arrays, eliminating the need for loops.
3. **Precision:** NumPy uses **IEEE floating-point standards**, ensuring high precision in angle calculations.
4. **Radians Support:** Functions expect inputs in **radians**, aligning with most scientific computing standards.
5. **Speed & Scale:** Because it's optimized for numerical arrays, NumPy can compute millions of trigonometric values in milliseconds.

Example Code:

```
import numpy as np
angles_deg = np.array([0, 30, 45, 60, 90])
angles_rad = np.radians(angles_deg)

sin_values = np.sin(angles_rad)

print("Sine values:", sin_values)
```

Output:

```
Sine values: [0.         0.5       0.70710678 0.8660254  1.        ]
```

NOTE

This showcases **accurate and fast computation** of sine values for multiple angles — the essence of the quotation.

Answer 2:

This statement highlights **NumPy's versatility** as a comprehensive mathematical toolkit. Let's break it down:

1. **Unified Library:** NumPy offers a **centralized library** for performing a wide variety of mathematical operations from angle-based calculations to array logic.
2. **Trigonometric Functions:** Functions like `np.sin()`, `np.cos()`, `np.tan()` allow for efficient computation of angles useful in signal processing, physics, and engineering.
3. **Hyperbolic Functions:** Functions such as `np.sinh()`, `np.cosh()`, and `np.tanh()` model hyperbolic curves, used in areas like relativity theory and machine learning.
4. **Set Operations:** With `np.union1d()`, `np.intersect1d()`, and `np.setdiff1d()`, NumPy supports mathematical set operations — vital in data science, filtering, and logic comparisons.
5. **Consistent API:** These diverse operations share a **consistent syntax and performance**, making NumPy intuitive and powerful for developers.

Example Code:

```
import numpy as np
# Trigonometric
angles = np.array([0, np.pi/2, np.pi])
print("Sine:", np.sin(angles))
# Hyperbolic
print("Tanh:", np.tanh(angles))

# Set Operation
a = np.array([1, 2, 3])
b = np.array([3, 4, 5])
print("Union:", np.union1d(a, b))
```

Output:

```
Sine: [0.0000000e+00 1.0000000e+00 1.2246468e-16]
Tanh: [0.          0.91715234 0.99627208]
Union: [1 2 3 4 5]
```

This code confirms that NumPy handles all three types of operations seamlessly — **validating its power and integration**.

NOTE

9.9 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/Use/r/UnitUrls?unitId=1270	

9.10 REFERENCE

- <https://numpy.org/doc/stable/reference/routines.math.html>
- <https://jakevdp.github.io/PythonDataScienceHandbook/>
- <https://www.w3schools.com/python/numpy/default.asp>

9.11 EXERCISE

Part 1

A. Multiple choice question:

- What function does NumPy use to compute the sine of an angle (in radians)?**
 - np.sine()
 - np.sin()
 - np.trig_sin()
 - np.sind()
- Which NumPy function returns the hyperbolic cosine of an input array?**
 - np.hcos()
 - np.hyp_cos()
 - np.cosh()
 - np.hypercos()
- What will np.intersect1d([1, 2, 3], [2, 3, 4]) return?**
 - [1, 4]
 - [2, 3]
 - [1, 2, 3, 4]
 - [2, 4]
- Which NumPy function computes the union of two arrays?**
 - np.union()
 - np.union1d()
 - np.join()

NOTE

D. `np.merge1d()`

5. The function `np.tan(np.pi / 4)` will return a value closest to:

- A. 0
- B. 0.5
- C. 1
- D. Undefined

Answer with Explanations :

1. B) `np.sin()`

Explanation: `np.sin()` computes the sine of angles given in radians.

2. C) `np.cosh()`

Explanation: `np.cosh()` returns the hyperbolic cosine of each element.

3. B) `[2, 3]`

Explanation: `intersect1d()` gives common elements found in both arrays.

4. B) `np.union1d()`

Explanation: `np.union1d()` returns the sorted union of two arrays.

5. C) 1

Explanation: $\tan(\pi/4)$ equals 1, as it corresponds to 45 degrees.

B. Short answer type question

1. What is the purpose of `np.sin()` and how is the input interpreted?
2. How do you compute the union of two NumPy arrays containing unique values?
3. What does `np.tanh(x)` compute and in which field is it commonly used?

Part 2

C. Long answer type question (200 words)

1. Explain how NumPy's trigonometric functions support scientific and engineering computations.
2. Describe how NumPy handles hyperbolic functions and their applications in real-world problems.
3. How do NumPy's set operations enhance data analysis and array comparison tasks?

Part 3

D. Long Answer Type Questions (300 words)

1. Discuss mathematical operations available in NumPy. Explain how they are used in scientific computing and analytics.
2. Explain statistical functions in NumPy and how they help in analyzing datasets.
3. Describe trigonometric and exponential functions in NumPy with examples showing practical applications.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | UCI | Member of: AU | Joint Committee: UGC | UER | AICTE | Recognized by: UGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

BLOCK IV	DATA MANIPULATION WITH PANDAS	123-162
Unit 10	Introduction to Pandas	125
Unit 11	Data Cleaning in Pandas	139
Unit 12	Data Analysis with Pandas	151



This block introduces Pandas, a powerful Python library used for working with structured data. Students will learn how to create and use Series and DataFrames, handle tabular data, perform indexing and filtering, and clean messy datasets. The block also covers essential methods for exploring, transforming, and analyzing data in a professional manner. Students will gain hands-on experience in handling missing values, combining datasets, grouping data, and preparing information for further analysis. By the end of this block, students will have the skills needed for effective data processing in real-world applications.

UNIT 10

INTRODUCTION TO PANDAS

NOTE

UNIT STRUCTURE

- 10.1 Introduction**
- 10.2 Objective**
- 10.3 Getting Started with Pandas: From Series to DataFrames**
- 10.4 Data Analysis Made Easy: Importing and Exploring Data with Pandas**
- 10.5 Summary**
- 10.6 Glossary**
- 10.7 Answers to Check Your Progress.**
- 10.8 SUGGESTED READINGS – OER -OER**
- 10.9 Reference**
- 10.10 Exercise**

10.1 INTRODUCTION

Pandas is a powerful and flexible open-source data analysis and manipulation library for Python. It offers data structures like Series and DataFrames that simplify complex data workflows by organizing data in tabular formats. These structures are designed to handle heterogeneous data, making them essential for data wrangling, preprocessing, and statistical analysis in modern data science projects.

At the core of Pandas are Series, which represent one-dimensional labeled arrays, and DataFrames, which are two-dimensional, table-like structures. These allow users to easily access, manipulate, and filter data using intuitive and efficient operations. With built-in support for handling missing data, grouping, reshaping, and aggregating, Pandas enables streamlined and robust data exploration. Another key strength of Pandas is its ability to import data from various sources such as CSV, Excel, SQL, and JSON. Once loaded, users can perform descriptive statistics, sorting, indexing, merging, and visualization directly on the DataFrame. Whether working on large datasets or real-time data pipelines, Pandas is foundational for any data analyst or scientist aiming for clean, insightful, and reproducible data workflows.

10.2 OBJECTIVE

After completing this Unit we will be able to:

- Define Pandas and its core data structures (Series and DataFrame).
- Explain the importance of Pandas in data analysis.
- Create and manipulate Series and DataFrames.
- Demonstrate data import and export techniques.

NOTE**10.3 GETTING STARTED WITH PANDAS: FROM SERIES TO DATAFRAMES**

Pandas provides powerful tools like Series and DataFrames for handling structured data with ease and flexibility. Series represent one-dimensional labeled arrays, while DataFrames offer two-dimensional, tabular data structures. Together, they form the foundation for efficient data manipulation and analysis in Python.

1. Understanding Pandas Series**2. Creating DataFrames****3. Accessing Data****4. Loading External Data****5. Analyzing Data**

- 1. Understanding Pandas Series:** A Series is a one-dimensional labeled array capable of holding any data type. It acts like a column in a spreadsheet or a simple list with labels.

Example:

```
import pandas as pd

# Creating a basic Series from a list
data = [10, 20, 30, 40]
series = pd.Series(data)
print("Basic Series:")
print(series)

# Creating a Series with custom index labels
custom_series = pd.Series([100, 200, 300], index=['a', 'b', 'c'])
print("\nSeries with custom index:")
print(custom_series)

# Accessing elements in the Series
print("\nAccess element at index 'b':", custom_series['b'])

# Performing operations on Series
print("\nMultiply each element by 2:")
print(custom_series * 2)
```

Output

```
Basic Series:
```

NOTE

```
0  10
1  20
2  30
3  40
dtype: int64
```

Series with custom index:

```
a  100
b  200
c  300
dtype: int64
```

Access element at index 'b': 200

Multiply each element by 2:

```
a  200
b  400
c  600
dtype: int64
```

- 2. Creating DataFrames:** A DataFrame is a two-dimensional labeled data structure, similar to a table. It can be created using dictionaries, lists, or NumPy arrays.

Example:

```
import pandas as pd

# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'London', 'Paris']
}

df = pd.DataFrame(data)

# Display the DataFrame
print("DataFrame from dictionary:")
print(df)

# Creating a DataFrame from a list of lists with column names
data_list = [
```

DATA MANIPULATION WITH PANDAS

NOTE

```
['Tom', 28, 'Berlin'],
['Jerry', 22, 'Tokyo']
]
df2 = pd.DataFrame(data_list, columns=['Name', 'Age', 'City'])

print("\nDataFrame from list of lists:")
print(df2)

# Creating an empty DataFrame and then adding data
df_empty = pd.DataFrame(columns=['Name', 'Age', 'City'])
df_empty.loc[0] = ['Sam', 26, 'Toronto']
df_empty.loc[1] = ['Lily', 31, 'Sydney']

print("\nEmpty DataFrame filled manually:")
print(df_empty)
```

Output:

DataFrame from dictionary:

	Name	Age	City
0	Alice	25	New York
1	Bob	30	London
2	Charlie	35	Paris

DataFrame from list of lists:

	Name	Age	City
0	Tom	28	Berlin
1	Jerry	22	Tokyo

Empty DataFrame filled manually:

	Name	Age	City
0	Sam	26	Toronto
1	Lily	31	Sydney

- 3. Accessing Data:** Pandas offers powerful data access methods like **.loc[]**, **.iloc[]**, and column indexing. Use **.loc** for label-based access and **.iloc** for position-based access.

