What is NumPy?

NumPy (Numerical Python) is a powerful Python library used for numerical computations. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy is widely used in scientific computing, data analysis, machine learning, and AI applications.

NumPy is an essential library for numerical computations in Python. Its efficiency, speed, and ease of use make it indispensable for data science, AI, and scientific computing. Whether you're working with large datasets, complex mathematical functions, or machine learning models, NumPy is the go-to tool.

Why Do We Need NumPy?

1. Efficient Computation

- NumPy is significantly faster than Python lists because it uses C and Fortran under the hood.
- It provides optimized vectorized operations that eliminate the need for loops in numerical computations.

2. Memory Efficiency

- NumPy arrays consume less memory compared to Python lists due to their fixed data type and efficient storage.

3. Multi-Dimensional Arrays (ndarray)

- NumPy supports n-dimensional arrays, making it useful for handling matrices and tensor operations.

4. Broadcasting

- It allows arithmetic operations on arrays of different shapes without explicitly reshaping them.

5. Built-in Mathematical Functions

- Includes a vast range of mathematical functions like sin(), cos(), log(), mean(), std(), etc.

6. Integration with Other Libraries

- NumPy is the foundation of many data science and AI libraries, such as Pandas, SciPy, TensorFlow, and PyTorch.

How to Use NumPy?

1. Installation

pip install numpy

2. Importing NumPy

import numpy as np

3. Creating Arrays

```
# Creating a 1D array
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)

# Creating a 2D array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
```

4. Array Properties

```
print(arr2.shape)  # (2, 3) -> Rows, Columns
print(arr2.size)   # Total number of elements
print(arr2.dtype)  # Data type of elements
```

5. Special Arrays

```
np.zeros((3, 3))   # 3x3 matrix filled with zeros
np.ones((2, 2))    # 2x2 matrix filled with ones
np.eye(3)          # Identity matrix of size 3x3
np.arange(0, 10, 2) # Array from 0 to 10 with step 2
np.linspace(1, 5, 10) # 10 values between 1 and 5
```

6. Mathematical Operations

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

print(a + b)  # Element-wise addition
print(a - b)  # Element-wise subtraction
print(a * b)  # Element-wise multiplication
print(a / b)  # Element-wise division
print(np.dot(a, b))  # Dot product
```

7. Array Reshaping

```
arr = np.arange(1, 10)
reshaped = arr.reshape(3, 3)  # Reshapes 1D array into 3x3 matrix
print(reshaped)
```

8. Indexing & Slicing

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[1])    # 20 (Indexing)
print(arr[1:4])  # [20, 30, 40] (Slicing)
```

9. Aggregation Functions

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

print(np.sum(arr))  # Sum of elements
```

```
print(np.mean(arr)) # Mean of elements
print(np.std(arr))  # Standard deviation
print(np.min(arr))  # Minimum value
print(np.max(arr))  # Maximum value
```

10. Random Numbers

```
np.random.rand(3, 3)  # 3x3 matrix of random numbers between 0 and 1
np.random.randint(1, 100, (3, 3))  # 3x3 matrix of random integers from 1 to 100
```

Why Do We Need NumPy Arrays Instead of Python Lists or Standard Python Sequences?

Python lists are flexible and easy to use, but they have significant performance and memory limitations when dealing with large numerical computations. NumPy arrays (ndarray) are optimized for performance, memory efficiency, and numerical operations, making them a superior choice for numerical and scientific computing.

1. Performance: NumPy is Faster than Python Lists

Reason: NumPy Uses Optimized C Implementations

NumPy operations are implemented in **C and Fortran**, which makes them **significantly faster** than Python lists, which are dynamically typed and interpreted at runtime.

Example: Speed Comparison

Let's compare the speed of NumPy arrays and Python lists for an element-wise multiplication operation.

Let's compare the speed of NumPy arrays and Python lists for an element-wise multiplication operation.

