Day 4 - Session 1:
NumPy for Numerical Computation - Deep Dive

This notebook provides a deep dive into **NumPy(Numerical Python),** the foundational library for numerical computing in Python. NumPy's primary object, the `ndarray` (N-dimensional array), is critical for efficient storage and manipulation of large datasets, especially in scientific, engineering, and data science applications. We will cover array creation, indexing, slicing, vectorized operations, broadcasting, and fundamental linear algebra, suitable for learners from freshers to experienced programmers.

1. Introduction to NumPy: The Power of `ndarray`

Python lists are versatile, but they are not optimized for numerical operations on large datasets. They store heterogeneous objects and operations are often element-by-element, leading to slow performance for large arrays of numbers.

Why NumPy?

1. **Performance**: NumPy operations are implemented in C and Fortran, making them significantly faster than equivalent Python loop-based operations. This is crucial for computationally intensive tasks.
2. **Efficiency**: `ndarray` stores homogeneous data (all elements of the same type), allowing for compact memory storage and optimized operations.
3. **Mathematical Operations**: Provides a vast collection of high-level mathematical functions designed to operate on arrays.
4. **Foundation for Scientific Computing**: It's the backbone for many other scientific Python libraries like SciPy, Pandas, and Matplotlib.

The core object in NumPy is the `ndarray` (N-dimensional array), which is a grid of values, all of the same type, and is indexed by a tuple of non-negative integers.

Conceptual Approach:

We'll start by importing `numpy` (conventionally as np) and then demonstrate basic array creation.

Python
```python
import numpy as np  # Importing NumPy library for numerical operations

print("--- Introduction to NumPy ---")

# Creating a regular Python list
python_list = [1, 2, 3, 4, 5]

# Creating a NumPy array from the same values
numpy_array = np.array([1, 2, 3, 4, 5])

# Displaying both data structures and their types
print(f"Python list: {python_list}, Type: {type(python_list)}")
print(f"NumPy array: {numpy_array}, Type: {type(numpy_array)}")

# ----------------------------
# Comparing Operations
# ----------------------------

# Adding 1 to each element of a Python list requires a loop or list
comprehension
list_plus_one = [x + 1 for x in python_list]
print(f"List + 1 (manual using loop): {list_plus_one}")

# Adding 1 to a NumPy array is vectorized and more concise
# This applies the operation to each element internally
print(f"NumPy Array + 1 (vectorized): {numpy_array + 1}")

# Multiplication difference
# python_list * 2 would repeat the list (not element-wise multiplication)
# For learning purposes, this line is commented out:
# print(python_list * 2)  # Output: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

# NumPy performs element-wise multiplication automatically
print(f"NumPy Array * 2 (element-wise): {numpy_array * 2}")
```

2. NumPy Array Creation

NumPy provides various functions to create arrays, from converting Python lists to generating arrays with specific values or patterns.

2.1 From Python Lists and Tuples (`np.array()`)

The most common way to create an array is from a standard Python list or tuple.

Syntax: `numpy.array(object, dtype=None, ...)`

```python
print("--- Array Creation: From Python Lists/Tuples ---")

# --------------------------------------------------
# Creating a 1D NumPy array from a Python list
# --------------------------------------------------
data_1d = [1.0, 2.5, 3.0]                       # Python list of floats
arr_1d = np.array(data_1d)                      # Converted to NumPy array
print(f"1D Array: {arr_1d}, Shape: {arr_1d.shape}, Dtype: {arr_1d.dtype}")
# Shape: (3,) → 1D array with 3 elements
# Dtype: float64 by default since it contains decimal values

# --------------------------------------------------
# Creating a 2D NumPy array (Matrix)
# --------------------------------------------------
data_2d = [[1, 2, 3], [4, 5, 6]]                # List of lists = 2 rows x 3
columns
arr_2d = np.array(data_2d)                      # Converts to 2D NumPy array
print(f"\n2D Array:\n{arr_2d}, Shape: {arr_2d.shape}, Dtype: {arr_2d.dtype}")
# Shape: (2, 3) → 2 rows, 3 columns
# Dtype: int64 by default as all are integers

# --------------------------------------------------
# Explicitly specifying data types (dtype)
# --------------------------------------------------

# Integer array with 32-bit precision
int_array = np.array([1, 2, 3], dtype=np.int32)
print(f"\nInt32 Array: {int_array}, Dtype: {int_array.dtype}")
```

```python
# Floating-point array with 64-bit precision
float_array = np.array([1, 2, 3], dtype=np.float64)
print(f"Float64 Array: {float_array}, Dtype: {float_array.dtype}")

# Complex number array with 32-bit complex precision
complex_array = np.array([1 + 2j, 3 - 4j], dtype=np.complex64)
print(f"Complex Array: {complex_array}, Dtype: {complex_array.dtype}")

# --------------------------------------------------
# Creating an array from a Python tuple
# --------------------------------------------------
tuple_data = (7, 8, 9)
arr_from_tuple = np.array(tuple_data)
print(f"\nArray from Tuple: {arr_from_tuple}")
```

2.2 Arrays with Placeholder Values (`np.zeros(), np.ones(), np.empty()`)

Useful for pre-allocating memory for arrays you intend to fill later.

Syntax: `numpy.zeros(shape, dtype=float)`, `numpy.ones(shape, dtype=float)`, `numpy.empty(shape, dtype=float)`

Python
```python
print("\n--- Array Creation: Placeholders ---")

# --------------------------------------------------
# np.zeros: Creates an array filled with 0s
# --------------------------------------------------

# A 1D array of 5 zeros (default dtype is float64)
zeros_1d = np.zeros(5)
print(f"5 Zeros (1D): {zeros_1d}")

# A 2D array (3 rows, 4 columns) filled with zeros
zeros_2d = np.zeros((3, 4))  # Shape: (3, 4)
print(f"3x4 Zeros (2D):\n{zeros_2d}")
```

```
# --------------------------------------------------
# np.ones: Creates an array filled with 1s
# --------------------------------------------------

# A 2D array (2 rows, 3 columns) of 1s with 16-bit integer values
ones_2d = np.ones((2, 3), dtype=np.int16)
print(f"\n2x3 Ones (Int16):\n{ones_2d}")

# --------------------------------------------------
# np.empty: Creates an array without initializing values
# (Contents will be arbitrary garbage values from memory)
# --------------------------------------------------

# A 2x2 empty array (values are uninitialized, fast but risky for logic)
empty_array = np.empty((2, 2))
print(f"\n2x2 Empty Array:\n{empty_array}")
```

2.3 Arrays with Numerical Ranges (`np.arange()`, `np.linspace()`)

`arange` for step-based ranges, `linspace` for fixed number of points over a range.