Example:

```
import pandas as pd

# Sample DataFrame
data = {
```

NOTE

```

'Name': ['Alice', 'Bob', 'Charlie', 'David'],
'Age': [25, 30, 35, 28],
'City': ['New York', 'London', 'Paris', 'Berlin']
}

df = pd.DataFrame(data)

# Display entire DataFrame
print("Full DataFrame:")
print(df)

# Accessing a single column
print("\nAccessing the 'Name' column:")
print(df['Name'])

# Accessing multiple columns
print("\nAccessing 'Name' and 'City' columns:")
print(df[['Name', 'City']])

# Accessing a row by index
print("\nAccessing the first row (index 0):")
print(df.loc[0])

# Accessing a value using row and column
print("\nAccessing value at row 2, column 'City':")
print(df.loc[2, 'City'])

# Accessing rows with condition
print("\nAccessing rows where Age > 28:")
print(df[df['Age'] > 28])

```

Output:

```

Full DataFrame:
   Name  Age  City
0  Alice   25 New York
1   Bob   30  London
2 Charlie   35   Paris
3  David   28   Berlin

```

```

Accessing the 'Name' column:
0  Alice
1   Bob
2 Charlie

```

NOTE

```
3 David
Name: Name, dtype: object
```

```
...
```

Accessing rows where Age > 28:

```
   Name  Age  City
1   Bob   30 London
2  Charlie  35  Paris
```

- 4. Loading External Data:** Pandas can import data from CSV, Excel, JSON, SQL, and more formats. This makes it ideal for real-world data processing workflows.

Example:

```
import pandas as pd

# Load a CSV file into a DataFrame
df = pd.read_csv('sample_data.csv') # Replace with your actual file name or
path

# Display the first 5 rows
print("First 5 rows of the dataset:")
print(df.head())

# Display basic information
print("\nDataset Info:")
print(df.info())

# Display summary statistics
print("\nSummary Statistics:")
print(df.describe())
```

- 5. Analyzing Data:** With built-in functions like `.describe()`, `.head()`, `.info()`, and filtering tools, analysis becomes simple. You can quickly understand data distributions and summary statistics.

Example:

```
import pandas as pd

# Load dataset (use any CSV file)
```

NOTE

```
df = pd.read_csv('sample_data.csv') # Replace with your own file

# 1. Show first few rows
print("Head of the dataset:")
print(df.head())

# 2. Summary statistics
print("\nSummary Statistics:")
print(df.describe())

# 3. Data types of columns
print("\nData Types:")
print(df.dtypes)

# 4. Count missing values
print("\nMissing Values per Column:")
print(df.isnull().sum())

# 5. Frequency count of a specific column (e.g., 'Category')
if 'Category' in df.columns:
    print("\nValue Counts for 'Category':")
    print(df['Category'].value_counts())

# 6. Group by and mean (if a numerical column like 'Sales' exists)
if 'Region' in df.columns and 'Sales' in df.columns:
    print("\nAverage Sales by Region:")
    print(df.groupby('Region')['Sales'].mean())
```

10.4 DATA ANALYSIS MADE EASY: IMPORTING AND EXPLORING DATA WITH PANDAS

Pandas is a high-level Python library designed for data manipulation and analysis. It offers powerful data structures such as Series and DataFrames, which make it incredibly easy to import, clean, and explore structured datasets. Whether you're handling financial records, sensor data, or survey

NOTE

responses, Pandas enables a streamlined and readable approach to data analysis.

1. Importing Data with Pandas

The first step in any analysis is data acquisition. Pandas provides simple functions like `pd.read_csv()`, `pd.read_excel()`, and `pd.read_json()` to read files from various formats into DataFrames. This makes it easy to load real-world datasets stored locally or hosted online.

```
import pandas as pd
df = pd.read_csv('data.csv') # Load a CSV file into a DataFrame
```

You can also add arguments like `delimiter`, `header`, or `index_col` to control how the data is parsed.

2. Exploring Data Efficiently

Once data is loaded, Pandas provides a set of methods for quickly exploring the structure:

- `df.head()` – Views the first few rows.
- `df.info()` – Displays column names, types, and missing values.
- `df.describe()` – Returns statistical summaries like mean, median, min, and max.
- `df.columns`, `df.shape` – Help check structure and dimensions.

```
print(df.head())      # View top 5 rows
print(df.describe())  # Get numeric stats
print(df.isnull().sum()) # Count missing values per column
```

3. Analyzing Relationships and Patterns

Pandas allows easy **grouping**, **filtering**, and **aggregating**:

```
# Example: Average sales by region
df.groupby('Region')['Sales'].mean()
```

You can also filter rows with logical conditions:

```
# Filter rows where Sales > 5000
high_sales = df[df['Sales'] > 5000]
```

4. Why Pandas Makes Data Analysis Easy

Pandas abstracts the complexity of working with raw data. You don't need to write loops for common tasks—everything from filling missing values to merging datasets is one line of code. It integrates well with NumPy, Matplotlib, and other scientific libraries, making it a cornerstone of the data science workflow.

NOTE**Activity 10.1**

Create a Pandas DataFrame with columns: Name, Age, Marks.

Add at least 5 rows of data and display:

First 3 rows

Data types of each column

.....

.....

.....

☑ Check Your Progress 10.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. What Pandas does best is simplify complex data tasks.

.....

.....

.....

Q2. What makes Pandas powerful is its clean data handling.

.....

.....

.....

10.5 SUMMARY

Pandas is a powerful Python library designed for efficient data manipulation and analysis. It introduces two core data structures—Series and DataFrame—that allow users to handle one-dimensional and two-dimensional data, respectively. With Pandas, importing data from various sources like CSV, Excel, or SQL becomes seamless, enabling a streamlined workflow for data-driven tasks. The library supports powerful indexing, filtering, and transformation operations, making it easier to clean and structure data for analysis.

Moreover, Pandas provides built-in methods for descriptive statistics, grouping, merging, and time series analysis. Its intuitive syntax bridges the gap between raw data and actionable insights, especially for beginners in data science and analytics. Whether you're summarizing trends, cleaning missing values, or preparing data for machine learning, Pandas offers a user-friendly

NOTE

yet robust toolkit. Its compatibility with NumPy, Matplotlib, and scikit-learn also makes it a cornerstone of the Python data ecosystem. Ultimately, Pandas transforms raw datasets into structured formats, enabling fast, accurate, and reproducible data analysis.

10.6 GLOSSARY

- **Pandas** – A Python library used for data manipulation, analysis, and handling structured datasets.
- **Series** – A one-dimensional labeled array capable of holding data of any type.
- **DataFrame** – A two-dimensional tabular data structure with labeled rows and columns.
- **Index** – A label used to identify and access rows in a Series or DataFrame.
- **Column** – A vertical collection of related data values in a DataFrame.
- **Row** – A horizontal collection of data representing a single record in a DataFrame.
- **CSV File** – A Comma-Separated Values file commonly used to store and exchange tabular data.
- **Data Import** – The process of loading data from external sources into a Pandas DataFrame.
- **Data Export** – The process of saving data from a DataFrame to external file formats.
- **Label-based Indexing** – A method of selecting data using row and column labels.

10.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 10.1**Answer 1:**

Pandas is a powerful Python library that excels at simplifying complex data manipulation and analysis tasks. With just a few lines of code, it can handle large datasets, perform intricate operations like grouping, merging, filtering, and reshaping data that would otherwise require lengthy manual coding or spreadsheet effort. Whether you're dealing with messy real-world data or structured records, Pandas provides built-in functions that make the process intuitive and efficient.

For instance, cleaning missing values, computing statistical summaries, or transforming time series data becomes a straightforward task using Pandas' methods like `.dropna()`, `.describe()`, and `.resample()`. Its seamless integration with data import functions (like `read_csv()`, `read_excel()`) allows users to load and begin analyzing external data instantly.

By providing a high-level interface for working with structured data, Pandas abstracts much of the low-level logic—allowing data scientists, analysts, and developers to focus on insights rather than implementation. This makes

Pandas a go-to tool in data science pipelines, especially when quick, reliable, and readable solutions are needed.

Answer 2:

Pandas stands out in the data science ecosystem because of its ability to manage, clean, and prepare data efficiently. Real-world datasets often come with inconsistencies such as missing values, duplicate entries, or improperly formatted data. Pandas offers powerful and intuitive tools—like `.fillna()`, `.dropna()`, `.duplicated()`, and `.replace()`—that streamline the cleaning process with minimal code.

This clean data handling capability is crucial for reliable analysis. Pandas not only allows easy detection of issues in datasets but also provides flexible ways to correct or reshape them, making the data analysis pipeline more robust and less error-prone. Whether you're dealing with numerical data, categorical values, or timestamps, Pandas ensures your data is clean, consistent, and ready for processing.

Ultimately, Pandas empowers users to move from raw data to actionable insights faster, making it an essential library for anyone working with structured data in Python.

NOTE

10.8 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/Use/UnitUrls?unitId=1271	

10.9 REFERENCE

- <https://pandas.pydata.org/docs/>
- <https://realpython.com/learning-paths/pandas-data-analysis/>

10.10 EXERCISE

Part 1

A. Multiple choice question

1. What is a Pandas Series?

- A) A 2D labeled data structure
- B) A single column of data with labels
- C) A list of lists
- D) A function for plotting data

NOTE

2. Which function is used to read a CSV file into a Pandas DataFrame?

- A) pandas.load_csv()
- B) pandas.read_file()
- C) pandas.read_csv()
- D) pandas.import_csv()

3. What is the default axis for DataFrame.sum() when applied?

- A) Axis 0 (column-wise)
- B) Axis 1 (row-wise)
- C) Both axes simultaneously
- D) There is no default axis

4. Which method would you use to display the first 5 rows of a DataFrame?

- A) df.tail()
- B) df.first(5)
- C) df.show()
- D) df.head()

5. What distinguishes a DataFrame from a Series in Pandas?

- A) DataFrame is immutable
- B) Series is 2D while DataFrame is 1D
- C) DataFrame can hold multiple columns
- D) Series cannot contain data

Answer with Explanations:

1. B) A single column of data with labels

Explanation: A Series is a one-dimensional labeled array that represents a single column of data.