```
import numpy as np
import time

# Creating large lists and arrays
size = 10**6
py_list1 = list(range(size))
py_list2 = list(range(size))
np_array1 = np.arange(size)
np_array2 = np.arange(size)

# Timing Python list multiplication
start = time.time()
py_result = [x * y for x, y in zip(py_list1, py_list2)]
end = time.time()
```

```
print("Python List Time:", end - start)

# Timing NumPy array multiplication
start = time.time()
np_result = np_array1 * np_array2  # Vectorized operation
end = time.time()
print("NumPy Array Time:", end – start)
```

Result: NumPy is typically **10-100x faster** than Python lists for large operations.

2. Memory Efficiency: NumPy Uses Less Memory

Reason: NumPy Stores Data More Compactly

Python lists store elements as **objects**, which introduce extra overhead. NumPy arrays store elements as **contiguous blocks of memory with fixed data types**, making them more space-efficient.

Example: Memory Usage Comparison

```
import sys

size = 1000

# Python list memory consumption
py_list = list(range(size))
print("Python List Memory (bytes):", sys.getsizeof(py_list) + sum(sys.getsizeof(i) for i in py_list))

# NumPy array memory consumption
np_array = np.arange(size)
print("NumPy Array Memory (bytes):", np_array.nbytes)
```

Result: NumPy arrays consume significantly **less memory** than Python lists.

3. Broadcasting: Element-wise Operations Without Loops

Reason: NumPy Supports Vectorized Operations

In Python lists, operations require explicit loops or list comprehensions, while NumPy arrays perform operations in a **vectorized manner** (applied to all elements simultaneously).

Example: Python List vs. NumPy Array Operations

```
# Using Python lists (Requires a loop)
py_list = [1, 2, 3, 4, 5]
py_result = [x * 2 for x in py_list]  # Requires explicit iteration

# Using NumPy (No loop required)
np_array = np.array([1, 2, 3, 4, 5])
np_result = np_array * 2  # Vectorized operation
```

Result: NumPy code is **cleaner, shorter, and faster**.

4. Multi-Dimensional Data Handling

Reason: NumPy Supports Multi-Dimensional Arrays (ndarray)

Python lists require **nested lists** to represent matrices, making indexing and operations cumbersome. NumPy provides **n-dimensional arrays (ndarray)**, allowing for **efficient matrix operations**.

Example: 2D Matrix Operations

```
# Python list (Nested list representation)
py_matrix = [[1, 2, 3], [4, 5, 6]]
py_matrix_transpose = [[py_matrix[j][i] for j in range(2)] for i in range(3)]  # Manual transpose

# NumPy (Direct operations)
np_matrix = np.array([[1, 2, 3], [4, 5, 6]])
np_transpose = np_matrix.T  # Transpose
```

Result: NumPy allows **built-in, optimized matrix operations**, avoiding manual loops.

5. Built-in Mathematical Functions

Reason: NumPy Provides Extensive Mathematical Functions

Python lists require manual implementations or math/statistics modules, while NumPy offers **efficient built-in functions**.

Example: Computing Mean and Standard Deviation

```
import statistics

py_list = [1, 2, 3, 4, 5]

# Using Python's statistics module
py_mean = statistics.mean(py_list)
py_std = statistics.stdev(py_list)

# Using NumPy (Optimized)
np_array = np.array([1, 2, 3, 4, 5])
np_mean = np_array.mean()
np_std = np_array.std()

print("Python Mean:", py_mean, " NumPy Mean:", np_mean)
print("Python Std Dev:", py_std, " NumPy Std Dev:", np_std)
```

Result: NumPy is more **efficient** for large datasets.

6. Advanced Operations: Linear Algebra & Random Number Generation

NumPy provides:

- **Linear Algebra** (e.g., matrix multiplication, eigenvalues, determinants)
- **Random Number Generation** (e.g., normal distribution, uniform distribution)
- **Fourier Transforms & Signal Processing**

Example: Matrix Multiplication

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

# Matrix multiplication
C = np.dot(A, B)
print(C)
```

Vectorization in NumPy

Vectorization is a technique in NumPy that allows operations to be applied to entire arrays (vectors) at once, **without the need for explicit loops**. This is possible because NumPy executes operations in **compiled C code** under the hood, making them significantly **faster and more efficient** than using Python loops.