Syntax: `numpy.arange([start,] stop[, step,], dtype=None)`, `numpy.linspace(start, stop, num=50, endpoint=True, ...)`

Python
```python
print("\n--- Array Creation: Numerical Ranges ---")

# --------------------------------------------------
# np.arange: Like Python's range(), but returns a NumPy array
# Useful for generating sequences with a fixed step size
# --------------------------------------------------

# Integers from 0 up to (but not including) 10, in steps of 2
arr_range = np.arange(0, 10, 2)  # [0, 2, 4, 6, 8]
print(f"Range (0 to 10, step 2): {arr_range}")
```

```
# Floating-point range from 0.5 to below 5.5 in steps of 1.0
arr_range_float = np.arange(0.5, 5.5, 1.0)
print(f"Range (0.5 to 5.5, step 1.0): {arr_range_float}")

# --------------------------------------------------
# np.linspace: Generates evenly spaced numbers over an interval
# Use this when you know how many values you want between two limits
# --------------------------------------------------

# 5 values between 0 and 10 (inclusive)
arr_linspace = np.linspace(0, 10, 5)
print(f"\nLinspace (0 to 10, 5 points): {arr_linspace}")
# Output: [  0.   2.5  5.   7.5 10.]
```

2.4 Random Arrays (`np.random.rand()`, `np.random.randn()`, `np.random.randint()`)

For simulations, generating random data is often necessary.

Syntax:

1. `numpy.random.rand(d0, d1, ...)`: Uniform distribution [0, 1)
2. `numpy.random.randn(d0, d1, ...)`: Standard normal distribution (mean 0, std dev 1)
3. `numpy.random.randint(low, high=None, size=None, dtype=int)`: Random integers

Python
```python
print("\n--- Array Creation: Random Numbers ---")

# --------------------------------------------------
# Uniform Distribution: np.random.rand()
# Generates values in [0.0, 1.0)
# --------------------------------------------------

# A 1D array of 5 random floats between 0 and 1
rand_1d = np.random.rand(5)
```

```python
print(f"5 Random Uniform (0,1): {rand_1d}")

# A 2D array (2 rows, 3 columns) of random floats between 0 and 1
rand_2d = np.random.rand(2, 3)
print(f"\n2x3 Random Uniform (0,1):\n{rand_2d}")

# --------------------------------------------------
# Standard Normal Distribution: np.random.randn()
# Mean = 0, Standard Deviation = 1
# --------------------------------------------------

# A 3x2 matrix with values from standard normal distribution
randn_array = np.random.randn(3, 2)
print(f"\n3x2 Random Normal (mu=0, sigma=1):\n{randn_array}")

# --------------------------------------------------
# Random Integers: np.random.randint()
# Generates random integers in a specified range
# --------------------------------------------------

# A 3x3 matrix with random integers from 1 to 9 (10 excluded)
rand_int_array = np.random.randint(1, 10, size=(3, 3))
print(f"\n3x3 Random Integers (1 to 9):\n{rand_int_array}")

# --------------------------------------------------
# Reproducibility: np.random.seed()
# Ensures the same random numbers are generated every time
# --------------------------------------------------

# Set the random seed (fixed start point for random number generation)
np.random.seed(42)
reproducible_rand = np.random.rand(3)
print(f"\nReproducible Random 1: {reproducible_rand}")

# Resetting the seed to the same value gives the same output
np.random.seed(42)
reproducible_rand_2 = np.random.rand(3)
print(f"Reproducible Random 2: {reproducible_rand_2}")  # Exactly same as above
```

Function	Description	Output Type
`np.random.rand()`	Uniform distribution `[0, 1)`	Floats
`np.random.randn()`	Standard normal distribution (mean=0, std=1)	Floats
`np.random.randint()`	Random integers in a specified range	Integers
`np.random.seed()`	Makes random outputs reproducible for debugging/training	No visible output

2.5 Identity Matrix (`np.eye()`, `np.identity()`)

An identity matrix is a square matrix with ones on the main diagonal and zeros elsewhere.

Syntax: `numpy.eye(N, M=None, k=0, dtype=float), numpy.identity(n, dtype=float)`

```python
print("\n--- Array Creation: Identity Matrix ---")

# --------------------------------------------------
# np.identity(n)
# Creates a square identity matrix of size n x n
# Identity matrix has 1s on the main diagonal and 0s elsewhere
# --------------------------------------------------

id_matrix_3x3 = np.identity(3)
print(f"3x3 Identity Matrix:\n{id_matrix_3x3}")
# Output:
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]

# --------------------------------------------------
# np.eye(N, M, k=0)
# More flexible than identity: allows rectangular shapes and diagonal offsets
# N = number of rows, M = number of columns, k = diagonal offset
# ---------------------------------------------------

# 4x5 matrix with 1s on the main diagonal (k=0)
eye_matrix_4x5 = np.eye(4, 5, k=0)
print(f"\n4x5 Identity-like Matrix (k=0):\n{eye_matrix_4x5}")
# Output:
# [[1. 0. 0. 0. 0.]
#  [0. 1. 0. 0. 0.]
#  [0. 0. 1. 0. 0.]
#  [0. 0. 0. 1. 0.]]

# 3x3 matrix with 1s on the diagonal shifted to the right by 1 (k=1)
eye_matrix_offset = np.eye(3, k=1)
print(f"\n3x3 Identity-like (k=1):\n{eye_matrix_offset}")
# Output:
# [[0. 1. 0.]
#  [0. 0. 1.]
#  [0. 0. 0.]]
```

Function	Description	Square?	Offset?
`np.identity(n)`	Creates square matrix with main diagonal	✅ Yes	❌ No
`np.eye(N, M, k)`	Creates matrix with diagonal at position `k`	✅ or ❌	✅ Yes

2.6 Full Array (`np.full()`)

Creates an array of a given shape filled with a specified fill value.

```python
print("\n--- Array Creation: Full Array ---")

# ---------------------------------------------------
# np.full(shape, fill_value)
# Creates an array of given shape, filled with the specified constant value
# ---------------------------------------------------

# Create a 3x3 array filled with the number 7
full_array_3x3 = np.full((3, 3), 7)
print(f"3x3 Array filled with 7s:\n{full_array_3x3}")
# Output:
# [[7 7 7]
#  [7 7 7]
#  [7 7 7]]

# Create a 1D array of length 5 filled with the string 'sensor'
full_array_str = np.full(5, "sensor")
print(f"\n1D Array filled with 'sensor': {full_array_str}")
# Output:
# ['sensor' 'sensor' 'sensor' 'sensor' 'sensor']
```

3. `ndarray` Attributes & Basics

The ndarray object has several important attributes that provide information about its structure and data.

Conceptual Approach:

Understanding these attributes (shape, dtype, ndim, size, itemsize) is crucial for working effectively with NumPy arrays. Reshaping allows you to change an array's dimensions without changing its data.