2. C) pandas.read_csv()

Explanation: read_csv() is the standard function to load a CSV into a DataFrame.

3. A) Axis 0 (column-wise)

Explanation: By default, sum() operates column-wise (down the rows).

4. D) df.head()

Explanation: df.head() displays the first five rows of a DataFrame.

5. C) DataFrame can hold multiple columns

Explanation: A DataFrame can contain multiple columns, unlike a Series which is single-column.

NOTE**B. Short answer type question**

1. What is the key difference between a Pandas Series and a DataFrame?
2. Name two common file formats supported by Pandas for data import.
3. Which Pandas method is used to get basic statistical summaries of a DataFrame?

Part 2**C. Long answer type question (200 words)**

1. Explain the structure and use of Series and DataFrames in Pandas. How do they differ and complement each other?
2. Describe how Pandas handles data import from external sources. What are the key functions and considerations?
3. How does Pandas simplify data analysis, and what are some of the common operations analysts perform with it?

Part 3**D. Long Answer Type Questions (300 words)**

1. Explain the need for Pandas in data analysis. Compare Series and DataFrame structures with examples.
2. Describe DataFrame creation using lists, dictionaries, and NumPy arrays. Explain indexing and selection operations.
3. Discuss the importance of data manipulation and the role of Pandas in handling tabular data efficiently.

UNIT 11

DATA CLEANING IN PANDAS

NOTE

UNIT STRUCTURE

- 11.1 Introduction
- 11.2 Objective
- 11.3 Pandas for Clean Data: Strategies to Handle Missing, Messy, and Duplicate Entries
- 11.4 Effective Data Cleaning with Pandas: From Nulls to Neat DataFrames
- 11.5 Summary
- 11.6 Glossary
- 11.7 Answers to Check Your Progress.
- 11.8 SUGGESTED READINGS – OER -OER
- 11.9 Reference
- 11.10 Exercise

11.1 INTRODUCTION

In the world of data analysis, the quality of your insights is only as good as the quality of your data. Real-world datasets often contain missing values, inconsistent formatting, and duplicate records, which can distort analysis and lead to incorrect conclusions. Pandas, a powerful Python library for data manipulation, offers robust tools to clean and prepare your data efficiently and effectively.

Handling missing values is one of the most common challenges in data preprocessing. Whether it's filling them with meaningful statistics like mean or median, or dropping them altogether, Pandas provides versatile functions like `fillna()`, `dropna()`, and `interpolate()` to manage them. Formatting issues—such as inconsistent date formats, capitalization, or data types—can be resolved using string methods and type conversion functions available in Pandas.

Another critical step is removing duplicate records that may skew your analysis. With functions like `drop_duplicates()` and `duplicated()`, you can identify and clean up repeated entries easily. Together, these operations form the backbone of data cleaning in Pandas, ensuring that your datasets are not only tidy but also ready for reliable analysis and modeling.

11.2 OBJECTIVE

After completing this Unit we will be able to:

- Identify missing and inconsistent data.
- Apply techniques for data cleaning and preprocessing.

NOTE

- Transform data using filtering, replacing, and formatting methods.
- Evaluate data quality after cleaning processes.

11.3 PANDAS FOR CLEAN DATA: STRATEGIES TO HANDLE MISSING, MESSY, AND DUPLICATE ENTRIES

Data cleaning is a crucial step in any data analysis or machine learning workflow. Pandas, a powerful Python library, provides intuitive and efficient tools to clean and prepare datasets for accurate analysis. Whether the data is incomplete, inconsistently formatted, or duplicated, Pandas offers structured solutions to transform it into a usable form.

- 1. Handling Missing Values:** Missing data can arise from various sources such as data entry errors, sensor failures, or merging mismatches. Pandas makes it easy to identify missing values using functions like `.isnull()` or `.notnull()`. Once detected, you can choose to remove them using `.dropna()` or fill them with specific values (mean, median, constant) using `.fillna()`. This flexibility allows for precise control over how you deal with incomplete data depending on the context.

Code example demonstrating various ways to handle missing values in a Pandas DataFrame using functions like `.isnull()`, `.fillna()`, and `.dropna()`.

```
import pandas as pd
import numpy as np

# Sample DataFrame with missing values
data = {
    'Name': ['Alice', 'Bob', 'Charlie', None],
    'Age': [25, np.nan, 30, 22],
    'Salary': [50000, 60000, None, 52000]
}

df = pd.DataFrame(data)

print("Original DataFrame with Missing Values:")
print(df)

# 1. Detect missing values
print("\nMissing value check (True = Missing):")
print(df.isnull())

# 2. Fill missing values
df_filled = df.fillna({
    'Name': 'Unknown',
    'Age': df['Age'].mean(),      # Fill with mean age
    'Salary': df['Salary'].median() # Fill with median salary
})
```

NOTE

```
print("\nDataFrame after filling missing values:")
print(df_filled)

# 3. Drop rows with any missing value
df_dropped = df.dropna()

print("\nDataFrame after dropping rows with missing values:")
print(df_dropped)
```

Output Summary:**Original DataFrame:**

	Name	Age	Salary
0	Alice	25.0	50000.0
1	Bob	NaN	60000.0
2	Charlie	30.0	NaN
3	None	22.0	52000.0

After Filling Missing Values:

	Name	Age	Salary
0	Alice	25.0	50000.0
1	Bob	25.7	60000.0
2	Charlie	30.0	52000.0
3	Unknown	22.0	52000.0

After Dropping Missing Values:

	Name	Age	Salary
0	Alice	25.0	50000.0

- 2. Resolving Formatting Issues:** Real-world datasets often come with formatting inconsistencies, such as mixed date formats, inconsistent casing in strings, or incorrect data types. With Pandas, you can use functions like `.str.lower()`, `.astype()`, and `.to_datetime()` to clean and standardize formats. These operations ensure that all values in a column follow a consistent structure, making analysis smoother and more accurate.

Here's a code example to resolve common formatting issues in a Pandas DataFrame, such as:

- Stripping whitespace

NOTE

- Converting to lowercase/uppercase
- Converting data types (e.g., strings to dates or numbers)

Example:

```
import pandas as pd

# Sample DataFrame with formatting issues
data = {
    'Name': [' Alice ', 'BOB', ' Charlie'],
    'Date_of_Birth': ['1990/01/01', '1992-05-10', 'March 15, 1988'],
    'Salary': [' 50000', '60000 ', ' 55000 ']
}

df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)

# Cleaning and formatting
df['Name'] = df['Name'].str.strip().str.title() # Remove whitespace and title case
df['Salary'] = df['Salary'].str.strip().astype(int) # Remove whitespace and convert to int
df['Date_of_Birth'] = pd.to_datetime(df['Date_of_Birth']) # Convert to datetime

print("\nFormatted DataFrame:")
print(df)
```

Output:

Before Cleaning:

	Name	Date_of_Birth	Salary
0	Alice	1990/01/01	50000
1	BOB	1992-05-10	60000
2	Charlie	March 15, 1988	55000

After Cleaning:

	Name	Date_of_Birth	Salary
0	Alice	1990-01-01	50000
1	Bob	1992-05-10	60000
2	Charlie	1988-03-15	55000

- 3. Removing Duplicate Entries:** Duplicate records skew data interpretation and must be identified and handled. Pandas provides

NOTE

.duplicated() to detect and .drop_duplicates() to remove repeated rows. You can customize this by specifying which columns to check for duplicates and whether to keep the first or last occurrence. This ensures the dataset maintains unique and meaningful records.

By mastering these core strategies with Pandas, data analysts and scientists can confidently prepare their data for deeper analysis. Clean data not only leads to better insights but also enhances the reliability and performance of models built on top of it.

effective example of how to remove duplicate entries in a DataFrame using Pandas:

```
import pandas as pd
# Sample DataFrame with duplicate entries
data = {
    'Name': ['Alice', 'Bob', 'Alice', 'David', 'Bob'],
    'Age': [25, 30, 25, 40, 30],
    'City': ['New York', 'Los Angeles', 'New York', 'Chicago', 'Los Angeles']
}

df = pd.DataFrame(data)

print("Original DataFrame with Duplicates:")
print(df)

# Remove duplicate rows
df_cleaned = df.drop_duplicates()

print("\nDataFrame after Removing Duplicates:")
print(df_cleaned)
```

Output:**Original DataFrame with Duplicates:**

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Alice	25	New York
3	David	40	Chicago
4	Bob	30	Los Angeles

NOTE**DataFrame after Removing Duplicates:**

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
3	David	40	Chicago

Tip: You can also use `df.drop_duplicates(subset=['Name'])` to remove duplicates based on specific columns only.

11.4 EFFECTIVE DATA CLEANING WITH PANDAS: FROM NULLS TO NEAT DATAFRAMES

In real-world datasets, messy or incomplete data is common, and data cleaning is essential to prepare this data for analysis. Pandas provides powerful and intuitive tools to clean data efficiently, making it one of the most used libraries in data preprocessing tasks. The process includes handling missing values, formatting issues, and duplicates — all of which can lead to inaccurate analysis if left untreated.

Handling missing values is the first major step. Pandas offers functions like `.isnull()`, `.fillna()`, and `.dropna()` to identify and manage nulls. Depending on the context, you may fill missing values with the mean, median, mode, or a custom value, or remove rows or columns entirely. This ensures the continuity of your dataset while maintaining statistical relevance.