Why Use Vectorization?

1. **Faster Execution:**
   - NumPy operations run in optimized C code, avoiding Python's slow loops.
2. **Simpler Code:**
   - No need for “for” loops or list comprehensions.
3. **Memory Efficient:**
   - NumPy arrays use contiguous memory blocks, reducing overhead.
4. **Parallel Execution:**
   - Takes advantage of **SIMD (Single Instruction Multiple Data)** processing.

Example: Without vs. With Vectorization

Using Python Loops (Slow)

```
import numpy as np
import time

# Creating large arrays
size = 10**6
py_list1 = list(range(size))
py_list2 = list(range(size))
```

```
start = time.time()
result = [x * y for x, y in zip(py_list1, py_list2)]  # Loop-based multiplication
end = time.time()

print("Python Loop Time:", end – start)
```

Using NumPy Vectorization (Fast)

```
# Using NumPy (Vectorized)
np_array1 = np.arange(size)
np_array2 = np.arange(size)

start = time.time()
result = np_array1 * np_array2  # Vectorized multiplication
end = time.time()

print("NumPy Vectorization Time:", end – start)
```

Result: NumPy's vectorized operations can be **10-100x faster** than using Python loops!

Broadcasting in NumPy

What is Broadcasting?

Broadcasting is a feature in **NumPy** that allows operations between arrays of different shapes **without** the need for explicit loops or reshaping. Instead of manually adjusting array dimensions, NumPy **automatically expands** smaller arrays so that element-wise operations can be performed efficiently.

Why is Broadcasting Useful?

Avoids Explicit Loops → Faster execution
Memory Efficient → No unnecessary copies of arrays
Simplifies Code → Cleaner and more readable

Broadcasting Rules

For NumPy to perform broadcasting, it follows **three simple rules** to match array shapes:

- If the dimensions are different, NumPy automatically adds missing dimensions to the smaller array (left-padding with 1s).

- If one dimension is 1, NumPy stretches it to match the other dimension.
- If dimensions are incompatible (neither is 1 and they are different), an error occurs.

Examples of Broadcasting

Scalar and Array Broadcasting

```
import numpy as np

arr = np.array([1, 2, 3])  # Shape: (3,)
scalar = 10  # Shape: ()

result = arr + scalar  # Broadcasting applies here
print(result)  # [11 12 13]
```

NumPy automatically expands scalar **to match** arr**.**
Shape transformation: (3,) + () → (3,)

NumPy Arrays

NumPy arrays (ndarray) are multi-dimensional, fast, and memory-efficient structures used for numerical operations. Let's explore their creation, access, assignment, slicing, and attributes.

1. Creating NumPy Arrays

```
import numpy as np

# 1D Array
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)

# 2D Array (Matrix)
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)

# 3D Array
arr3 = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(arr3)
```

NumPy arrays are more efficient than Python lists for numerical computations.

2. Accessing Elements in NumPy Arrays

1D Array Indexing

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

```
print(arr[0])  # First element → 10
print(arr[-1]) # Last element → 50
```

2D Array Indexing

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

print(arr[0, 0])  # First row, first column → 1
print(arr[1, 2])  # Second row, third column → 6
```

3D Array Indexing

```
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(arr[0, 1, 1])  # First block, second row, second column → 4
```

NumPy allows direct indexing without nested loops!

3. Assigning Values to NumPy Arrays

```
arr = np.array([10, 20, 30])
arr[1] = 99  # Modifies second element
print(arr)  # [10 99 30]
```

NumPy arrays are mutable, meaning values can be changed.

4. Slicing NumPy Arrays

1D Array Slicing

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

print(arr[1:4])  # [20 30 40] (Elements from index 1 to 3)
print(arr[:3])   # [10 20 30] (First three elements)
print(arr[::2])  # [10 30 50] (Every second element)
```

2D Array Slicing

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

print(arr[:2, 1:])  # Extracts first two rows, from second column onward
```

Slicing does not create a new copy but a view! (Changes in slices affect the original array.)

5. Array Attributes

NumPy arrays have various attributes that describe their properties.

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

print(arr.ndim)   # Number of dimensions (2D → 2)
print(arr.shape)  # Shape (Rows, Columns) → (2, 3)
```

```
print(arr.size)   # Total number of elements → 6
print(arr.dtype)  # Data type of elements → int
print(arr.itemsize)  # Memory size of each element (bytes)
```

Attributes help understand the structure and storage details of the array.