Code Implementation:

Python
```python
print("--- ndarray Attributes & Basics ---")

import numpy as np

# Creating a 2D NumPy array (2 rows x 3 columns)
arr_example = np.array([[10, 20, 30], [40, 50, 60]])

# Display the array
print(f"Array:\n{arr_example}")

# --------------------------------------------------
# Basic Attributes of ndarray
# --------------------------------------------------

# shape: returns (rows, columns)
print(f"Shape: {arr_example.shape} (2 rows, 3 columns)")

# dtype: data type of array elements
print(f"Data Type: {arr_example.dtype} (e.g., int64 or int32)")

# ndim: number of dimensions (axes)
print(f"Number of Dimensions: {arr_example.ndim}")

# size: total number of elements in the array
print(f"Total Elements: {arr_example.size}")

# itemsize: size of one element in bytes (e.g., 4 bytes for int32)
print(f"Size of one element (bytes): {arr_example.itemsize}")
```

```python
# --------------------------------------------------
# Reshaping Arrays
# --------------------------------------------------

print("\n--- Reshaping Arrays ---")

# Create a 1D array with 12 elements
arr_1d_flat = np.arange(12)
print(f"Original 1D Array: {arr_1d_flat}, Shape: {arr_1d_flat.shape}")

# Reshape to a 2D array: 3 rows, 4 columns
arr_2d_reshaped = arr_1d_flat.reshape(3, 4)
print(f"Reshaped to 3x4:\n{arr_2d_reshaped}, Shape: {arr_2d_reshaped.shape}")

# Using -1 to let NumPy automatically calculate the number of rows
arr_inferred_rows = arr_1d_flat.reshape(-1, 3)
print(f"Reshaped to (-1, 3):\n{arr_inferred_rows}, Shape:
{arr_inferred_rows.shape}")
# -1 means: "calculate the number of rows needed to fit 3 columns"

# --------------------------------------------------
# Flattening Arrays
# --------------------------------------------------

# flatten(): returns a COPY as a 1D array
arr_flat = arr_2d_reshaped.flatten()
print(f"Flattened array (copy): {arr_flat}, Shape: {arr_flat.shape}")

# ravel(): returns a VIEW if possible (more memory efficient)
arr_ravel = arr_2d_reshaped.ravel()
print(f"Raveled array (view): {arr_ravel}, Shape: {arr_ravel.shape}")
```

Attribute/Method	Description
`.shape`	Tuple of dimensions (rows, columns)
`.dtype`	Type of elements (e.g., int32, float64)

`.ndim`	Number of dimensions (axes)
`.size`	Total number of elements
`.itemsize`	Memory (in bytes) for one element
`.strides`	Byte steps to move in memory (row-wise, col-wise)
`.reshape()`	Changes shape without changing data
`.flatten()`	Creates a new 1D copy
`.ravel()`	Flattens with a view (shares memory if possible)

4. Array Indexing and Slicing

Accessing and extracting specific parts of an `ndarray` is crucial. NumPy offers powerful indexing and slicing capabilities similar to Python lists, but extended for multiple dimensions, boolean masks, and arbitrary index arrays.

Conceptual Approach:

Indexing uses square brackets `[]` to select elements. Slicing uses `[start:stop:step]` to select a range. Multi-dimensional arrays use comma-separated indices/slices for each dimension.

4.1 Basic Indexing (1D, 2D, 3D)

```python
print("--- Basic Indexing ---")

import numpy as np

# --------------------------------------------------
# 1D Array (Vector) Indexing
# --------------------------------------------------

sensor_data_1d = np.array([10, 12, 11, 15, 9])
print(f"1D Array: {sensor_data_1d}")
```

```
# Access elements by position (0-based indexing)
print(f"First element: {sensor_data_1d[0]}")     # Output: 10
print(f"Last element: {sensor_data_1d[-1]}")     # Output: 9 (negative index =
from end)
print(f"Third element: {sensor_data_1d[2]}")     # Output: 11

# --------------------------------------------------
# 2D Array (Matrix) Indexing
# Example: Sensor grid readings (3 rows x 3 columns)
# --------------------------------------------------

sensor_grid_2d = np.array([
    [10.1, 10.2, 10.3],   # Row 0
    [20.1, 20.2, 20.3],   # Row 1
    [30.1, 30.2, 30.3]    # Row 2
])

print(f"\n2D Array (Sensor Grid):\n{sensor_grid_2d}")

# Accessing individual elements using [row, column]
print(f"Element at [1, 2]: {sensor_grid_2d[1, 2]}")  # Output: 20.3

# Access a whole row (row 0)
print(f"First row: {sensor_grid_2d[0]}")             # Output: [10.1 10.2 10.3]

# Access a whole column (column 1)
print(f"Second column: {sensor_grid_2d[:, 1]}")      # Output: [10.2 20.2 30.2]
# ':' means all rows, '1' selects the second column

# --------------------------------------------------
# 3D Array (Tensor) Indexing
# Example: Volumetric data or layered sensor matrix
# Shape = (layers, rows, columns) = (3, 3, 3)
# --------------------------------------------------

volumetric_data = np.arange(27).reshape(3, 3, 3)
print(f"\n3D Array (Volumetric Data):\n{volumetric_data}")

# Access element at Layer 1, Row 2, Column 0
print(f"Element at [1, 2, 0]: {volumetric_data[1, 2, 0]}") # Output: 21
```

```
# Access a full slice (entire Layer 0)
print(f"Slice (layer 0):\n{volumetric_data[0, :, :]}")
# ':' selects all rows and columns within that layer
```

4.2 Slicing

Slicing works similarly to Python lists with `[start:stop:step]`.

Python
```python
print("\n--- Slicing ---")

import numpy as np

# ---------------------------------------------------
# 1D Array Slicing
# ---------------------------------------------------

# Create a 1D array with values from 10 to 100, step 10
data_points = np.arange(10, 101, 10)
print(f"1D Data Points: {data_points}")
# Output: [ 10  20  30  40  50  60  70  80  90 100]

# Slice: first 3 elements (index 0 to 2)
print(f"First 3 points: {data_points[:3]}")  # [10 20 30]

# Slice: elements from index 4 to 7 (exclusive of index 8)
print(f"Points from index 4 to 7: {data_points[4:8]}")  # [50 60 70 80]

# Slice: every second element from start to end
print(f"Every second point: {data_points[::2]}")  # [10 30 50 70 90]

# Slice: last 3 elements
print(f"Last 3 points: {data_points[-3:]}")  # [80 90 100]

# ---------------------------------------------------
# 2D Array Slicing
# ---------------------------------------------------
```

```
sensor_matrix = np.array([
    [1, 2, 3, 4],     # Row 0
    [5, 6, 7, 8],     # Row 1
    [9, 10, 11, 12],  # Row 2
    [13, 14, 15, 16]  # Row 3
])

print(f"\n2D Sensor Matrix:\n{sensor_matrix}")

# Slice rows 1 to 2 (inclusive), columns 0 to 2 (exclusive of col 3)
sub_matrix = sensor_matrix[1:3, 0:3]
print(f"Sub-matrix (rows 1-2, cols 0-2):\n{sub_matrix}")
# Output:
# [[ 5  6  7]
#  [ 9 10 11]]

# Get the last column from all rows
last_column = sensor_matrix[:, -1]
print(f"Last column: {last_column}")  # [ 4  8 12 16]

# Get every 2nd element from the first row
first_row_sparse = sensor_matrix[0, ::2]
print(f"First row, every 2nd element: {first_row_sparse}")  # [1 3]
```

4.3 Boolean Indexing (Masking)

Selects elements based on a boolean condition applied to the array. Returns elements where the condition is `True`.