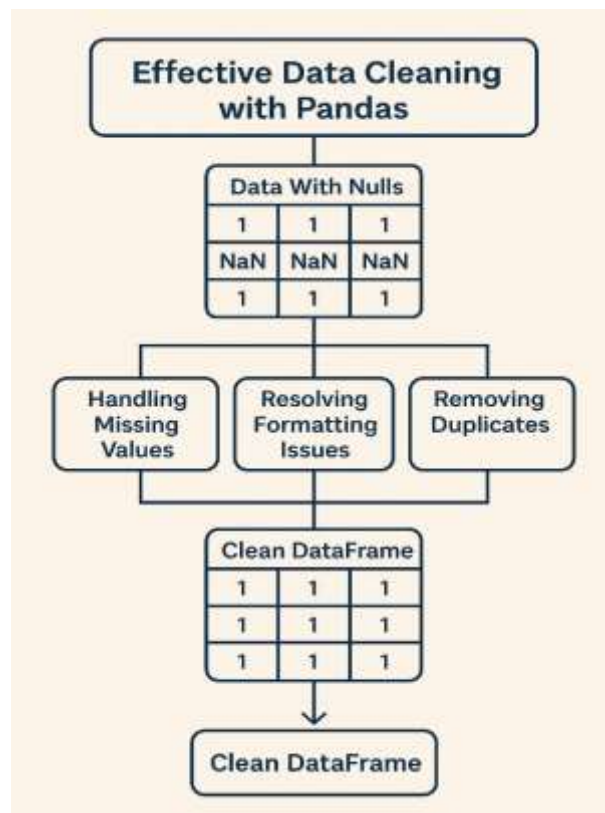


Figure 11.1: Data Cleaning

NOTE

Next, formatting issues such as inconsistent capitalization, whitespace, or incorrect data types can be resolved using methods like `.str.strip()`, `.astype()`, or `.replace()`. These ensure data consistency, especially when dealing with categorical text or numeric fields. Lastly, duplicate entries are detected using `.duplicated()` and removed with `.drop_duplicates()` to avoid redundancy and inflated statistics. By applying these steps, you can transform raw data into clean, structured, and analysis-ready DataFrames, enabling deeper and more accurate insights.

**Activity 11.1**

Create a DataFrame with some missing values.

Perform the following:

Detect missing values

Replace missing values with the column mean

Drop rows with more than 1 missing value

.....

☑ Check Your Progress: 11.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. What Pandas cleans is not just data but confusion, making analysis clear and actionable.

.....

Q2. What makes Pandas essential is its ability to turn messy datasets into meaningful insights with just a few lines of code.

.....

11.5 SUMMARY

Data cleaning is a critical step in any data analysis process, and Pandas provides powerful, intuitive tools to handle it efficiently. Missing values, often represented as NaN, can be identified and treated using methods like `fillna()`, `dropna()`, or interpolation techniques depending on the dataset and context. These tools help ensure that missing data does not mislead analysis or modeling efforts.

NOTE

Formatting issues such as inconsistent data types or incorrect date formats are easily managed in Pandas using functions like `astype()` for type conversions and `to_datetime()` for parsing dates. Ensuring proper formatting allows for accurate sorting, filtering, and comparisons in the data.

Pandas also helps detect and remove duplicate entries using `duplicated()` and `drop_duplicates()`, which prevent skewed results during aggregation or analysis. By addressing these common data issues, Pandas transforms messy, unreliable datasets into clean, structured DataFrames that are ready for meaningful analysis. This makes it a vital tool in every data professional's toolkit.

11.6 GLOSSARY

- **Data Cleaning** – The process of identifying and correcting errors, inconsistencies, and missing values in a dataset.
- **Missing Values** – Data entries that are absent or represented as null values.
- **Null Value** – A placeholder indicating that no valid data is available for a particular field.
- **Duplicate Records** – Repeated rows in a dataset that may affect data accuracy.
- **Data Imputation** – The process of replacing missing values with estimated or calculated values.
- **Outlier** – A data value that differs significantly from the rest of the dataset.
- **Data Type Conversion** – Changing the format of data from one type to another, such as integer to string.
- **Filtering** – Selecting rows or columns that satisfy specified conditions.
- **Data Validation** – The process of verifying that data is accurate, complete, and consistent.
- **Data Consistency** – The uniformity and reliability of data across a dataset.

11.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 10.1**Answer 1:**

The quote *"What Pandas cleans is not just data—but confusion, making analysis clear and actionable."* highlights the practical strength of Pandas in real-world data science. In data analysis, raw datasets often contain missing values, inconsistent formats, or duplicate entries that can lead to misleading results or analysis errors. Pandas provides a suite of functions like `.dropna()`, `.fillna()`, `.duplicated()`, and `.drop_duplicates()`—that make it easy to clean and prepare data effectively.

By resolving these issues, Pandas helps analysts focus on interpreting meaningful insights rather than getting stuck in data wrangling. Clean data is

critical for building accurate models and drawing reliable conclusions, and Pandas simplifies that journey by transforming chaos into clarity.

Answer 2:

The quote "What makes Pandas essential is its ability to turn messy datasets into meaningful insights with just a few lines of code." emphasizes Pandas' core strength—efficient and intuitive data manipulation. Whether dealing with missing values, inconsistent formatting, or duplicate records, Pandas offers simple yet powerful tools like `fillna()`, `astype()`, `drop_duplicates()`, and `value_counts()` that allow users to clean, transform, and explore data quickly.

Instead of writing lengthy loops or complex logic, Pandas enables data scientists and analysts to perform operations in a concise and readable manner. This makes it ideal not only for cleaning data but also for performing tasks such as grouping, filtering, and summarizing data. As a result, even large and unstructured datasets become easier to analyze, allowing users to generate actionable insights efficiently and reliably.

NOTE

11.8 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/Use/UnitUrls?unitId=1272	

11.9 REFERENCE

- https://pandas.pydata.org/docs/user_guide/missing_data.html
- <https://realpython.com/python-data-cleaning-numpy-pandas/>
- <https://towardsdatascience.com/the-ultimate-guide-to-data-cleaning-3969843991d4>

11.10 EXERCISE

Part 1

A. Multiple choice question

1. Which of the following Pandas functions is used to remove missing values from a DataFrame?

- A. `fillna()`
- B. `dropna()`
- C. `isnull()`
- D. `replace()`

NOTE**2. What does the duplicated() function in Pandas return?**

- A. A list of unique values
- B. A DataFrame without duplicates
- C. A Boolean Series indicating duplicate rows
- D. The number of missing values

3. Which method is commonly used to replace missing values with the mean in a DataFrame column?

- A. replace()
- B. interpolate()
- C. fillna(df['column'].mean ())
- D. dropna()

4. What is the purpose of the astype() function in Pandas?

- A. To remove missing values
- B. To change the data type of a column
- C. To detect outliers
- D. To sort the DataFrame

5. If you want to identify formatting issues like extra spaces in a text column, which Pandas method would you typically use?

- A. strip()
- B. astype()
- C. isnull()
- D. drop_duplicates()

Answer with Explanations:**1. B) dropna()**

Explanation: dropna() removes rows or columns containing missing (NaN) values.

2. C) A Boolean Series indicating duplicate rows

Explanation: duplicated() returns a Boolean Series marking rows that are duplicates.

3. C) fillna(df['column'].mean())

Explanation: fillna(df['column'].mean()) replaces missing values with the column's mean.

4. B) To change the data type of a column

Explanation: astype() converts a column to a specified data type.

5. A) strip()

Explanation: strip() removes leading/trailing spaces, helping identify formatting issues.

B. Short answer type question

1. What is the purpose of the fillna() function in Pandas?
2. How does Pandas help identify duplicate rows in a dataset?
3. What method would you use to convert a column with improperly formatted numbers (as strings) into a numeric type?

Part 2**C. Long answer type question (200 words)**

1. Explain how Pandas handles missing data and describe various strategies for dealing with missing values in a dataset.
2. Discuss how formatting issues can impact data analysis and describe how Pandas helps to resolve such issues.
3. What are duplicate values in a dataset, and how can Pandas be used to identify and handle them effectively?

Part 3**D. Long Answer Type Questions (300 words)**

1. Explain the concept of missing data. Describe techniques such as detection, handling, replacement, and removal.
2. Discuss data cleaning operations such as handling duplicates, formatting columns, and correcting inconsistencies.
3. Explain the importance of clean data in analytics and how Pandas simplifies the data preprocessing pipeline.

NOTE

UNIT 12

DATA ANALYSIS WITH PANDAS

NOTE

UNIT STRUCTURE

- 12.1 Introduction
- 12.2 Objective
- 12.3 Discovering Relationships: Correlation Analysis with Pandas
- 12.4 Visualizing Insights: Data Trends and Patterns Using Pandas
- 12.5 From Data to Discovery: Exploring Correlations and Charts with Pandas
- 12.6 Summary
- 12.7 Glossary
- 12.8 Answers to check your progress
- 12.9 SUGGESTED READINGS – OER-OER
- 12.10 Reference
- 12.11 Exercise

12.1 INTRODUCTION

Pandas is a powerful data manipulation library in Python that allows analysts to perform deep, insightful data analysis with ease. When working with real-world datasets, discovering relationships between variables is crucial—and this is where correlation analysis becomes essential. Pandas offers built-in methods such as `.corr()` to calculate statistical relationships between columns, helping analysts uncover hidden patterns in the data.

Beyond numeric computation, visualization transforms raw data into meaningful stories. Using Pandas alongside visualization libraries like Matplotlib and Seaborn makes it possible to create intuitive charts—like heatmaps, scatter plots, and line graphs—that bring data to life.

Together, correlation metrics and visual tools enable users to move beyond spreadsheets and dive into pattern recognition, trend discovery, and evidence-backed decision making. This combined approach simplifies complex analysis and enhances interpretability for both technical and non-technical audiences.

12.2 OBJECTIVE

After completing this Unit we will be able to:

- Analyze datasets using grouping and aggregation techniques.
- Apply statistical analysis using Pandas functions.
- Interpret trends and patterns in data.
- Evaluate results to support decision-making.

NOTE**12.3 DISCOVERING RELATIONSHIPS: CORRELATION ANALYSIS WITH PANDAS**

Correlation analysis with Pandas is a powerful technique for discovering relationships between numerical variables in a dataset. It helps analysts understand how one variable may change in response to another. Using methods like Pearson, Kendall, and Spearman correlations, Pandas provides built-in functions to quantify the strength and direction of these relationships.

The `.corr()` method in Pandas calculates pairwise correlation between columns in a DataFrame. For instance, a correlation coefficient close to 1 implies a strong positive relationship, while one near -1 suggests a strong negative relationship. A value around 0 indicates no linear relationship.

Correlation analysis is commonly visualized using heatmaps, making it easier to interpret large sets of relationships quickly. Libraries like Seaborn or Matplotlib are often used alongside Pandas for this purpose.

By identifying highly correlated features, analysts can detect redundancy in data, which is useful in feature selection for machine learning. Conversely, spotting weak or no correlation may highlight unrelated variables or noisy data.