6. Array Dimension (ndim)

```
arr1 = np.array([1, 2, 3])  # 1D Array → ndim = 1
arr2 = np.array([[1, 2, 3], [4, 5, 6]])  # 2D Array → ndim = 2
arr3 = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])  # 3D Array → ndim = 3

print(arr1.ndim)  # 1
print(arr2.ndim)  # 2
print(arr3.ndim)  # 3
```

Higher dimensions are useful for machine learning, image processing, and tensor operations.

7. Array Shape (shape)

```
arr1 = np.array([1, 2, 3])  # Shape → (3,)
arr2 = np.array([[1, 2, 3], [4, 5, 6]])  # Shape → (2,3)
arr3 = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])  # Shape → (2,2,2)

print(arr1.shape)  # (3,)
print(arr2.shape)  # (2,3)
print(arr3.shape)  # (2,2,2)
```

Shape represents (rows, columns, depth, etc.) **and helps in reshaping and broadcasting.**

8. Array Size (size)

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

print(arr1.size)  # 3
print(arr2.size)  # 6
```

Size tells the total number of elements present in the array.

9. Changing the Shape of an Array

Using reshape() - Reshape is useful when working with machine learning models that require specific input dimensions.

```
arr = np.arange(1, 10)  # [1 2 3 4 5 6 7 8 9]
```

```
reshaped_arr = arr.reshape(3, 3)  # Converts 1D to 3x3 matrix
print(reshaped_arr)
```

10. Creating Special Arrays

Creating a Zeros Array - Creates an array filled with zeros of a given shape.

```
import numpy as np
zeros_arr = np.zeros((3, 4))  # 3x4 matrix of zeros
print(zeros_arr)
```

11. Creating an Ones Array - Creates an array filled with ones.

```
ones_arr = np.ones((2, 3))  # 2x3 matrix of ones
print(ones_arr)
```

12. Creating an Empty Array - Creates an array with uninitialized values (useful for efficiency).

```
empty_arr = np.empty((2, 2))  # Creates an uninitialized array (values are random)
print(empty_arr)
```

13. Creating Ranges & Linearly Spaced Arrays

Using np.arange() for Range Creation - Creates a sequence from start to end (exclusive) with step.

```
arr = np.arange(1, 10, 2)  # [1 3 5 7 9]
print(arr)
```

Using np.linspace() for Linearly Spaced Values - Creates num evenly spaced values between start and end.

```
arr = np.linspace(0, 10, 5)  # [0. 2.5 5. 7.5 10.]
print(arr)
```

14. Sorting Arrays - Sorts an array in ascending order.

```
arr = np.array([3, 1, 4, 2, 5])
sorted_arr = np.sort(arr)  # [1 2 3 4 5]
print(sorted_arr)
```

15. Concatenating Arrays

Concatenating Along Rows (Axis=0) - Joins two arrays along rows.

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

concat_arr = np.concatenate((arr1, arr2), axis=0)
print(concat_arr)
```

Concatenating Along Columns (Axis=1) - Joins two arrays along columns.

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

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

16. Reshaping Arrays - Reshapes a 1D array into a 2D array.

```
arr = np.arange(6)
reshaped_arr = arr.reshape(2, 3)
print(reshaped_arr)
```

17. Adding a New Dimension

Using np.newaxis - Adds an extra dimension (e.g., converting 1D to 2D).

```
arr = np.array([1, 2, 3])
arr_2d = arr[:, np.newaxis]
print(arr_2d.shape)  # (3, 1)
```

Using np.expand_dims() - Expands the array along a specified axis.

```
arr = np.array([1, 2, 3])
expanded_arr = np.expand_dims(arr, axis=0)
print(expanded_arr.shape)  # (1, 3)
```

18. Slicing Arrays - Extracts part of the array.

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[1:4])  # [20 30 40] (Extract elements 1 to 3)
```

19. Condition-Based Slicing - Selects elements that meet a condition.