Python
```
print("\n--- Boolean Indexing (Masking) ---")

import numpy as np
```

```python
# ---------------------------------------------------
# Sensor data: temperatures in °C
# ---------------------------------------------------
temperatures = np.array([22.5, 28.0, 19.3, 31.2, 25.0, 18.9, 30.5])
print(f"Temperatures: {temperatures}")

# ---------------------------------------------------
# Boolean Mask: Find temperatures > 28.0
# ---------------------------------------------------
high_temp_mask = temperatures > 28.0
print(f"High Temp Mask: {high_temp_mask}")
# Output: [False False False  True False False  True]

# Apply mask to get only the high temperatures
filtered_high_temps = temperatures[high_temp_mask]
print(f"Filtered High Temps: {filtered_high_temps}")
# Output: [31.2 30.5]

# ---------------------------------------------------
# Combine conditions using logical operators:
# (temperatures > 30) OR (temperatures < 20)
# Use: & (AND), | (OR), ~ (NOT)
# ---------------------------------------------------
critical_temp_mask = (temperatures > 30.0) | (temperatures < 20.0)
print(f"Critical Temp Mask: {critical_temp_mask}")
# Output: [False False  True  True False  True  True]

critical_temps = temperatures[critical_temp_mask]
print(f"Critical Temps: {critical_temps}")
# Output: [19.3 31.2 18.9 30.5]

# ---------------------------------------------------
# Modify array values based on condition
# Example: Set all temps < 20 to a minimum value of 20.0
# ---------------------------------------------------
temperatures[temperatures < 20.0] = 20.0
print(f"Temperatures after setting min: {temperatures}")
# Output: [22.5 28.  20.  31.2 25.  20.  30.5]
```

4.4 Fancy Indexing (Array of Indices)

Selects elements based on an array of integer indices. Allows selecting non-contiguous or repeated elements.

```python
import numpy as np

print("\n--- Fancy Indexing: 1D and 2D Simplified Examples ---")

# --------------------------------------------------
# 1️⃣ Fancy Indexing in 1D Arrays
# --------------------------------------------------

channels = np.array(['CH_A', 'CH_B', 'CH_C', 'CH_D', 'CH_E'])
print("1D Channel List:", channels)

# Select specific channels using index positions
selected_indices = [0, 2, 4]
selected_channels = channels[selected_indices]
print("Selected Channels [0, 2, 4]:", selected_channels)
# Output: ['CH_A' 'CH_C' 'CH_E']

# Reorder or repeat specific items
reordered = channels[[3, 1, 3, 0]]
print("Reordered/Repeated Channels [3, 1, 3, 0]:", reordered)
# Output: ['CH_D' 'CH_B' 'CH_D' 'CH_A']


# --------------------------------------------------
# 2️⃣ Fancy Indexing in 2D Arrays - Element-Wise Selection
# --------------------------------------------------

matrix = np.array([
    [1, 2, 3],    # row 0
    [4, 5, 6],    # row 1
    [7, 8, 9]     # row 2
])

print("\n2D Matrix:\n", matrix)

# Select elements at (0,1) and (2,2)
```

```
row_indices = [0, 2]
col_indices = [1, 2]

paired_elements = matrix[row_indices, col_indices]
print("\nSelected Elements (Paired Indexing):", paired_elements)
# Output: [2 9]


# --------------------------------------------------
# 3️⃣ Submatrix Extraction using np.ix_()
# --------------------------------------------------

# Select full submatrix: rows 0 & 2, columns 1 & 2
submatrix = matrix[np.ix_([0, 2], [1, 2])]
print("\nSubmatrix (Rows 0 & 2, Cols 1 & 2):\n", submatrix)
# Output:
# [[2 3]
#  [8 9]]
```

5. Vectorized Operations (Element-wise Operations & Universal Functions - ufuncs)

One of NumPy's greatest strengths is its ability to perform operations on entire arrays at once, without explicit Python loops. These are called **vectorized operations**. They are implemented as highly optimized C functions underneath, making them significantly faster.

Universal Functions (ufuncs) are NumPy functions that operate element-by-element on `ndarrays`. Many standard mathematical functions (`np.sin`, `np.exp`, `np.sqrt`) are ufuncs.

5.1 Arithmetic Operations

Standard arithmetic operators (`+`, `-`, `*`, `/`, `**`) perform element-wise operations on arrays.

Python

```python
print("--- Vectorized Arithmetic Operations ---")

import numpy as np

# --------------------------------------------------
# Define two 1D arrays: voltages and currents
# Both arrays must have the same shape for element-wise operations
# --------------------------------------------------

voltages = np.array([1.2, 1.5, 1.1])   # in Volts
currents = np.array([0.5, 0.4, 0.6])   # in Amperes

# --------------------------------------------------
# 1. Element-wise addition
# --------------------------------------------------
sum_volt_curr = voltages + currents
print(f"Voltages + Currents: {sum_volt_curr}")
# Output: [1.7 1.9 1.7]

# --------------------------------------------------
# 2. Element-wise multiplication (Power = V × I)
# --------------------------------------------------
powers = voltages * currents
print(f"Powers (V × I): {powers}")
# Output: [0.6  0.6  0.66] → in Watts

# --------------------------------------------------
# 3. Scalar operation: Multiply entire array by a constant
# --------------------------------------------------
scaled_voltages = voltages * 2.0
print(f"Scaled Voltages (x2): {scaled_voltages}")
# Output: [2.4 3.0 2.2]

# --------------------------------------------------
# 4. Element-wise division
# --------------------------------------------------
ratio = voltages / currents
print(f"Voltage/Current Ratio: {ratio}")
# Output: [2.4 3.75 1.83] → acts like resistance (Ohm's law)
# --------------------------------------------------
# 5. Element-wise exponentiation
# --------------------------------------------------
squared_voltages = voltages**2
print(f"Squared Voltages: {squared_voltages}")
# Output: [1.44 2.25 1.21]
```

5.2 Comparison and Logical Operators

Comparison operators (`>`, `<`, `==`, `<=`, `>=`) also work element-wise and return a boolean array. Logical operators (`np.logical_and`, `np.logical_or`, `np.logical_not`) combine boolean arrays.

Python
```python
print("\n--- Vectorized Comparison & Logical Operations ---")

import numpy as np

# --------------------------------------------------
# Temperature readings (in °C)
# --------------------------------------------------
temperatures = np.array([20, 25, 30, 35, 40])
print("Temperatures:", temperatures)

# --------------------------------------------------
# 1. Element-wise Comparison (creates a boolean array)
# --------------------------------------------------

# Check which values are greater than 30
is_hot = temperatures > 30
print(f"Is Hot (>30)? {is_hot}")
# Output: [False False False  True  True]

# Check which values are between 20 and 30 (inclusive)
```

```
is_nominal = (temperatures >= 20) & (temperatures <= 30)
# Use & for element-wise logical AND (not and)
print(f"Is Nominal (20-30)? {is_nominal}")
# Output: [ True  True  True False False]

# --------------------------------------------------
# 2. Using np.logical_and and np.logical_or
# These are alternative ways to combine conditions
# --------------------------------------------------

# Define individual conditions
condition1 = temperatures > 25
condition2 = temperatures < 35

# Combine using logical_and: values >25 and <35
combined_condition = np.logical_and(condition1, condition2)
print(f"Combined (25 < temp < 35): {combined_condition}")
# Output: [False False  True  True False]

# Filter actual temperature values matching the condition
filtered_values = temperatures[combined_condition]
print(f"Filtered values (25 < temp < 35): {filtered_values}")
# Output: [30 35]
```

5.3 Mathematical Universal Functions (ufuncs)

NumPy provides a wide array of ufuncs for element-wise mathematical operations.