Overall, Pandas correlation tools offer a fast, efficient way to uncover patterns, ensure data quality, and guide deeper exploratory data analysis or predictive modeling efforts.

Python code example for Discovering Relationships: Correlation Analysis with Pandas using `.corr()` and a heatmap for visualization:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Sample dataset
data = {
    'Study_Hours': [2, 3, 4, 5, 6, 7, 8],
    'Test_Score': [55, 60, 65, 70, 75, 80, 85],
    'Sleep_Hours': [8, 7, 6, 5, 4, 3, 2]
}

df = pd.DataFrame(data)

# Correlation matrix
correlation_matrix = df.corr()
print("Correlation Matrix:")
print(correlation_matrix)
```

NOTE

```
# Heatmap for visualization
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap")
plt.show()
```

Explanation:

- df.corr() computes the correlation between all numeric columns.
- sns.heatmap() displays a color-coded matrix for easy interpretation.
- In this example, Study_Hours and Test_Score will show strong positive correlation, while Sleep_Hours will show a negative relationship with Study_Hours and Test_Score.

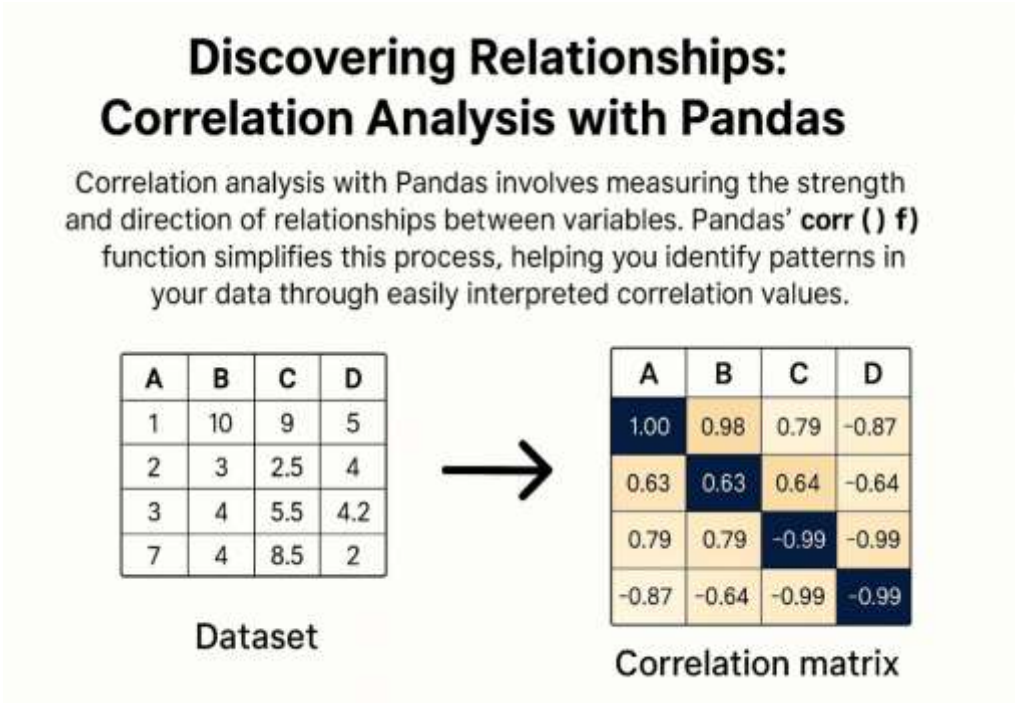


Figure 12.1: Pandas Correlation Analysis

12.4 VISUALIZING INSIGHTS: DATA TRENDS AND PATTERNS USING PANDAS

Pandas, when combined with visualization libraries like Matplotlib and Seaborn, becomes a powerful tool for uncovering trends, patterns, and relationships in data. Through simple syntax and built-in compatibility with these libraries, Pandas allows users to generate meaningful visualizations directly from DataFrames. This makes it ideal for quick exploratory data analysis (EDA) and report-ready visual storytelling.

Using `plot()`, `hist()`, `boxplot()`, or passing data to Seaborn functions like `sns.lineplot()` or `sns.barplot()`, users can visualize time series, category distributions, outliers, and correlations. For example, plotting sales over time

NOTE

reveals trends, while a boxplot of student scores may highlight variability and outliers.

These visualizations help analysts quickly identify:

- Outliers that might skew analysis
- Patterns such as seasonality or growth
- Relationships between features (e.g., income vs. spending)

By turning raw numbers into clear visuals, Pandas empowers users to transform data exploration into data explanation. It enables stakeholders, even without technical background, to grasp the story behind the numbers. In short, Pandas visualization isn't just about making charts—it's about making data meaningful.

Example: Sales Trend Analysis**1. Import Libraries**

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Create Sample Data

```
# Simulated monthly sales data
data = {
    'Month': pd.date_range(start='2024-01-01', periods=12, freq='M'),
    'Sales': [200, 220, 210, 250, 265, 270, 300, 310, 305, 330, 350, 360]
}
df = pd.DataFrame(data)
```

3. Plot Line Chart for Trend