```
arr = np.array([10, 20, 30, 40, 50])
filtered_arr = arr[arr > 25]  # [30 40 50]
print(filtered_arr)
```

20. Stacking Arrays

Vertical Stacking (vstack) - Stacks arrays row-wise (vertically).

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

vstacked = np.vstack((arr1, arr2))
print(vstacked)
```

Horizontal Stacking (hstack) - Stacks arrays column-wise (horizontally).

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

hstacked = np.hstack((arr1, arr2))
print(hstacked)
```

21. Splitting Arrays

Horizontal Splitting - Splits an array into multiple sub-arrays along columns.

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

split_arr = np.hsplit(arr, 2)
print(split_arr)
```

22. Views vs Copies in NumPy

View (Shallow Copy) - Changes in a view affect the original array.

```
arr = np.array([1, 2, 3, 4])
view_arr = arr.view()
view_arr[0] = 99  # Modifies original
print(arr)  # [99 2 3 4]
```

Copy (Deep Copy) - A copy creates a separate array.

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

```
copy_arr = arr.copy()
copy_arr[0] = 99  # Does NOT modify original
print(arr)  # [1 2 3 4]
```

23. Basic Arithmetic Operations in NumPy

NumPy allows **element-wise arithmetic operations** on arrays without loops.

Addition of Arrays

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result = arr1 + arr2  # Element-wise addition
print(result)  # [5 7 9]
```

Subtraction of Arrays

```
result = arr1 - arr2  # Element-wise subtraction
print(result)  # [-3 -3 -3]
```

Multiplication of Arrays

```
result = arr1 * arr2  # Element-wise multiplication
print(result)  # [4 10 18]
```

Division of Arrays

```
result = arr1 / arr2  # Element-wise division
print(result)  # [0.25 0.4  0.5]
```

NumPy automatically handles division by zero, returning inf **instead of an error.**

24. Aggregate Functions

NumPy provides **fast** aggregate functions for numerical computations.

Sum of Elements

```
arr = np.array([1, 2, 3, 4, 5])
print(np.sum(arr))  # 15
```

Minimum & Maximum Value

```
print(np.min(arr))  # 1
print(np.max(arr))  # 5
```

Product of All Elements

```
print(np.prod(arr))  # 120 (1*2*3*4*5)
```

Mean (Average)

```
print(np.mean(arr))  # 3.0
```

Standard Deviation (std)

```
print(np.std(arr))  # 1.4142135623730951
```

Standard deviation measures data dispersion.

25. Random Number Generation

NumPy has a built-in random module (np.random).

Generate Random Numbers (0 to 1)

```
rand_arr = np.random.rand(3, 3)  # 3x3 matrix of random numbers between 0 and 1
print(rand_arr)
```

Generate Random Integers

```
rand_int = np.random.randint(1, 100, (3, 3))  # 3x3 matrix with random integers from 1 to 100
print(rand_int)
```

Generate Normally Distributed Random Numbers

```
rand_norm = np.random.randn(5)  # 5 random numbers from normal distribution (mean=0, std=1)
print(rand_norm)
```

Useful for statistics & machine learning.

26. Transposing a Matrix

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
transposed = np.transpose(arr)
print(transposed)
```

Flips the matrix along the diagonal (rows → columns, columns → rows).

27. Reversing & Flipping Arrays

Reverse a 1D Array

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

Reverse Rows in a 2D Array

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
reversed_rows = arr[::-1, :]
print(reversed_rows)
```

Reverse Columns in a 2D Array

```
reversed_cols = arr[:, ::-1]
print(reversed_cols)
```

Flipping rows or columns is useful for image processing and data transformations.

28. Flattening Multidimensional Arrays

Using flatten() - Creates a new 1D array (copy of original).

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

Using ravel() - Returns a flattened view (does not create a copy).

```
flat_arr_ravel = arr.ravel()
print(flat_arr_ravel)  # [1 2 3 4]
```

Important Things to Keep in Mind While Using NumPy & Common Pitfalls

While NumPy is powerful and efficient, there are several things you **must keep in mind** to avoid performance issues, incorrect results, or unexpected behaviour. Here's a list of best practices and common pitfalls to watch out for.

1. Avoid Using Python Loops - Use Vectorization

The Problem: Using Loops for Operations

Using for loops instead of NumPy's **vectorized operations** is **slow and inefficient**.

Incorrect (Using Loops)

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

for i in arr:
   result.append(i * 2)  # Loop-based multiplication

print(result)
```

Correct (Vectorized Operations)

```
result = arr * 2  # Fast and efficient
print(result)
```

Why? NumPy performs **operations in optimized C code**, which is **significantly faster** than Python loops.

2. Be Careful with Data Types (dtype)

The Problem: Implicit Type Conversion

NumPy assigns a **data type (**dtype**) automatically**, but sometimes this can cause issues.

Incorrect (Mismatched Types)

```
arr = np.array([1, 2, 3.5, 4])  # Mixed integer & float
print(arr.dtype)  # float64 (unexpected if you wanted integers)
```

Correct (Explicitly Defining dtype)

```
arr = np.array([1, 2, 3, 4], dtype=np.int32)  # Force integers
print(arr.dtype)  # int32
```

Why? Specifying dtype ensures consistency and avoids unintended float or integer conversions.

3. Be Cautious with Floating Point Precision

The Problem: Precision Errors in Floating-Point Arithmetic

Floating-point numbers can **introduce rounding errors**.

Incorrect (Expecting Exact Equality)

```
a = np.array([0.1, 0.2, 0.3])
print(np.sum(a) == 0.6)  # False (precision issue)
```

Correct (Using np.isclose)

```
print(np.isclose(np.sum(a), 0.6))  # True
```

Why? Floating-point arithmetic is **not always exact**, so use np.isclose() instead of ==.

4. Be Aware of Broadcasting Limitations

The Problem: Incompatible Shapes in Broadcasting

NumPy **broadcasting** allows operations between arrays of different shapes, but sometimes it fails.

Incorrect (Mismatched Shapes)

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])  # Shape (2,3)
arr2 = np.array([10, 20])  # Shape (2,)

result = arr1 + arr2  # ERROR: Shape mismatch
```

Correct (Reshape for Compatibility)

```
arr2 = arr2[:, np.newaxis]  # Convert to shape (2,1)
result = arr1 + arr2  # Now it works!
print(result)
```

Why? Ensure **shapes are compatible** for broadcasting to avoid shape mismatch errors.

5. Avoid Using copy=False Carelessly

The Problem: Modifying an Array by Accident

Using **views instead of copies** can lead to **unexpected modifications**.

Incorrect (Unintended Modification)

```
arr = np.array([1, 2, 3])
view_arr = arr.view()  # Creates a view, not a copy

view_arr[0] = 99  # Changes original array too!
print(arr)  # [99 2 3]
```

Correct (Ensure a Copy is Created)

```
copy_arr = arr.copy()  # Creates an independent copy
copy_arr[0] = 99

print(arr)  # [1 2 3] (Original remains unchanged)
```

Why? If you don't want changes in one array to affect another, always use copy().

6. Avoid Memory Overhead with Large Arrays

The Problem: Creating Huge Arrays Can Crash Your System

NumPy can allocate **very large arrays**, leading to memory overflow.

Incorrect (Large Memory Allocation)

```
huge_arr = np.zeros((100000, 100000))  # May crash!
```

Correct (Use Memory Efficient Methods)

```
huge_arr = np.zeros((10000, 10000), dtype=np.float32)  # Use smaller `dtype`
```

Why? Optimize memory by using smaller dtype like float32 instead of float64.

7. Use Boolean Masking Instead of Loops for Filtering

The Problem: Slow Filtering with Loops

Using **loops** for conditional selection is **inefficient**.

Incorrect (Using Loops for Filtering)

```
arr = np.array([10, 20, 30, 40, 50])
result = [x for x in arr if x > 25]  # Slow
print(result)
```

Correct (Using Boolean Masking)

```
result = arr[arr > 25]  # Fast and efficient
print(result)  # [30 40 50]
```

Why? Boolean masking is much **faster than loops**.

8. Be Aware of np.empty() Behavior

The Problem: np.empty() Does Not Initialize Values

Using np.empty() does **not** fill the array with zeros.

Incorrect (Expecting Zeros)

```
arr = np.empty((2, 3))
print(arr)  # Contains random uninitialized values
```

Correct (Use np.zeros() If You Need Zeros)

```
arr = np.zeros((2, 3))  # Explicitly initialize with zeros
```

Why? np.empty() is for efficiency, not for initializing values.

9. Avoid Modifying Arrays During Iteration

The Problem: Changing an Array While Iterating Causes Issues

If you **modify an array** inside a loop, it may lead to **unexpected results**.

Incorrect (Modifying While Iterating)

```
arr = np.array([1, 2, 3, 4])
for i in arr:
   i *= 2  # Does not modify the original array

print(arr)  # [1 2 3 4] (No change)
```

Correct (Use Vectorized Operations)

```
arr = arr * 2
print(arr)  # [2 4 6 8] (Correct result)
```

Why? Direct assignments inside loops **do not modify** the array in-place.

10. Be Careful When Using np.append()

The Problem: np.append() is Slow for Large Arrays

Appending elements in NumPy **creates a new array** every time, making it inefficient.

Incorrect (Repeated np.append())

```
arr = np.array([1, 2, 3])
for i in range(10000):
   arr = np.append(arr, i)  # Slow!
```

Correct (Use np.concatenate() or Lists)

```
arr = np.array([1, 2, 3])
arr = np.concatenate([arr, np.arange(10000)])
```

Why? np.concatenate() **is more efficient** than multiple np.append() calls.

Summary: Key Takeaways

- Use vectorized operations instead of loops
- Specify dtype explicitly when needed
- Use np.isclose() instead of == for floating-point comparisons
- Check array shapes when broadcasting
- Use .copy() if you need an independent array
- Optimize memory usage with smaller dtype
- Use Boolean indexing instead of loops
- Understand np.empty() does not initialize values
- Avoid modifying arrays while iterating
- Use np.concatenate() instead of repeated np.append()

Where Not to Use NumPy?

While **NumPy** is a powerful tool for numerical computations, there are cases where using NumPy **is not the best choice**. Below are scenarios where NumPy should **not be used**, along with better alternatives.

1. Small Data or Simple Lists

The Problem: Overhead of NumPy for Small Data

NumPy is optimized for **large numerical computations**, but for **small lists**, the overhead of importing and using NumPy is unnecessary.

Using NumPy for Small Lists

```
import numpy as np
arr = np.array([1, 2, 3])  # Unnecessary for small lists
print(arr[1])  # Accessing elements
```

Better Alternative: Python Lists

```
lst = [1, 2, 3]  # Simple and memory efficient
print(lst[1])
```

Why? Python lists are **more efficient** for small datasets because NumPy **introduces additional overhead**.

2. Non-Numerical Data Processing

The Problem: NumPy is Built for Numbers, Not Strings

NumPy is **not** designed for handling **strings, objects, or mixed data types** efficiently.

Using NumPy for Strings

```
import numpy as np
arr = np.array(["apple", "banana", "cherry"])  # Strings in NumPy (inefficient)
print(arr.dtype)  # dtype='<U6'
```

Better Alternative: Python Lists or Pandas

```
fruits = ["apple", "banana", "cherry"]  # Use a simple list for strings
```

Why? NumPy arrays are **optimized for numerical data**, while lists and Pandas **handle text better**.

3. Dynamic or Growing Arrays

The Problem: NumPy Arrays Have Fixed Sizes

NumPy arrays are **static** in size, meaning they are **inefficient for dynamic resizing**.

Using NumPy for Dynamic Lists

```
import numpy as np
arr = np.array([1, 2, 3])
arr = np.append(arr, [4, 5, 6])  # Inefficient for large-scale append operations
```

Better Alternative: Python Lists

```
lst = [1, 2, 3]
lst.append(4)  # Fast and efficient
```

Why? Python lists **grow dynamically**, while NumPy **creates a new array** every time you append.

4. Deep Learning & Complex Neural Networks

The Problem: NumPy Lacks GPU Support

For deep learning, NumPy does **not** utilize **GPU acceleration** or automatic differentiation.

Using NumPy for Deep Learning

```
import numpy as np
arr = np.random.rand(1000, 1000)  # Large matrix, but no GPU acceleration
```

Better Alternative: Use TensorFlow or PyTorch

```
import torch
tensor = torch.rand(1000, 1000).cuda()  # Uses GPU acceleration
```

Why? PyTorch and TensorFlow **support GPUs and are optimized for deep learning**.

5. Handling Large Datasets That Don't Fit in Memory

The Problem: NumPy Loads Everything into RAM

NumPy **loads entire datasets into memory**, which can cause **memory overflow** for very large datasets.

Using NumPy for Large Datasets

```
import numpy as np
large_arr = np.random.rand(100000000)  # Consumes a lot of RAM!
```

Better Alternative: Use Pandas, Dask, or Vaex

```
import dask.array as da
large_arr = da.random.random(100000000)  # Uses disk-based computation
```

Why? Dask and **Vaex** can handle large datasets by **processing them in chunks** instead of loading them into memory.

6. High-Performance Computing with Multi-Core Processing

The Problem: NumPy is Single-Threaded for Most Operations

While NumPy **supports multi-threading**, many operations **run in a single core**, making it **suboptimal for parallel computing**.

Using NumPy for Parallel Processing

```
import numpy as np
arr = np.arange(1000000)
result = np.sin(arr)  # Mostly runs on a single CPU core
```

Better Alternative: Use Numba or Dask

```
from numba import jit
import numpy as np

@jit(nopython=True)
def compute(arr):
    return np.sin(arr)

arr = np.arange(1000000)
result = compute(arr)  # Uses multiple CPU cores
```

Why? Numba and **Dask** offer better performance for multi-core computing.

7. Image Processing (Without Specialized Libraries)

The Problem: NumPy Lacks Specialized Image Processing Functions

NumPy **can** store and manipulate image data, but **it does not offer specialized functions** like filtering, edge detection, or transformations.

Using NumPy for Image Processing

```
import numpy as np
image = np.zeros((256, 256, 3))  # Stores image but lacks processing functions
```

Better Alternative: Use OpenCV or PIL

```
import cv2
image = cv2.imread("image.jpg")  # Reads image efficiently
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  # Converts to grayscale
```

Why? OpenCV and **PIL** provide specialized tools for image manipulation.

8. Handling SQL-Like Data (Tables, Relational Data)

The Problem: NumPy Lacks Database Functionality

NumPy **is not designed** for working with **tabular, structured, or relational data**.

Using NumPy for DataFrames

```
import numpy as np
data = np.array([["Alice", 25], ["Bob", 30], ["Charlie", 22]])
```

Better Alternative: Use Pandas

```
import pandas as pd
data = pd.DataFrame({"Name": ["Alice", "Bob", "Charlie"], "Age": [25, 30, 22]})
print(data)
```

Why? Pandas provides **faster indexing, filtering, and manipulation** for tabular data.

9. Object-Oriented Programming (OOP) & Complex Data Structures

The Problem: NumPy Arrays Do Not Support Complex Objects Well

NumPy **only supports homogeneous data types**, making it **difficult to store complex objects**.

Using NumPy for Object-Oriented Programming

```
import numpy as np
class Person:
```

```
    def __init__(self, name, age):
        self.name = name
        self.age = age

arr = np.array([Person("Alice", 25), Person("Bob", 30)])  # Works but inefficient
```

Better Alternative: Use Lists or Pandas

```
people = [Person("Alice", 25), Person("Bob", 30)]
```

Why? Python lists and **Pandas DataFrames** are **better for heterogeneous data**.