Common Mathematical ufuncs

Function	Description	Example
`np.sin()`	Sine of elements (radians)	`np.sin(angles_rad)`
`np.cos()`	Cosine of elements (radians)	`np.cos(angles_rad)`

`np.tan()`	Tangent of elements (radians)	`np.tan(angles_rad)`
`np.exp()`	Exponential of elements (`e^x`)	`np.exp(data)`
`np.log()`	Natural logarithm of elements	`np.log(data)`
`np.sqrt()`	Square root of elements	`np.sqrt(values)`
`np.abs()`	Absolute value of elements	`np.abs(deviations)`
`np.ceil()`	Ceiling of elements (round up)	`np.ceil(temps)`
`np.floor()`	Floor of elements (round down)	`np.floor(temps)`
`np.round()`	Round to nearest integer	`np.round(values)`
`np.maximum()`	Element-wise maximum of two arrays/scalars	`np.maximum(arr1, arr2)`
`np.minimum()`	Element-wise minimum of two arrays/scalars	`np.minimum(arr1, arr2)`

Python
```python
print("\n--- Universal Functions (ufuncs): Simplified ---")

import numpy as np

# --------------------------------------------------
# 1. Square root of each element
# --------------------------------------------------
numbers = np.array([4, 9, 16, 25])
sqrt_result = np.sqrt(numbers)
print(f"Square root of {numbers}: {sqrt_result}")
# Output: [2. 3. 4. 5.]

# --------------------------------------------------
# 2. Absolute values (ignores sign)
# --------------------------------------------------
values = np.array([-3, 0, 2, -7])
abs_result = np.abs(values)
print(f"Absolute values of {values}: {abs_result}")
# Output: [3 0 2 7]

# --------------------------------------------------
# 3. Exponentials (e^x for each element)
# --------------------------------------------------
x = np.array([0, 1, 2])
exp_result = np.exp(x)
print(f"Exponential of {x}: {exp_result}")
# Output: [1.         2.71828183 7.3890561 ]

# --------------------------------------------------
# 4. Trigonometric functions (sin, cos)
# Angles are in degrees, so convert to radians first
# --------------------------------------------------
angles_deg = np.array([0, 30, 90])
angles_rad = np.radians(angles_deg)

sin_values = np.sin(angles_rad)
cos_values = np.cos(angles_rad)

print(f"Angles (degrees): {angles_deg}")
print(f"Sin values: {sin_values}")
print(f"Cos values: {cos_values}")
# Output:
# Sin: [0.0, 0.5, 1.0]
# Cos: [1.0, 0.866, 0.0]
```

6. Broadcasting

Broadcasting is a powerful mechanism that allows NumPy to perform operations on arrays of different shapes. It automatically "stretches" the smaller array across the larger array so that they have compatible shapes for element-wise operations.

Conceptual Approach:

NumPy follows a strict set of rules for broadcasting:

1. **Dimensions Compatibility:** Starting from the trailing dimension, the sizes of the dimensions must either be equal, or one of them must be 1.
2. **Size 1 Dimensions:** If one of the dimensions is 1, it's stretched to match the other array's dimension.
3. **Non-existent Dimensions:** A dimension that doesn't exist is treated as having a size of 1.

If these rules are not met, a ValueError (broadcast error) is raised.

Code Implementation:

```python
import numpy as np

print("\n--- Broadcasting: Simplified Examples ---")

# --------------------------------------------------
# 1. Scalar + Array (Basic Broadcasting)
# --------------------------------------------------
arr = np.array([1, 2, 3])
result = arr + 10  # 10 is broadcast to each element
print("Array:", arr)
print("Array + 10:", result)
# Output: [11 12 13]

# --------------------------------------------------
# 2. Add Row Vector to 2D Matrix
# --------------------------------------------------
matrix = np.array([
    [1, 2, 3],
    [4, 5, 6]
])
row = np.array([10, 20, 30])  # Shape: (3,)

# Broadcasted across rows
added = matrix + row
```

```python
print("\nMatrix:\n", matrix)
print("Row Vector:", row)
print("Matrix + Row Vector:\n", added)
# Output:
# [[11 22 33]
#  [14 25 36]]

# --------------------------------------------------
# 3. Add Column Vector to 2D Matrix
# --------------------------------------------------
col = np.array([[100], [200]])  # Shape: (2,1)

# Broadcasted across columns
added_col = matrix + col
print("\nColumn Vector:\n", col)
print("Matrix + Column Vector:\n", added_col)
# Output:
# [[101 102 103]
#  [204 205 206]]
```

7. Array Manipulation

NumPy provides functions to combine arrays (`concatenate`, `stack`), split arrays (`split`), and modify their dimensions (`newaxis`, `expand_dims`, `squeeze`).

7.1 Concatenation (`np.concatenate()`, `np.vstack()`, `np.hstack()`)

Combining arrays along an existing axis.

Syntax: `numpy.concatenate((a1, a2, ...), axis=0)`

1. `np.vstack((a1, a2, ...))`: Stack arrays vertically (row-wise).
2. `np.hstack((a1, a2, ...))`: Stack arrays horizontally (column-wise).

Python

```python
import numpy as np

print("\n--- Array Manipulation: Concatenation & Stacking (Simplified) ---")

# --------------------------------------------------
# 1. Concatenating 1D Arrays
# --------------------------------------------------

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

concat_1d = np.concatenate((arr1, arr2))
print("1D Arrays:")
print("arr1:", arr1)
print("arr2:", arr2)
print("Concatenated (1D):", concat_1d)
# Output: [1 2 3 4 5 6]

# --------------------------------------------------
# 2. Concatenating 2D Arrays
# --------------------------------------------------

m1 = np.array([[10, 20], [30, 40]])
m2 = np.array([[50, 60], [70, 80]])

print("\n2D Arrays:")
print("Matrix 1:\n", m1)
print("Matrix 2:\n", m2)

# Vertical concatenation (adds rows)
v_cat = np.concatenate((m1, m2), axis=0)
print("\nConcatenate (axis=0 - vertical):\n", v_cat)

# Horizontal concatenation (adds columns)
h_cat = np.concatenate((m1, m2), axis=1)
print("\nConcatenate (axis=1 - horizontal):\n", h_cat)

# vstack & hstack do the same as axis=0 and axis=1
print("\nvstack:\n", np.vstack((m1, m2)))
print("hstack:\n", np.hstack((m1, m2)))
```

```python
# --------------------------------------------------
# 3. Stacking Arrays with np.stack
# Adds a NEW axis (3rd dimension)
# --------------------------------------------------

# Stack along new axis=0 → results in 3D array
stack_axis0 = np.stack((m1, m2), axis=0)
print("\nnp.stack with axis=0 (creates 3D array):")
print(stack_axis0)
print("Shape:", stack_axis0.shape)
# Shape: (2, 2, 2) → 2 matrices stacked on top of each other

# Stack along axis=1 → stacks row-wise as inner dimension
stack_axis1 = np.stack((m1, m2), axis=1)
print("\nnp.stack with axis=1:")
print(stack_axis1)
print("Shape:", stack_axis1.shape)
# Shape: (2, 2, 2)

# Stack along axis=2 → each element becomes a depth-wise layer
stack_axis2 = np.stack((m1, m2), axis=2)
print("\nnp.stack with axis=2:")
print(stack_axis2)
print("Shape:", stack_axis2.shape)
# Shape: (2, 2, 2)
```

7.2 Splitting Arrays (`np.split()`, `np.vsplit()`, `np.hsplit()`)

Dividing an array into multiple sub-arrays.

Syntax: `numpy.split(ary, indices_or_sections, axis=0)`

1. `np.vsplit(ary, indices_or_sections)`: Split vertically (along rows).
2. `np.hsplit(ary, indices_or_sections)`: Split horizontally (along columns).

Python

```python
import numpy as np

print("\n--- Array Manipulation: Splitting ---")

# --------------------------------------------------
# Create a 4x4 matrix to demonstrate splitting
# --------------------------------------------------
large_array = np.arange(16).reshape(4, 4)
print("Large Array:\n", large_array)
# Output:
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]
#  [12 13 14 15]]

# --------------------------------------------------
# 1. Vertical Split (Split by rows) → like breaking matrix into blocks stacked
vertically
# --------------------------------------------------
v_split_arrays = np.vsplit(large_array, 2)  # Split into 2 equal row blocks

print("\nVertical Split (2 parts):")
print(v_split_arrays[0])
print("---")
print(v_split_arrays[1])

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

# --------------------------------------------------
# 2. Horizontal Split (Split by columns) → slice vertically like Excel columns
# --------------------------------------------------
# Split before column index 1, and before column index 3
# This gives 3 parts: columns 0, columns 1–2, columns 3
h_split_arrays = np.hsplit(large_array, [1, 3])

print("\nHorizontal Split at column indices [1, 3]:")
```

```python
for arr in h_split_arrays:
    print(arr)
    print("---")

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

7.3 Adding/Removing Dimensions (`np.newaxis`, `np.expand_dims`, `np.squeeze`)

Modifying the number of dimensions of an array.

Syntax:

1. `arr[np.newaxis, :] or arr[:, np.newaxis]`: Adds a new axis.
2. `numpy.expand_dims(a, axis)`: Adds a new axis at a specified position.
3. `numpy.squeeze(a, axis=None)`: Removes dimensions of size 1.

Python

```python
import numpy as np

print("\n--- Array Manipulation: Adding/Removing Dimensions ---")

# --------------------------------------------------
# Start with a 1D array
# --------------------------------------------------
data_1d = np.array([1, 2, 3])
print(f"Original 1D array: {data_1d}")
print(f"Shape: {data_1d.shape}")  # Shape: (3,)

# --------------------------------------------------
# Add a new axis to make it a 2D row vector → shape becomes (1, 3)
# --------------------------------------------------
row_vector = data_1d[np.newaxis, :]  # or data_1d[None, :]
print(f"\nAs Row Vector:\n{row_vector}")
print(f"Shape: {row_vector.shape}")  # Shape: (1, 3)

# --------------------------------------------------
# Add a new axis to make it a 2D column vector → shape becomes (3, 1)
# --------------------------------------------------
col_vector = data_1d[:, np.newaxis]  # or data_1d[:, None]
print(f"\nAs Column Vector:\n{col_vector}")
print(f"Shape: {col_vector.shape}")  # Shape: (3, 1)

# --------------------------------------------------
# Use np.expand_dims() to insert a new axis at any position
# --------------------------------------------------
expanded_array = np.expand_dims(data_1d, axis=0)  # Shape becomes (1, 3)
print(f"\nExpanded dims (axis=0):\n{expanded_array}")
print(f"Shape: {expanded_array.shape}")

# --------------------------------------------------
# np.squeeze(): Remove dimensions with size 1
# --------------------------------------------------
redundant_dims_array = np.array([[[1], [2], [3]]])  # Shape: (1, 3, 1)
print(f"\nArray with Redundant Dims:\n{redundant_dims_array}")
print(f"Shape: {redundant_dims_array.shape}")  # Shape: (1, 3, 1)

squeezed_array = np.squeeze(redundant_dims_array)
print(f"\nSqueezed Array:\n{squeezed_array}")
print(f"Shape: {squeezed_array.shape}")  # Shape: (3,)
```

8. Linear Algebra Operations

NumPy provides a robust set of functions in the `numpy.linalg` submodule for linear algebra operations, which are fundamental in many engineering and scientific calculations (e.g., solving systems of equations, matrix transformations, principal component analysis).

8.1 Dot Product (`np.dot()`, `@` operator)

For 1D arrays, it's the inner product. For 2D arrays, it's matrix multiplication.

Syntax: `numpy.dot(a, b), a @ b` (matrix multiplication operator)

```python
import numpy as np

print("\n--- Linear Algebra: Dot Product & Matrix Multiplication (Simplified)
---")

# ---------------------------------------------------
# 1. Dot Product of Two 1D Vectors
# ---------------------------------------------------
a = np.array([1, 2])
b = np.array([3, 4])

# Dot product = 1*3 + 2*4 = 3 + 8 = 11
dot_result = np.dot(a, b)
print("Vector A:", a)
print("Vector B:", b)
print("Dot Product (A . B):", dot_result)

# ---------------------------------------------------
# 2. Matrix Multiplication (2x2 matrices)
# ---------------------------------------------------
m1 = np.array([[1, 2],
               [3, 4]])

m2 = np.array([[5, 6],
               [7, 8]])
```

```
# Matrix multiplication using @
matrix_result = m1 @ m2
print("\nMatrix 1:\n", m1)
print("Matrix 2:\n", m2)
print("Matrix Multiplication (m1 @ m2):\n", matrix_result)

# -------------------------------------------------
# 3. Rotation of a 2D Vector (90° Counter-Clockwise)
# -------------------------------------------------
theta = np.radians(90)  # Convert 90 degrees to radians

# 2D rotation matrix
rotation = np.array([
    [np.cos(theta), -np.sin(theta)],
    [np.sin(theta),  np.cos(theta)]
])

vector = np.array([1, 0])  # Unit vector along x-axis

rotated = rotation @ vector
print("\nOriginal Vector:", vector)
print("Rotation Matrix (90°):\n", np.round(rotation, 2))
print("Rotated Vector:", np.round(rotated, 2))  # Rounded for clarity
```

8.2 Transpose (.T, np.transpose())

Swaps the axes of an array. For a 2D matrix, rows become columns and columns become rows.

Syntax: `arr.T, numpy.transpose(arr)`

Python
```python
import numpy as np

print("\n--- Linear Algebra: Transpose ---")
```

```python
# ------------------------------------------
# Original matrix with 2 rows and 3 columns
# ------------------------------------------
matrix_t = np.array([
    [1, 2, 3],
    [4, 5, 6]
])

print("Original Matrix (2x3):")
print(matrix_t)