```
plt.figure(figsize=(10, 5))
sns.lineplot(x='Month', y='Sales', data=df, marker='o')
plt.title('Monthly Sales Trend')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.grid(True)
plt.tight_layout()
plt.show()
```

What This Shows:

- The line chart clearly visualizes the upward trend in sales.
- You can quickly spot growth patterns and use this insight for forecasting.
- Adding marker='o' highlights each month's value clearly.

12.5 FROM DATA TO DISCOVERY: EXPLORING CORRELATIONS AND CHARTS WITH PANDAS

NOTE

In the world of data analysis, finding meaningful patterns often starts with identifying relationships between variables. Pandas, combined with visualization libraries like Matplotlib and Seaborn, empowers users to explore data from raw form to actionable insight. Correlation analysis is a powerful statistical tool used to determine how closely related two or more variables are. Pandas offers a simple `.corr()` method to compute correlation matrices, helping analysts uncover trends, associations, or potential causal relationships hidden within the data.

Beyond numerical insight, visualization makes patterns intuitive. Pandas integrates seamlessly with Matplotlib to create line charts, bar plots, and scatter plots, turning raw numbers into digestible visuals. For example, a heatmap generated from a correlation matrix can highlight strong or weak correlations using colors, simplifying interpretation. This is especially useful in large datasets where relationships are not immediately obvious. By visualizing these relationships, analysts can better guide decisions, feature selections for models, or highlight unusual behavior in data.

Ultimately, Pandas serves as a bridge between data collection and meaningful discovery. Whether you're cleaning, analyzing, or visualizing, it simplifies the entire process—transforming rows and columns into rich, data-driven stories.'



Activity 12.1

TITLE OF ACTIVITY: Data Aggregation, Sorting, and Summarization Using Pandas DataFrames

OBJECTIVE: To build practical capabilities in descriptive data analysis by manipulating, sorting, and extracting segmented statistical insights from multi-row tabular datasets using the Pandas library.

MATERIAL NEEDED: A computer or smartphone with a Python execution environment (such as Google Colab, Jupyter Notebook, or VS Code) with the pandas library installed, and a digital study journal.

ACTIVITY: Let us put our data manipulation skills to work by creating and analyzing a structured business dataset. Your task is to construct or import a Pandas DataFrame containing at least 10 rows of data. Ensure your dataset features multiple numerical columns (such as sales figures, age, or salary) along with at least one categorical column (such as department, region, or product type). Once your DataFrame is ready, write a Python script to perform and display the outputs of the following three tasks:

1. Apply the `.describe()` method to generate and display the comprehensive summary statistics table of your numerical columns.
2. Sort your entire dataset by one of your primary numeric columns using appropriate sorting parameters.
3. Group your data based on your categorical column and compute the mathematical mean to compare how the values differ across categories.

NOTE

Ensure your code uses clear print statements to separate and label the output of each operation.

EXPECTATION: You should produce a clean, executable, and error-free Python script using standard Pandas functions (`df.describe()`, `df.sort_values()`, and `df.groupby()`). Your final submission must display the source code alongside the neatly rendered tabular results for all three data manipulation steps.

REFLECTION: Look closely at the insights you uncovered after grouping your data. How does looking at a grouped mean differ from analyzing the global mean provided across the entire dataset by `.describe()`? Reflect on why grouping data is a foundational strategy for managers and data analysts when trying to uncover hidden trends, disparities, or performance variations among different operational segments.

☑ Check Your Progress: 12.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. What Pandas reveals through correlations, visualization makes unforgettable.

.....

.....

.....

Q2. What turns raw data into insight is Pandas' power to connect numbers and visuals.

.....

.....

.....

12.6 SUMMARY

Data analysis using Pandas provides a powerful foundation for exploring relationships within datasets through correlation and visualization techniques. Correlation analysis helps identify how variables are related, whether positively, negatively, or not at all, offering essential clues about data behavior and trends. With Pandas, calculating correlation coefficients is both simple and insightful, enabling users to make data-driven assumptions.

In addition to correlation, Pandas supports seamless integration with visualization libraries like Matplotlib and Seaborn to plot trends and patterns effectively. Graphs such as heatmaps, line plots, and scatter plots enhance interpretability and uncover hidden patterns in complex datasets. Together, these tools transform raw data into meaningful visuals, supporting better decision-making and clearer communication of insights.

12.7 GLOSSARY

- **Data Analysis** – The process of examining, organizing, and interpreting data to extract meaningful insights.
- **Descriptive Statistics** – Statistical measures such as mean, median, mode, and standard deviation used to summarize data.
- **Grouping** – Organizing data into categories for analysis using common values.
- **Aggregation** – Combining multiple data values to produce summary statistics.
- **Pivot Table** – A data summarization tool that reorganizes and aggregates information in a tabular format.
- **Correlation** – A statistical measure indicating the strength and direction of the relationship between variables.
- **Sorting** – Arranging data in ascending or descending order based on one or more columns.
- **Filtering** – Selecting specific rows from a dataset based on defined conditions.
- **Data Visualization** – The graphical representation of data using charts, graphs, or plots.
- **Summary Report** – A concise presentation of analyzed data highlighting key findings and trends.

NOTE

12.8 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 10.1

Answer 1:

This quotation emphasizes the dual power of Pandas in data analysis: it not only helps uncover relationships between variables through correlation analysis, but also strengthens understanding by visualizing those relationships. Correlation quantifies how strongly two variables move together, while visualizations like heatmaps or scatter plots make those patterns instantly clear and memorable. Together, they transform raw numbers into meaningful insights.

Answer: 2

This quotation captures how Pandas transforms raw data into actionable insight by bridging numerical relationships and visual interpretation. With features for correlation analysis and plotting, Pandas allows users to not only compute statistical connections but also visualize them clearly. This integration helps analysts move from data to understanding efficiently and intuitively.

NOTE**12.9 SUGGESTED READINGS – OER**

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/User/UnitUrls?unitId=1273	

12.10 REFERENCE

- https://pandas.pydata.org/docs/user_guide/visualization.html
- <https://pandas.pydata.org/docs/>

12.11 EXERCISE**Part 1****A. Multiple choice question**

- What does the .corr() function in Pandas compute**
 - Covariance
 - Cumulative sum
 - Correlation matrix
 - Conditional probability
- Which Pandas-compatible library is most commonly used for plotting visualizations like heatmaps and pairplots?**
 - NumPy
 - Seaborn
 - SciPy
 - Matplotlib only
- What type of correlation does .corr() calculate by default?**
 - Spearman
 - Kendall
 - Pearson
 - None
- Which Pandas function is used to generate basic line plots?**
 - df.show()

NOTE

- B. `df.plot()`
- C. `df.display()`
- D. `df.visualize()`

5. What is a heatmap typically used for in data analysis?

- A. Showing temperature data only
- B. Highlighting missing values
- C. Displaying correlations visually
- D. Sorting categorical data

Answer with Explanations :**1. C) Correlation matrix**

Explanation: `corr()` computes correlations between numerical columns.

2. B) Seaborn

Explanation: Seaborn works well with Pandas and is widely used for heatmaps and pairplots.

3. C) Pearson

Explanation: Pandas uses Pearson correlation by default, measuring linear relationships.

4. B) `df.plot()`

Explanation: `df.plot()` quickly generates basic visualizations like line plots.

5. C) Displaying correlations visually

Explanation: Heatmaps visually show correlation strengths using color intensity.

B. Short answer type question

1. What is the purpose of using the `.corr()` function in Pandas?
2. Name two popular Python libraries used with Pandas for data visualization.
3. How can a heatmap be helpful in data analysis?

Part 2**C. Long answer type question (200 words)**

1. Explain how correlation analysis works in Pandas and how it helps in understanding data relationships.
2. Discuss how visualization using Pandas, Matplotlib, and Seaborn enhances understanding of data patterns.
3. Describe a complete workflow using Pandas for correlation and visualization, including example use cases.

NOTE

Part 3

D. Long Answer Type Questions: (300 words)

1. Discuss the process of exploring datasets using Pandas. Explain summary statistics and data visualization integration.
2. Describe grouping, aggregation, merging, and joining of data. Explain their importance in relational data analysis.
3. Explain sorting, filtering, and transformation methods in Pandas used for analytical tasks.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AICTE GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | UCI | Member of: AMU | Joint Committee: JUGC | DBR | AICTE | Recognized by: UGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

3. How many hours did you need for studying the units? :

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

BLOCK V	PYTHON MATPLOTLIB	163-186
Unit 13	Introduction to Matplotlib	165
Unit 14	Advanced Matplotlib	175



This block focuses on data visualization using Matplotlib, a widely used Python library for creating graphs and charts. Students will learn how to plot data, customize visual elements, and present information effectively through different types of charts such as line plots, bar charts, histograms, and scatter plots. The block also covers advanced plotting techniques that improve readability and presentation quality, including subplotting, styling, annotations, and formatting. By mastering Matplotlib, students will be able to visualize data clearly and communicate insights in a professional and visually appealing manner.

UNIT 13

INTRODUCTION TO MATPLOTLIB

NOTE

UNIT STRUCTURE

- 13.1 Introduction**
- 13.2 Objective**
- 13.3 Getting Started with Matplotlib: Mastering Pyplot for Data Visualization**
- 13.4 Customizing Visuals with Matplotlib: From Basic Plots to Polished Charts**
- 13.5 Summary**
- 13.6 Glossary**
- 13.7 Answers to check your progress**
- 13.8 SUGGESTED READINGS – OER - OER**
- 13.9 Reference**
- 13.10 Exercise**

13.1 INTRODUCTION

Matplotlib is one of the most widely used libraries for data visualization in Python, offering robust tools to create a wide range of static, animated, and interactive plots.

At its core, Matplotlib's pyplot module provides a simple interface that closely resembles MATLAB, making it accessible even for beginners. With just a few lines of code, users can generate line charts, bar graphs, scatter plots, and more, turning raw data into meaningful visuals.

Understanding the basics of pyplot is the first step in mastering data presentation, from defining axes and labels to plotting trends. Beyond basic charts, Matplotlib also allows for deep customization—colors, styles, markers, and figure sizes can all be controlled for more professional results. Whether you're building dashboards, reports, or academic visuals, Matplotlib's versatility supports a wide range of analytical needs. This module introduces the foundation of plotting, key functions, and customization techniques to help you communicate insights effectively.

13.2 OBJECTIVE

After completing this Unit we will be able to:

- Define Matplotlib and its components.
- Create basic plots such as line, bar, and pie charts.
- Apply labeling, titles, and legends in visualizations.
- Interpret graphical data representations.

NOTE

**13.3 GETTING STARTED WITH MATPLOTLIB: MASTERING
PYPLOT FOR DATA VISUALIZATION**

Matplotlib is one of Python's most widely used libraries for creating static, animated, and interactive visualizations. At the heart of Matplotlib lies the Pyplot module, which offers a simple interface similar to MATLAB's plotting functions. Pyplot enables quick and intuitive plotting by offering functions for basic charts like line graphs, bar plots, and scatter plots with just a few lines of code. For beginners, this is often the go-to tool for visualizing data quickly.

Mastering Pyplot begins with understanding how to use `plt.plot()` for line charts, `plt.scatter()` for point-based visualizations, and `plt.bar()` for categorical comparisons. Beyond these basics, Pyplot supports a wide range of customization features. Users can change figure size, add labels and legends, tweak axis scales, and even apply styles like `seaborn`, `ggplot`, or `dark_background` to enhance the visual impact of their plots.

As you progress, Pyplot helps turn raw data into meaningful visual insights. Whether you're exploring trends, identifying outliers, or simply presenting your results, Pyplot acts as a powerful canvas. By mastering its features—from basic plots to full customization—you empower yourself to communicate data more effectively and make your analysis more engaging and informative.

Python code example using Matplotlib's Pyplot to demonstrate basic plotting, labeling, and customization:

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Create a line plot
plt.plot(x, y, color='blue', marker='o', linestyle='--', label='Sales')

# Add titles and labels
plt.title("Monthly Sales Data")
plt.xlabel("Month")
plt.ylabel("Sales in Units")

# Add grid and legend
plt.grid(True)
plt.legend()

# Customize style
plt.style.use('seaborn-v0_8')
```

NOTE

```
# Show plot
plt.show()
```

What this code does:

- Plots a dashed line with circle markers.
- Adds a title, axis labels, and a legend.
- Uses the seaborn style for cleaner aesthetics.
- Displays the grid for better readability.

13.4 CUSTOMIZING VISUALS WITH MATPLOTLIB: FROM BASIC PLOTS TO POLISHED CHARTS

Matplotlib, particularly its pyplot module, is a powerful Python library used for creating static, animated, and interactive visualizations. Customizing your plots helps communicate data insights more effectively. You can tailor nearly every aspect of a plot — colors, markers, line styles, fonts, legends, and even grid layouts — to make the visual appealing and insightful.

For example, customizing titles with specific font sizes and weights makes the chart easier to read. Adding legends helps distinguish between data series, while tweaking tick marks and axis ranges makes the graph easier to interpret. You can also use styles like ggplot or seaborn to give your charts a polished, publication-quality look with minimal effort.

Let's go from a basic plot to a fully customized chart using code:

Python Code Example: From Basic to Polished Plot

```
import matplotlib.pyplot as plt

# Sample data
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May']
sales_2024 = [100, 120, 130, 150, 170]
sales_2025 = [90, 110, 140, 160, 180]

# Use style for polished visuals
plt.style.use('ggplot')

# Create plot
plt.plot(months, sales_2024, label='2024 Sales', color='blue', marker='o',
         linestyle='--')
plt.plot(months, sales_2025, label='2025 Sales', color='green', marker='s',
         linestyle='-.')

# Customize
plt.title("Monthly Sales Comparison", fontsize=14, fontweight='bold')
```

NOTE

```
plt.xlabel("Months", fontsize=12)
plt.ylabel("Sales Units", fontsize=12)
plt.legend(loc='upper left', fontsize=10)
plt.grid(True, linestyle='--', linewidth=0.5)

# Set figure size and save
plt.figure(figsize=(8, 5))
plt.tight_layout()
plt.show()
```

Table 13.1: Basic vs Customized Plot Features in Matplotlib

Feature	Basic Plot	Customized Plot
Style	Default	ggplot, seaborn, etc. for visual aesthetics
Colors	Automatic	Defined (color='blue', etc.)
Markers & Lines	Plain line	Styled markers and line types ('o', '--', 's', etc.)
Title & Labels	Minimal or missing	Font size, bold, clear axis titles
Legends	Not included	Descriptive, positioned (loc='upper left')
Grid	Often off	Enabled for readability (grid=True)
Layout	Cramped	tight_layout() for better spacing
Figure Size	Default	Defined using figure(figsize=(8, 5))

**Activity 13.1**

Plot a line graph showing the growth of a variable (e.g., sales) over 12 months.

Label the X-axis as “Months” and Y-axis as “Sales”.

.....

.....

.....

NOTE**☑ Check Your Progress: 13.1**

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. What Matplotlib draws is more than plots it tells the story behind your data.

.....

Q2. What makes Pyplot powerful is its ability to turn raw numbers into clear visual insights.

.....

13.5 SUMMARY

Matplotlib is a foundational library in Python for data visualization, and its pyplot module offers an easy-to-use interface for creating a variety of plots. From simple line graphs to complex visualizations, Pyplot allows users to display data trends and patterns effectively. Its intuitive structure mimics MATLAB's plotting style, making it accessible to beginners while powerful enough for advanced users. By learning the basics, users can quickly visualize raw data and uncover insights that might be hidden in numbers.

Plotting in Pyplot begins with defining data and selecting an appropriate chart type such as bar, line, or scatter plots. Essential functions like `plot()`, `bar()`, and `scatter()` make data representation simple and readable. Axes labeling, titles, and gridlines help in adding context and clarity to each chart. Through iterative refinement, one can build informative and professional visualizations suited for academic, business, or scientific purposes.

Customization in Matplotlib plays a crucial role in refining visuals to enhance clarity and appeal. Users can adjust color schemes, line styles, markers, figure size, and annotations. This flexibility allows plots to be tailored for reports, presentations, or dashboards. Mastering these techniques empowers users to communicate data stories effectively and persuasively.

13.6 GLOSSARY

- **Matplotlib** – A Python library used for creating static, animated, and interactive data visualizations.
- **Plot** – A graphical representation of data points on a coordinate system.

NOTE

- **Figure** – The entire window or canvas that contains one or more plots.
- **Axes** – The plotting area within a figure where data is displayed.
- **Line Plot** – A chart that connects data points with straight lines to show trends over time or continuous data.
- **Bar Chart** – A graph that uses rectangular bars to compare values across different categories.
- **Scatter Plot** – A graph that displays individual data points to show relationships between two variables.
- **Histogram** – A graphical representation of the frequency distribution of numerical data.
- **Label** – Descriptive text used to identify axes, titles, legends, or data points in a chart.
- **Legend** – A guide that explains the meaning of different colors, symbols, or line styles used in a graph.

13.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 13.1

Answer 1:

The statement "What Matplotlib draws is more than plots—it tells the story behind your data." emphasizes the role of **data visualization** in conveying meaningful insights. While Matplotlib, especially through its pyplot module, is a tool for creating charts and graphs, its power lies in **making data understandable and memorable**. A well-crafted line chart, bar plot, or scatter plot can reveal trends, patterns, and relationships that raw data alone cannot.

For example, plotting temperature changes over time can instantly highlight seasonal patterns, while a correlation plot can uncover hidden relationships between variables. Thus, Matplotlib doesn't just plot points—it transforms data into a **visual narrative** that guides analysis, informs decisions, and enhances communication.

Answer 2:

The statement "What makes Pyplot powerful is its ability to turn raw numbers into clear visual insights." highlights the core strength of **Pyplot**, a submodule of **Matplotlib**. Pyplot allows users to transform numerical data into easily understandable **graphs and charts** such as line plots, bar graphs, scatter plots, and histograms with minimal code.

Instead of interpreting large tables of numbers, Pyplot enables quick generation of visuals that make patterns, trends, and anomalies immediately clear. For instance, using `plt.plot()` to visualize time-series data can help identify spikes,

dips, and steady growth. With options for titles, labels, legends, and styles, Pyplot enhances interpretability and presentation.

In summary, Pyplot is powerful because it bridges the gap between data and understanding, making complex datasets accessible, interpretable, and visually compelling.

NOTE

13.8 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/Use/UnitUrls?unitId=1274	

13.9 REFERENCE

- <https://matplotlib.org/stable/contents.html>
- <https://realpython.com/python-matplotlib-guide/>
- https://www.w3schools.com/python/matplotlib_intro.asp

13.10 EXERCISE

Part 1

A. Multiple choice question :

- What is the primary purpose of Matplotlib in Python?**
 - Image processing
 - Web development
 - Data visualization
 - Machine learning
- Which function is commonly used to create line plots in Matplotlib?**
 - plt.bar()
 - plt.plot()
 - plt.line()
 - plt.graph()
- What command is typically used to display a plot on the screen?**
 - show.plot()
 - plt.run()

NOTE

C. plt.display()

D. plt.show()

4. Which module within Matplotlib provides a simple interface for plotting?

A. matplotlib.core

B. matplotlib.pyplot

C. matplotlib.graph

D. pyplot.base

5. To customize the title of a plot, which function is used?

A. plt.name()

B. plt.label()

C. plt.header()

D. plt.title()

Answer with Explanations :**1. C) Data visualization**

Matplotlib is the primary Python library for creating static visual plots.

2. B) plt.plot()

plt.plot() is the standard function for creating line charts.

3. D) plt.show()

plt.show() displays the generated plot on the screen.

4. B) matplotlib.pyplot

The pyplot module provides an easy interface similar to MATLAB-style plotting.

5. D) plt.title()

plt.title() sets or customizes the plot's title.

B. Short answer type question

1. What is the role of plt.plot() in Matplotlib?
2. How do you display a plot after creating it with Matplotlib?
3. Name any two customization features available in Matplotlib plots.
4. Explain the importance of data visualization.

Part 2**C. Long answer type question (200 words)**

1. Explain the role of Pyplot in Matplotlib and how it simplifies the process of data visualization.

NOTE

2. Describe different types of plots available in Pyplot and their typical use cases.
3. How can you customize plots in Pyplot to improve clarity and presentation? Provide examples.
4. Discuss different types of basic plots—line, bar, scatter, and histogram. Explain when each type is used.

Part 3**D. Long Answer Type Questions: (300 words)**

1. Explain the importance of NumPy in scientific computing. Discuss how NumPy arrays differ from Python lists, emphasizing memory efficiency, speed, and functionality.
2. Describe the structure and features of a NumPy ndarray. Explain its attributes such as shape, size, ndim, dtype, and itemsize, with examples.
3. Discuss the process of installing NumPy, creating basic arrays, and performing simple array operations. Explain how these operations form the foundation for advanced numerical computing.
4. Explain labeling, legends, colors, styles, and formatting in Matplotlib and their importance in enhancing visual clarity.

UNIT 14

ADVANCED MATPLOTLIB

NOTE

UNIT STRUCTURE

- 14.1 Introduction**
- 14.2 Objective**
- 14.3 Mastering Matplotlib Layouts: From Subplots to Pie Charts**
- 14.4 Advanced Plotting Techniques with Matplotlib: Visualizing Data with Style and Clarity**
- 14.5 Summary**
- 14.6 Glossary**
- 14.7 Answers to check your progress**
- 14.8 SUGGESTED READINGS – OER - OER**
- 14.9 Reference**
- 14.10 Exercise**

14.1 INTRODUCTION

Matplotlib is a versatile plotting library in Python, and mastering its advanced features unlocks powerful ways to present and understand data. Moving beyond basic line plots, this module focuses on key visualization tools such as subplots, scatter plots, bar charts, histograms, and pie charts.

Each of these plot types offers a unique way to represent complex data distributions, relationships, and categories in a visual format that's easier to interpret.

Subplots allow multiple charts to be displayed in one figure, enabling side-by-side comparisons. Scatter plots highlight correlations and outliers, bar charts display categorical data, and histograms reveal data frequency and distribution.

Pie charts provide a clear view of proportions in a dataset. Together, these advanced techniques make Matplotlib a comprehensive tool for scientific visualization, exploratory data analysis, and reporting.

14.2 OBJECTIVE

After completing this Unit we will be able to:

- Customize plots using styles, colors, and layouts.
- Create advanced visualizations such as subplots and histograms.
- Analyze data through complex graphical representations.
- Evaluate visualization effectiveness for data communication.

NOTE

14.3 MASTERING MATPLOTLIB LAYOUTS: FROM SUBPLOTS TO PIE CHARTS

1. Subplots: Organizing Multiple Visuals

Subplots allow you to display multiple plots in a single figure, making it easier to compare datasets side by side. The most common method is `plt.subplots(nrows, ncols)`, which creates a grid of plots:

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 2)
axes[0, 0].plot([1, 2, 3], [1, 4, 9])
axes[0, 1].bar([1, 2, 3], [3, 2, 5])
axes[1, 0].scatter([1, 2, 3], [5, 2, 7])
axes[1, 1].hist([1, 2, 2, 3, 3, 3, 4])
plt.tight_layout()
plt.show()
```

This layout approach is ideal for dashboards, reports, or when comparing different visualizations of the same dataset. `tight_layout()` ensures that subplots don't overlap.

2. GridSpec for Advanced Layout Control

For more control, `matplotlib.gridspec` lets you define custom-sized grids and allocate space differently. This is useful when one chart (e.g., a time series) needs more space than others.

```
import matplotlib.gridspec as gridspec

fig = plt.figure()
gs = gridspec.GridSpec(2, 2, width_ratios=[2, 1])
ax1 = fig.add_subplot(gs[0])
ax2 = fig.add_subplot(gs[1])
ax3 = fig.add_subplot(gs[2:])
```

With this, you can create asymmetric, publication-style layouts.

3 Scatter and Bar Charts in Layouts

Scatter plots are great for visualizing relationships, and bar charts are perfect for categorical data. When used together in a layout, they provide both context and detail.