# ------------------------------------------
# Transpose the matrix: rows become columns
# and columns become rows
# ------------------------------------------
transposed_matrix = matrix_t.T

print("\nTransposed Matrix (3x2):")
print(transposed_matrix)
```

8.3 Determinant (`np.linalg.det()`)

Calculates the determinant of a square matrix.

Syntax: `numpy.linalg.det(a)`

Python
```python
import numpy as np

print("\n--- Linear Algebra: Determinant ---")

# ------------------------------------------
# Define a 2x2 square matrix
# ------------------------------------------
```

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

print("Matrix:")
print(matrix_d)

# ------------------------------------------
# Calculate the determinant
# Formula for 2x2 matrix [[a, b], [c, d]] is: (a*d - b*c)
# For this matrix: (1*4 - 2*3) = 4 - 6 = -2
# ------------------------------------------
determinant = np.linalg.det(matrix_d)

print("\nDeterminant:", determinant)
```

8.4 Inverse (`np.linalg.inv()`)

Calculates the multiplicative inverse of a square matrix. $A \cdot A^{-1} = I$ (Identity Matrix). The inverse exists only if the determinant is non-zero.

Syntax: `numpy.linalg.inv(a)`

Python
```python
import numpy as np

print("\n--- Linear Algebra: Inverse ---")

# ------------------------------------------
# Define a square matrix (2x2)
# ------------------------------------------
matrix_inv = np.array([
    [1, 2],
    [3, 4]
])
```

```python
print("Original Matrix:")
print(matrix_inv)

# ------------------------------------------
# Compute the inverse of the matrix
# Only possible if the matrix is square and has a non-zero determinant
# ------------------------------------------
inverse_matrix = np.linalg.inv(matrix_inv)

print("\nInverse Matrix:")
print(inverse_matrix)

# ------------------------------------------
# Verify the result:
# Multiplying a matrix with its inverse gives the identity matrix
# M @ M_inv ≈ Identity
# ------------------------------------------
identity_check = matrix_inv @ inverse_matrix
print("\nVerification (Matrix × Inverse ≈ Identity):")
print(identity_check.round(5))  # Rounded to 5 decimal places
```

8.5 Eigenvalues and Eigenvectors (`np.linalg.eig()`, `np.linalg.eigh()`)

Eigenvalues and eigenvectors are fundamental concepts in linear algebra with applications in vibration analysis, quantum mechanics, and principal component analysis.

1. `np.linalg.eig()`: For general matrices. Returns eigenvalues and eigenvectors.
2. `np.linalg.eigh()`: For symmetric or Hermitian matrices (more efficient and numerically stable).

Syntax: `w, v = numpy.linalg.eig(a)` (w are eigenvalues, v are eigenvectors)

Python

```python
import numpy as np

print("\n--- Linear Algebra: Eigenvalues and Eigenvectors (Simplified) ---")

# --------------------------------------------------
# A simple 2x2 symmetric matrix (easy to work with)
# Often used in statistics or physics (e.g., covariance matrix)
# --------------------------------------------------
A = np.array([
    [2, 1],
    [1, 2]
])

print("Matrix A:\n", A)

# --------------------------------------------------
# Compute eigenvalues and eigenvectors
# --------------------------------------------------
eigenvalues, eigenvectors = np.linalg.eig(A)

print("\nEigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)

# --------------------------------------------------
# Verify the relation: A @ v = λ * v for the first eigenpair
# --------------------------------------------------
v1 = eigenvectors[:, 0]   # First eigenvector
λ1 = eigenvalues[0]       # First eigenvalue

print("\nVerification:")
print("A @ v1:", A @ v1)
print("λ1 * v1:", λ1 * v1)
# These two results should be nearly identical
```

8.6 Solving Linear Equations (`np.linalg.solve()`)

Solves a system of linear equations of the form Ax=b, where A is a square matrix, x is the unknown vector, and b is a constant vector.

Syntax: `numpy.linalg.solve(a, b)`

```python
import numpy as np

print("\n--- Linear Algebra: Solving Linear Equations (Ax = b) ---")

# ---------------------------------------------
# We want to solve the following system:
#     x + 2y = 9
#   3x + 4y = 23
# ---------------------------------------------

# Matrix A contains the coefficients of x and y
A = np.array([
    [1, 2],  # Equation 1: 1*x + 2*y
    [3, 4]   # Equation 2: 3*x + 4*y
])

# Vector b contains the right-hand side values
b = np.array([9, 23])

print("Matrix A (Coefficients):\n", A)
print("Vector b (Results):", b)

# ------------------------------------------------------
# Solve for vector x = [x, y] using np.linalg.solve
# ------------------------------------------------------
solution = np.linalg.solve(A, b)

print("\nSolution Vector [x, y]:", solution)  # Should be [5, 2]

# ------------------------------------------------------
# Verification: Multiply A with solution → Should get back b
# ------------------------------------------------------
print("Verification (A @ x):", A @ solution)
```

9. Statistical Operations

NumPy provides a wide range of functions for basic statistical analysis, which can be applied to entire arrays or along specific axes (dimensions).

9.1 Mean, Median, Standard Deviation, Variance

Syntax:

1. `arr.mean(axis=None)` or `np.mean(arr, axis=None)`
2. `np.median(arr, axis=None)`
3. `arr.std(axis=None)` or `np.std(arr, axis=None)`
4. `arr.var(axis=None)` or `np.var(arr, axis=None)`

Python
```python
import numpy as np

print("--- Statistical Operations ---")

# Simulated sensor readings over 3 time points (columns)
# Each row represents a different sensor
sensor_readings_daily = np.array([
    [25.0, 25.5, 26.0],     # Sensor 1
    [101.0, 101.2, 100.8],  # Sensor 2
    [5.1, 5.0, 5.2]         # Sensor 3
])

print("Sensor Readings (Each Row = 1 Sensor, Each Col = 1 Time Point):")
print(sensor_readings_daily)

# ---------------------------
# Mean (Average)
# ---------------------------

# Mean of all values (entire matrix)
overall_mean = sensor_readings_daily.mean()
print(f"\nOverall Mean (All Values): {overall_mean:.2f}")

# Mean across each column → time-wise average across all sensors
```

```python
mean_per_column = sensor_readings_daily.mean(axis=0)
print(f"Mean per Time Point (Column-wise): {mean_per_column.round(2)}")

# Mean across each row → average reading for each sensor
mean_per_row = sensor_readings_daily.mean(axis=1)
print(f"Mean per Sensor (Row-wise): {mean_per_row.round(2)}")

# ---------------------------
# Standard Deviation
# ---------------------------

# Spread of all values
std_dev_all = sensor_readings_daily.std()
print(f"\nOverall Standard Deviation: {std_dev_all:.2f}")

# Spread of readings per sensor
std_dev_per_row = sensor_readings_daily.std(axis=1)
print(f"Standard Deviation per Sensor: {std_dev_per_row.round(2)}")

# ---------------------------
# Variance
# ---------------------------

# Variance = Standard Deviation²
variance_all = sensor_readings_daily.var()
print(f"\nOverall Variance: {variance_all:.2f}")

# ---------------------------
# Median
# ---------------------------

# Median of all values
median_all = np.median(sensor_readings_daily)
print(f"\nOverall Median: {median_all:.2f}")

# Median at each time point across sensors
median_per_column = np.median(sensor_readings_daily, axis=0)
print(f"Median per Time Point: {median_per_column.round(2)}")
```

9.2 Min, Max, Sum, Product

Syntax:

1. `arr.min(axis=None)` or `np.min(arr, axis=None)`
2. `arr.max(axis=None)` or `np.max(arr, axis=None)`
3. `arr.sum(axis=None)` or `np.sum(arr, axis=None)`
4. `arr.prod(axis=None)` or `np.prod(arr, axis=None)`

Python
```python
import numpy as np

print("\n--- Min, Max, Sum, Product ---")

# Sample 2D array of sensor-like data
data_values = np.array([
    [10, 5, 8],     # Row 0
    [12, 15, 7]     # Row 1
])
print("Data Values:\n", data_values)

# ---------------------------
# Maximum and Minimum Values
# ---------------------------

# Find the largest value in the entire array
max_val = data_values.max()
print(f"\nMaximum Value (Overall): {max_val}")

# Find the smallest value in each column (axis=0 → column-wise)
min_per_column = data_values.min(axis=0)
print(f"Minimum per Column: {min_per_column}")

# ---------------------------
# Summation
# ---------------------------

# Add all values in the array
total_sum = data_values.sum()
print(f"\nTotal Sum of All Elements: {total_sum}")
```

```
# Sum each row (axis=1 → row-wise)
sum_per_row = data_values.sum(axis=1)
print(f"Sum per Row: {sum_per_row}")

# ---------------------------
# Product
# ---------------------------

# Multiply all values in the array
total_product = data_values.prod()
print(f"\nProduct of All Elements: {total_product}")
```

10. Memory Layout and Performance Benefits

Understanding how NumPy stores data and performs operations is key to leveraging its performance advantages.

10.1 Memory Layout (Briefly)

NumPy arrays are stored in a contiguous block of memory. This allows for efficient caching and vectorized operations by the underlying C/Fortran code.

1. **C-contiguous (row-major):** Elements of a row are stored adjacently in memory. This is NumPy's default. `arr.flags['C_CONTIGUOUS']` will be `True`.
2. **F-contiguous (column-major):** Elements of a column are stored adjacently. `arr.flags['F_CONTIGUOUS']` will be `True`.

Python
```python
import numpy as np

print("--- Memory Layout ---")
```

```
# Create a 2x3 matrix in row-major (C-style) order
c_order_array = np.arange(6).reshape(2, 3)
print("C-order Array (2x3):\n", c_order_array)

# Check memory layout flags
print("C-contiguous? ", c_order_array.flags['C_CONTIGUOUS'])  # ✅ True →
row-wise in memory
print("F-contiguous? ", c_order_array.flags['F_CONTIGUOUS'])  # ❌ False

# Now transpose the array (creates a view, changes layout)
f_order_array = c_order_array.T
print("\nF-order Array (Transposed View - 3x2):\n", f_order_array)

# After transpose, the memory layout changes
print("C-contiguous? ", f_order_array.flags['C_CONTIGUOUS'])  # ❌ Now it's not
row-wise
print("F-contiguous? ", f_order_array.flags['F_CONTIGUOUS'])  # ✅ It's now
column-wise
```

10.2 Performance Benefits: Python Loops vs. NumPy Vectorization

This is where NumPy truly shines. For numerical operations on large datasets, NumPy's vectorized approach is vastly superior.

```python
import numpy as np
import time

print("\n--- Performance Benefits: Python Loops vs. NumPy ---")

# Create a large NumPy array with 1 million random values
large_data = np.random.rand(1_000_000)

# Scalar to add to each element
scalar_to_add = 5.0
```

```
# -------------------------
# Method 1: Using a Python loop
# -------------------------
start_time_py = time.perf_counter()

# Convert to list for fair comparison (as list comprehensions operate on Python
lists)
python_list_result = [x + scalar_to_add for x in large_data.tolist()]

end_time_py = time.perf_counter()
time_py = end_time_py - start_time_py

print(f"Python loop execution time: {time_py:.6f} seconds")

# -------------------------
# Method 2: Using NumPy vectorized operation
# -------------------------
start_time_np = time.perf_counter()

# NumPy handles the operation in C-level backend (much faster)
numpy_array_result = large_data + scalar_to_add

end_time_np = time.perf_counter()
time_np = end_time_np - start_time_np

print(f"NumPy vectorized execution time: {time_np:.6f} seconds")

# -------------------------
# Comparison
# -------------------------
print(f"\n✅ NumPy is approximately {time_py / time_np:.2f} times faster for
this operation.")

# Optional: Verify correctness (first 5 values)
# print("Python List Result (first 5):", python_list_result[:5])
# print("NumPy Result (first 5):", numpy_array_result[:5])
```

11. Key Takeaways from Day 4, Session 1

This session has provided a comprehensive introduction to NumPy, the indispensable library for numerical computation in Python.

1. **NumPy's Core (ndarray):** Understood the fundamental ndarray object as a highly efficient, homogeneous, N-dimensional array, superior to Python lists for numerical tasks.
2. **Array Creation:** Mastered various methods for creating arrays, from Python lists to specialized functions like np.zeros(), np.arange(), np.linspace(), and random number generators.
3. **ndarray Attributes:** Learned about essential array attributes such as shape, dtype, ndim, and size, and how to reshape arrays.
4. **Powerful Indexing & Slicing:** Gained proficiency in accessing and manipulating array elements using basic indexing, multi-dimensional slicing, boolean masking, and advanced fancy indexing.
5. **Vectorized Operations & Ufuncs:** Discovered the immense performance benefits of NumPy's vectorized operations and universal functions (ufuncs), which apply operations element-wise on entire arrays, leveraging underlying C/Fortran implementations.
6. **Broadcasting:** Understood how NumPy intelligently handles operations between arrays of different shapes through broadcasting rules, simplifying code and enhancing flexibility.
7. **Array Manipulation:** Explored functions for concatenating, stacking, splitting, and adding/removing dimensions of arrays.
8. **Linear Algebra Fundamentals:** Learned to perform critical linear algebra operations such as dot product, matrix multiplication, transpose, determinant, inverse, eigenvalues/eigenvectors, and solving linear equations using numpy.linalg.
9. **Statistical Analysis:** Utilized NumPy's built-in functions for calculating common statistics (mean, median, std dev, min, max, sum, prod) on arrays, including operations along specific axes.
10. **Performance Insight:** Gained an appreciation for the significant speedup offered by NumPy's contiguous memory layout and vectorized operations compared to native Python loops, crucial for large-scale engineering and scientific data processing.

By mastering these NumPy fundamentals, you are now well-equipped to efficiently handle and process numerical data, forming the basis for advanced data analysis, scientific simulations, and machine learning tasks in your engineering and scientific endeavors.

ISRO URSC – Python Training | Analog Data | Rajath Kumar

📩 **For queries:** rajath@analogdata.ai| 📱 (+91) 96633 53992 | 🌐 https://analogdata.ai

---