```
fig, axs = plt.subplots(1, 2, figsize=(10, 4))
axs[0].scatter([1, 2, 3], [4, 5, 6])
axs[1].bar(['A', 'B', 'C'], [10, 15, 7])
```

This helps compare numeric trends against categorical counts side-by-side.

NOTE

4 Pie Charts: Proportional Representations

Pie charts are useful for showing percentage data, especially when categories are few. You can use `explode`, `color`, `shadow`, and `autopct` for enhanced readability:

```
labels = ['A', 'B', 'C', 'D']
sizes = [25, 35, 20, 20]
explode = (0.1, 0, 0, 0)

plt.pie(sizes, labels=labels, explode=explode, autopct='%1.1f%%',
        shadow=True)

plt.title("Category Distribution")

plt.show()
```

You can even place a pie chart in a subplot or convert it into a donut chart by adding a white circle at the center.

5 Combining Different Chart Types

One powerful aspect of Matplotlib layouts is the ability to combine chart types. For example, you can display a pie chart, a bar chart, and a line chart in a 2x2 grid to present multi-dimensional analysis clearly.

6 Exporting Complex Layouts

Once your layout is complete, exporting with high resolution is key:

```
plt.savefig("multi_plot_layout.png", dpi=300, bbox_inches='tight')
```

Mastering Matplotlib layouts—from structured subplots to styled pie charts—lets you present data insights cleanly and impactfully. Whether it's for a report, dashboard, or publication, thoughtful layout design ensures your audience sees the story behind the numbers with ease and precision.

14.4 ADVANCED PLOTTING TECHNIQUES WITH MATPLOTLIB: VISUALIZING DATA WITH STYLE AND CLARITY

Matplotlib, one of Python's most widely used data visualization libraries, offers a wide array of advanced plotting techniques that go beyond basic line and bar charts. These advanced methods allow data scientists, analysts, and developers to communicate complex datasets more effectively by making the visuals both informative and aesthetically appealing. Mastering these techniques transforms simple plots into dynamic tools for storytelling and analysis.

- 1. Subplots and Layouts:** Subplots let users display multiple plots in a single figure using `plt.subplot()` or `plt.subplots()`. This is particularly

NOTE

helpful for comparing trends across different dimensions. Layout customization—such as shared axes, spacing, or figure size—makes the final output more organized and interpretable.

2. **Scatter Plots with Customization:** Scatter plots are essential for showing the relationship between two variables. Advanced options allow customization of marker size, color, shape, and even data-driven aesthetics like color maps (cmap) and transparency (alpha). This adds depth and clarity to your visual exploration.
3. **Bar Charts and Histograms:** Bar charts can be stacked, grouped, or horizontal to represent categorical data in different styles. Histograms reveal data distribution by binning continuous variables, with added features like density plots (density=True) or overlays. Fine-tuning color and orientation helps improve interpretability.
4. **Pie Charts and Donut Plots:** Although best used for limited categories, pie charts are useful for depicting proportions. Adding features like explode, shadows, or converting to donut plots (by adding a white circle in the center) makes them more visually engaging while maintaining clarity.
5. **Styling and Themes:** Matplotlib provides styling capabilities through `plt.style.use()` to apply themes like `seaborn`, `ggplot`, or `fivethirtyeight`. Additional elements such as grid lines, tick customization, titles, labels, and legends allow you to polish your charts for presentations or reports.
6. **Saving and Exporting:** Finally, using `plt.savefig()` lets you export figures in high-resolution formats like PNG, PDF, or SVG. You can specify DPI, transparent backgrounds, and bounding boxes—ensuring your visuals are publication-ready.



Activity 14.1

TITLE OF ACTIVITY: Multi-Panel Visualizations Using Matplotlib Subplots

OBJECTIVE: To build proficiency in creating multi-panel figures, managing complex subplot layouts, and selecting appropriate visual representations for diverse data types using Matplotlib.

MATERIAL NEEDED: A computer or smartphone with a Python programming environment installed (such as Google Colab, Jupyter Notebook, or VS Code), the matplotlib library, and a digital learning journal.

ACTIVITY: Let us elevate your data visualization skills by moving beyond single plots and mastering multi-panel dashboards. Your task is to write a Python script that instantiates a single figure window containing exactly two distinct subplots:

1. **Subplot 1 (Bar Chart):** Create a clear bar chart that displays mock final marks obtained by a student across 5 independent subjects

NOTE

(e.g., Mathematics, Physics, Data Science, English, and Management).

2. **Subplot 2 (Scatter Plot):** Create a scatter plot that maps and compares a sample distribution of student ages against their respective examination marks to identify if any visible trend exists.

To ensure your visualization complies with professional reporting standards, add a distinct title and explicit X and Y-axis labels to each individual subplot.

EXPECTATION: You should produce a clean, error-free Python script utilizing Matplotlib's subplot management functions (such as `plt.subplots()` or `plt.subplot()`). Your final output must render both plots dynamically in a unified frame, showing clearly legible labels, appropriate grid lines, and distinct colors for each plot type.

REFLECTION: Consider the visual design of executive dashboards. Why is it structurally more effective to combine these two distinct datasets into a single multi-panel figure using subplots rather than rendering them as isolated, independent graphs? Reflect on how layout management and clear labeling directly affect a business stakeholder's ability to interpret multi-dimensional data quickly without cognitive fatigue.

☑ Check Your Progress: 14.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. What Matplotlib reveals through plots, data explains without words.

.....

.....

.....

Q2. What transforms raw numbers into insight is a well-crafted chart.

.....

.....

.....

14.5 SUMMARY

Advanced Matplotlib techniques empower users to present data in visually compelling ways using a variety of chart types. Subplots allow for organizing multiple visualizations in one figure, making it easier to compare data across different dimensions or categories. Scatter plots are particularly useful in exploring relationships between variables, often used to identify trends, clusters, or outliers in two-dimensional data. By customizing markers, colors, and sizes, scatter plots can convey additional layers of insight. Bar charts and histograms are essential tools for analyzing frequency and distribution. Bar

NOTE

charts are ideal for categorical comparisons, where each bar represents a category and its corresponding value. Histograms, on the other hand, group continuous data into bins to show frequency distributions, aiding in understanding data spread and skewness. Pie charts, though best used sparingly, help visualize proportions and compositions within a dataset. These tools, when combined with Matplotlib's rich customization features—such as labeling, color palettes, and layout controls—enhance the clarity and effectiveness of data storytelling. Together, these advanced plotting techniques transform raw numbers into meaningful visuals that support informed decision-making.

14.6 GLOSSARY

- **Subplot** – A smaller plot within a figure used to display multiple graphs together.
- **Annotation** – Text or symbols added to a chart to highlight specific data points or observations.
- **Grid** – Horizontal and vertical reference lines that improve the readability of a graph.
- **Axis Limits** – The minimum and maximum values displayed on the x-axis and y-axis.
- **Tick Marks** – Small marks and labels on the axes that indicate measurement intervals.
- **Color Map (Colormap)** – A predefined range of colors used to represent numerical values in visualizations.
- **Style Sheet** – A collection of predefined formatting settings that control the appearance of plots.
- **Marker** – A symbol used to represent individual data points in a plot.
- **Error Bar** – A graphical element that indicates the variability or uncertainty of data values.
- **3D Plot** – A visualization that represents data using three axes: x, y, and z.

14.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 14.1**Answer 1:**

The quotation "**What Matplotlib reveals through plots, data explains without words.**" emphasizes the core purpose of **data visualization**—to communicate insights visually without relying on lengthy explanations.

Matplotlib, a powerful Python plotting library, transforms numerical data into meaningful visual formats like **scatter plots, bar charts, histograms, pie charts**, and **subplots**. These visuals allow users to identify **patterns, trends**,

NOTE

outliers, and relationships quickly, which might be harder to interpret in raw tabular data.

For example, a bar chart can instantly show sales comparisons across months, while a scatter plot might reveal a correlation between two variables. Instead of reading rows of numbers, viewers can "see" the story the data is telling.

Thus, Matplotlib turns numbers into visuals that **speak for themselves**—explaining the underlying message **without needing words**.

Answer 2:

The quote "**What transforms raw numbers into insight is a well-crafted chart.**" highlights the vital role of **data visualization** in data analysis. Raw numbers, though informative, often lack immediate clarity or interpretability. When represented through a well-designed chart—such as a line graph, histogram, scatter plot, or pie chart—these numbers begin to **tell a story**.

Using tools like **Matplotlib**, analysts can convert datasets into visual forms that **reveal patterns, trends, and correlations** at a glance. For example, a histogram can show the distribution of exam scores, while a pie chart can represent market share visually. These visual cues help users make quicker, more informed decisions.

In essence, a well-crafted chart does more than display data—it **communicates meaning**, making abstract numbers easier to understand, compare, and act upon. Therefore, visualization is not just about presentation but is a critical step in extracting **insight** from information.

14.8 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/Use/UnitUrls?unitId=1275	

14.9 REFERENCE

- https://matplotlib.org/stable/plot_types/index.html
- <https://realpython.com/python-matplotlib-guide/>
- <https://www.datacamp.com/courses/introduction-to-data-visualization-with-python>

NOTE

14.10 EXERCISE

Part 1

A. Multiple choice question:

1. What is the purpose of using `plt.subplot()` in Matplotlib

- A. To create a 3D plot
- B. To divide a figure into multiple plots
- C. To change the color scheme
- D. To save a plot to a file

2. Which function is used to create a scatter plot in Matplotlib?

- A. `plt.line()`
- B. `plt.bar()`
- C. `plt.scatter()`
- D. `plt.hist()`

3. In a histogram, what do the bars represent?

- A. Exact values in the dataset
- B. Frequency of values in intervals
- C. Relationships between variables
- D. Pie chart segments

4. Which argument in `plt.pie()` controls the percentage display on a pie chart?

- A. `colors`
- B. `explode`
- C. `labels`
- D. `autopct`

5. To place multiple plots in one figure, which of the following is correct?

- A. Use `plt.figure()` repeatedly
- B. Use `plt.grid()`
- C. Use `plt.subplots()` with axes
- D. Use `plt.pie()`

Answer with Explanations:

1. B) To divide a figure into multiple plots

Explanation: `plt.subplot()` divides the figure into a grid and allows multiple plots in one window.

2. C) `plt.scatter()`

Explanation: `plt.scatter()` creates a scatter plot of x-y points.

NOTE

3. **B)** Frequency of values in intervals

Explanation: Histogram bars show the frequency of data values within intervals (bins).

4. **D)** autopct

Explanation: autopct displays percentage values on pie chart slices.

5. **C)** Use plt.subplots() with axes

Explanation: plt.subplots() creates multiple plots using separate axes within one figure.

B. Short answer type question:

1. What is the primary difference between a scatter plot and a bar chart in Matplotlib?
2. How do you create a histogram in Matplotlib and what does it represent?
3. What is the use of plt.subplots() in Matplotlib?

Part 2

C. Long answer type Question (200 words)

1. Explain the purpose and advantages of using subplots in Matplotlib. How can subplots improve data visualization?
2. How does a scatter plot help in understanding relationships between two variables? Provide an example using Matplotlib.
3. Compare and contrast bar charts, histograms, and pie charts in Matplotlib. When should each be used?

Part 3

D. Long Answer Type Questions (300 words)

1. Explain the concept of subplots and how multi-plot figures are created. Discuss their importance in comparative analysis.
2. Describe customization options such as annotations, grid lines, color maps, and styling themes.
3. Discuss advanced visualization techniques and how Matplotlib integrates with NumPy and Pandas for powerful visual analytics.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | UCI | Member of: APU | Joint Committee: UGC | UER | AICTE | Recognized by: UGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:
