Why Use NumPy?

Python lists are flexible but **slow** for numerical computing because they:

- Store elements as **pointers** instead of a continuous block of memory.
- Lack **vectorized operations**, relying on loops instead.
- Have significant **overhead** due to dynamic typing.

NumPy's Superpowers:

- **Faster** than Python lists (C-optimized backend)
- **Uses less memory** (efficient storage)
- **Supports vectorized operations** (no explicit loops needed)
- **Has built-in mathematical functions**

NumPy vs. Python Lists – Performance Test

Let's compare Python lists with NumPy arrays using a simple example.

Example 1: Adding Two Lists vs. NumPy Arrays

```python
import numpy as np
import time

# Python list
size = 1_000_000
list1 = list(range(size))
list2 = list(range(size))

start = time.time()
result = [x + y for x, y in zip(list1, list2)]
end = time.time()
```

```python
print("Python list addition time:", end - start)

# NumPy array
arr1 = np.array(list1)
arr2 = np.array(list2)

start = time.time()
result = arr1 + arr2  # Vectorized operation
end = time.time()
print("NumPy array addition time:", end - start)
```

Key Takeaway: NumPy is significantly **faster** because it performs operations in **C**, avoiding Python loops.

Creating NumPy Arrays

```python
import numpy as np

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

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

# Checking type and shape
print("Type:", type(arr1))
print("Shape:", arr2.shape)
```

◇ **NumPy stores data in a contiguous memory block**, making access faster than lists. ◇ `shape` shows the **dimensions** of an array.

Memory Efficiency – NumPy vs. Lists

Let’s check memory consumption.

```
import sys

list_data = list(range(1000))
numpy_data = np.array(list_data)

print("Python list size:", sys.getsizeof(list_data) * len(list_data), "bytes")
print("NumPy array size:", numpy_data.nbytes, "bytes")
```

NumPy arrays use significantly less memory compared to Python lists.

Vectorization – No More Loops!

NumPy avoids loops by applying operations to entire arrays at once using SIMD (Single Instruction, Multiple Data) and other low-level optimizations. SIMD is a CPU-level optimization provided by modern processors.

Example 2: Squaring Elements

```
# Python list (loop-based)
list_squares = [x ** 2 for x in list1]

# NumPy (vectorized)
numpy_squares = arr1 ** 2
```

- NumPy is **cleaner** and **faster**!

Summary

- NumPy is **faster** than Python lists because it is optimized in **C**.

- It consumes **less memory** due to efficient storage.
- It provides **vectorized operations**, removing the need for slow loops.
- Essential for **data science** and **machine learning** workflows.

Exercises for Practice

- Create a NumPy array with values from **10 to 100** and print its shape.
- Compare the time taken to multiply **two Python lists** vs. **two NumPy arrays**.
- Find the **memory size** of a NumPy array with **1 million elements**.

Creating NumPy Arrays

Why NumPy Arrays?

NumPy arrays are the **core** of numerical computing in Python. They are:

- **Faster** than Python lists (C-optimized)
- **Memory-efficient** (store data in a contiguous block)
- **Support vectorized operations that support SIMD** (no slow Python loops)
- **Used in ML, Data Science, and AI**

---

1. Creating NumPy Arrays

From Python Lists:

```
import numpy as np


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

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

```
print(arr1)  # [1 2 3 4 5]
print(arr2)
# [[1 2 3]
#  [4 5 6]]
```

◇ Unlike lists, all elements must have the same data type.

Creating Arrays from Scratch:

```
np.zeros((3, 3))    # 3x3 array of zeros
np.ones((2, 4))     # 2x4 array of ones
np.full((2, 2), 7)  # 2x2 array filled with 7
np.eye(4)           # 4x4 identity matrix
np.arange(1, 10, 2) # [1, 3, 5, 7, 9] (like range)
np.linspace(0, 1, 5) # [0. 0.25 0.5 0.75 1.] (evenly spaced)
```

Key Takeaway: NumPy offers powerful shortcuts to create arrays **without loops!**

---

2. Checking Array Properties

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

print("Shape:", arr.shape)   # (2, 3) → 2 rows, 3 columns
print("Size:", arr.size)     # 6 → total elements
print("Dimensions:", arr.ndim) # 2 → 2D array
print("Data type:", arr.dtype) # int64 (or int32 on Windows)
```

◇ **NumPy arrays are strongly typed**, meaning all elements share the same data type.

---

3. Changing Data Types

```
arr = np.array([1, 2, 3], dtype=np.float32)  # Explicit type
print(arr.dtype)  # float32

arr_int = arr.astype(np.int32)  # Convert float to int
print(arr_int)  # [1 2 3]
```

- **Efficient memory usage** by choosing the right data type.

---

4. Reshaping and Flattening Arrays

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

reshaped = arr.reshape((3, 2))  # Change shape
print(reshaped)
# [[1 2]
#  [3 4]
#  [5 6]]

flattened = arr.flatten()  # Convert 2D → 1D
print(flattened)  # [1 2 3 4 5 6]
```

Indexing and slicing

Lets now learn about indexing and slicing in Numpy

Indexing (Same as Python Lists)

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

Slicing (Extracting Parts of an Array)

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

print(arr[1:4])  # [20 30 40] (slice from index 1 to 3)
print(arr[:3])   # [10 20 30] (first 3 elements)
print(arr[::2])  # [10 30 50] (every 2nd element)
```

Slicing returns a view, not a copy! Changes affect the original array.**

This might seem counterintuitive since Python lists create copies when sliced. But in NumPy, slicing returns a view of the original array. Both the sliced array and the original array share the same data in memory, so changes in the slice affect the original array.

Why does this happen?

- Memory Efficiency: Avoids unnecessary copies, making operations faster and saving memory.
- Performance: Enables faster access and manipulation of large datasets without duplicating data.

```
sliced = arr[1:4]
sliced[0] = 999
print(arr)  # [10 999 30 40 50]
```

- Use `.copy()` if you need an independent copy.

---

6. Fancy Indexing & Boolean Masking

Fancy Indexing (Select Multiple Elements)

```
arr = np.array([10, 20, 30, 40, 50])
idx = [0, 2, 4]  # Indices to select
print(arr[idx])  # [10 30 50]
```

Boolean Masking (Filter Data)

```
arr = np.array([10, 20, 30, 40, 50])
mask = arr > 25  # Condition: values greater than 25
print(arr[mask])  # [30 40 50]
```

This is a powerful way to filter large datasets efficiently!

Summary

- **NumPy arrays** are faster, memory-efficient alternatives to lists.
- You can create arrays using `np.array()` , `np.zeros()` , `np.ones()` , etc.
- **Indexing & slicing** allow efficient data manipulation.
- **Reshaping & flattening** change array structures without copying data.
- **Fancy indexing & boolean masking** help filter and access specific data.

Exercises for Practice

- Create a **3×3** array filled with random numbers and print its shape.
- Convert an array of floats `[1.1, 2.2, 3.3]` into integers.
- Use **fancy indexing** to extract **even numbers** from `[1, 2, 3, 4, 5, 6]` .
- Reshape a **1D array of size 9** into **a 3×3 matrix**.

- Use **boolean masking** to filter numbers **greater than 50** in an array.

Multidimensional Indexing and Axis

NumPy allows you to efficiently work with **multidimensional arrays**, where indexing and axis manipulation play a crucial role. Understanding how indexing works across multiple dimensions is essential for data science and machine learning tasks.

---

1. Understanding Axes in NumPy

Each dimension in a NumPy array is called an **axis**. Axes are numbered starting from **0**.

For example:

- **1D array** → 1 axis (axis 0)
- **2D array** → 2 axes (axis 0 = rows, axis 1 = columns)
- **3D array** → 3 axes (axis 0 = depth, axis 1 = rows, axis 2 = columns)

Example: Axes in a 2D Array

```
import numpy as np

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

print(arr)
```

Output:

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

- **Axis 0 (rows)** → Operations move **down** the columns.
- **Axis 1 (columns)** → Operations move **across** the rows.

Summing along axes:

```
print(np.sum(arr, axis=0))  # Sum along rows (down each column)
print(np.sum(arr, axis=1))  # Sum along columns (across each row)
```

Output:

```
[12 15 18]  # Column-wise sum
[ 6 15 24]  # Row-wise sum
```

---

2. Indexing in Multidimensional Arrays

You can access elements using **row and column indices**.

```
# Accessing an element
print(arr[1, 2])  # Row index 1, Column index 2 → Output: 6
```

You can also use **slicing** to extract parts of an array:

```
print(arr[0:2, 1:3])  # Extracts first 2 rows and last 2 columns
```

Output:

```
[[2 3]
 [5 6]]
```

3. Indexing in 3D Arrays

For **3D arrays**, the first index refers to the "depth" (sheets of data).

```
arr3D = np.array([[[1, 2, 3], [4, 5, 6]],
                  [[7, 8, 9], [10, 11, 12]]])

# Output of arr3D.shape is → (depth, rows, columns)
print(arr3D.shape)  # Output: (2, 2, 3)
```

Accessing elements in 3D:

```
# First sheet, second row, third column
print(arr3D[0, 1, 2])  # Output: 6

print(arr3D[:, 0, :])   # Get the first row from both sheets
```

---

4. Practical Example: Selecting Data Along Axes

```
# Get all rows of the first column
first_col = arr[:, 0]
print(first_col)  # Output: [1 4 7]
```

```
# Get the first row from each "sheet" in a 3D array
first_rows = arr3D[:, 0, :]
print(first_rows)
```

Output:

```
[[ 1  2  3]
 [ 7  8  9]]
```

5. Changing Data Along an Axis

```
# Replace all elements in column 1 with 0
arr[:, 1] = 0
print(arr)
```

Output:

```
[[1 0 3]
 [4 0 6]
 [7 0 9]]
```

6. Summary

- Axis 0 = rows (vertical movement), Axis 1 = columns (horizontal movement)
- Indexing works as `arr[row, column]` for 2D arrays and `arr[depth, row, column]` for 3D arrays
- Slicing allows extracting subarrays
- Operations along axes help efficiently manipulate data without loops

Data Types in NumPy

Let's learn about NumPy's **data types** and explore how they affect memory usage and performance in your arrays.

1. Introduction to NumPy Data Types

NumPy arrays are **homogeneous**, meaning that they can only store elements of the same type. This is different from Python lists, which can hold mixed data types. NumPy supports various **data types** (also called **dtypes**), and understanding them is crucial for optimizing memory usage and performance.

Common Data Types in NumPy:

- `int32`, `int64`: Integer types with different bit sizes.
- `float32`, `float64`: Floating-point types with different precision.
- `bool`: Boolean data type.
- `complex64`, `complex128`: Complex number types.
- `object`: For storing objects (e.g., Python objects, strings).

You can check the dtype of a NumPy array using the `.dtype` attribute.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr.dtype)  # Output: int64 (or int32 depending on the system)
```

---

2. Changing Data Types

You can **cast** (convert) the data type of an array using the `.astype()` method. This is useful when you need to change the type for a specific operation or when you want to reduce memory usage.

Example: Changing Data Types

```
arr = np.array([1.5, 2.7, 3.9])
print(arr.dtype)  # Output: float64

arr_int = arr.astype(np.int32)  # Converting float to int
print(arr_int)     # Output: [1 2 3]
print(arr_int.dtype)  # Output: int32
```

Example: Downcasting to Save Memory

```
arr_large = np.array([1000000, 2000000, 3000000], dtype=np.int64)
arr_small = arr_large.astype(np.int32)  # Downcasting to a smaller dtype
```

```
print(arr_small)  # Output: [1000000 2000000 3000000]
print(arr_small.dtype)  # Output: int32
```

---

3. Why Data Types Matter in NumPy

The choice of data type affects: - **Memory Usage**: Smaller data types use less memory. - **Performance**: Operations on smaller data types are faster due to less data being processed. - **Precision**: Choosing the appropriate data type ensures that you don't lose precision (e.g., using `float32` instead of `float64` if you don't need that extra precision).

Example: Memory Usage

```
arr_int64 = np.array([1, 2, 3], dtype=np.int64)
arr_int32 = np.array([1, 2, 3], dtype=np.int32)

print(arr_int64.nbytes)  # Output: 24 bytes (3 elements * 8 bytes each)
print(arr_int32.nbytes)  # Output: 12 bytes (3 elements * 4 bytes each)
```

---

4. String Data Type in NumPy

Although NumPy arrays typically store numerical data, you can also store strings by using the `dtype='str'` or `dtype='U'` (Unicode string) format. However, working with strings in NumPy is **less efficient** than using lists or Python's built-in string types.

Example: String Array

```
arr = np.array(['apple', 'banana', 'cherry'], dtype='U10')  # Unicode string array
print(arr)
```

---

5. Complex Numbers

NumPy also supports **complex numbers**, which consist of a real and imaginary part. You can store complex numbers using `complex64` or `complex128` data types.

Example: Complex Numbers

```
arr = np.array([1 + 2j, 3 + 4j, 5 + 6j], dtype='complex128')
print(arr)
```

---

6. Object Data Type

If you need to store mixed or complex data types (e.g., Python objects), you can use `dtype='object'`. However, this type sacrifices performance, so it should only be used when absolutely necessary.

Example: Object Data Type

```
arr = np.array([{'a': 1}, [1, 2, 3], 'hello'], dtype=object)
print(arr)
```

---

7. Choosing the Right Data Type

Choosing the correct data type is essential for: - **Optimizing memory**: Using the smallest data type that fits your data. - **Improving performance**: Smaller types generally lead to faster operations. - **Ensuring precision**: Avoid truncating or losing important decimal places or values.

Summary:

- **NumPy arrays are homogeneous**, meaning all elements must be of the same type.
- Use `.astype()` to **change data types** and optimize memory and performance.

- The choice of data type affects **memory usage**, **performance**, and **precision**.
- Be mindful of **complex numbers** and **object data types**, which can increase memory usage and reduce performance.

Broadcasting in NumPy

Now, we'll explore how to make your code faster with **vectorization** and **broadcasting** in NumPy. These techniques are key to boosting performance in numerical operations by avoiding slow loops and memory inefficiency.

1. Why Loops Are Slow

In Python, loops are typically slow because: - **Python's interpreter**: Every iteration of the loop requires Python to interpret the loop logic, which is inherently slower than lower-level, compiled code. - **High overhead**: Each loop iteration in Python involves additional overhead for function calls, memory access, and index management.

While Python loops are convenient, they don't take advantage of the **optimized memory and computation** that libraries like **NumPy** provide.

Example: Looping Over Arrays in Python

```python
import numpy as np

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

# Using a loop to square each element (slow)
for num in arr:
    result.append(num ** 2)

print(result)  # Output: [1, 4, 9, 16, 25]
```

This works, but it's not efficient. Each loop iteration is slow, especially with large datasets.

2. Vectorization: Fixing the Loop Problem

Vectorization allows you to perform operations on entire arrays **at once**, instead of iterating over elements one by one. This is made possible by **NumPy's optimized C-based backend** that executes operations in compiled code, which is much faster than Python loops.

Vectorized operations are also **more readable** and compact, making your code easier to maintain.

Example: Vectorized Operation

```
arr = np.array([1, 2, 3, 4, 5])
result = arr ** 2  # Vectorized operation
print(result)  # Output: [1 4 9 16 25]
```

Here, the operation is applied to all elements of the array simultaneously, and it's much faster than looping over the array.

Why is it Faster?

- **Low-level implementation**: NumPy's vectorized operations are implemented in **C** (compiled language), which is much faster than Python loops.
- **Batch processing**: NumPy processes multiple elements in parallel using **SIMD** (Single Instruction, Multiple Data), allowing multiple operations to be done simultaneously.

---

3. Broadcasting: Scaling Arrays Without Extra Memory

Broadcasting is a powerful feature of NumPy that allows you to perform operations on arrays of different shapes without creating copies. It "stretches" smaller arrays across larger arrays in a memory-efficient way, avoiding the overhead of creating multiple copies of data.

Example: Broadcasting with Scalar

Broadcasting is often used when you want to perform an operation on an array and a scalar value (e.g., add a number to all elements of an array).

```python
arr = np.array([1, 2, 3, 4, 5])
result = arr + 10  # Broadcasting: 10 is added to all elements
print(result)  # Output: [11 12 13 14 15]
```

Here, the scalar `10` is "broadcast" across the entire array, and no extra memory is used.

---

4. Broadcasting with Arrays of Different Shapes

Broadcasting becomes more powerful when you apply operations on arrays of **different shapes**. NumPy automatically adjusts the shapes of arrays to make them compatible for element-wise operations, without actually copying the data.

Example: Broadcasting with Two Arrays

```python
arr1 = np.array([1, 2, 3])
arr2 = np.array([10, 20, 30])

result = arr1 + arr2  # Element-wise addition
print(result)  # Output: [11 22 33]
```

NumPy automatically aligns the two arrays and performs element-wise addition, treating them as if they have the same shape.

Example: Broadcasting a 2D Array and a 1D Array

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

result = arr1 + arr2  # Broadcasting arr2 across arr1
print(result)
# Output:
# [[2 4 6]
#  [5 7 9]]
```

In this case, `arr2` is broadcast across the rows of `arr1`, adding `[1, 2, 3]` to each row.

How Broadcasting Works

1. **Dimensions must be compatible**: The size of the trailing dimensions of the arrays must be either the same or one of them must be 1.
2. **Stretching arrays**: If the shapes are compatible, NumPy stretches the smaller array to match the larger one, element-wise, without copying data.

---

5. Hands-on: Applying Broadcasting to Real-World Scenarios

Let's apply broadcasting to a real-world scenario: **scaling data** in machine learning.

Example: Normalizing Data Using Broadcasting

Imagine you have a dataset where each row represents a sample and each column represents a feature. You can **normalize** the data by subtracting the mean of each column and dividing by the standard deviation.

```
# Simulating a dataset (5 samples, 3 features)
data = np.array([[10, 20, 30],
                 [15, 25, 35],
                 [20, 30, 40],
                 [25, 35, 45],
                 [30, 40, 50]])

# Calculating mean and standard deviation for each feature (column)
mean = data.mean(axis=0)
std = data.std(axis=0)

# Normalizing the data using broadcasting
normalized_data = (data - mean) / std

print(normalized_data)
```

In this case, broadcasting allows you to subtract the mean and divide by the standard deviation for each feature without needing loops or creating copies of the data.

Summary:

- **Loops are slow** because Python's interpreter adds overhead, making iteration less efficient.
- **Vectorization** allows you to apply operations to entire arrays at once, greatly improving performance by utilizing NumPy's optimized C backend.
- **Broadcasting** enables operations between arrays of different shapes by automatically stretching the smaller array to match the shape of the larger array, without creating additional copies.
- **Real-world use**: Broadcasting can be used in data science tasks, such as **normalizing datasets**, without sacrificing memory or performance.

Built in Mathematical Functions in NumPy

Here are some common NumPy methods that are frequently used for statistical and mathematical operations:

1. `np.mean()` – Compute the **mean** (average) of an array.

   ```
   np.mean(arr)
   ```

2. `np.std()` – Compute the **standard deviation** of an array.

   ```
   np.std(arr)
   ```

3. `np.var()` – Compute the **variance** of an array.

```
np.var(arr)
```

4. **`np.min()`** – Compute the **minimum** value of an array.

```
np.min(arr)
```

5. **`np.max()`** – Compute the **maximum** value of an array.

```
np.max(arr)
```

6. **`np.sum()`** – Compute the **sum** of all elements in an array.

```
np.sum(arr)
```

7. **`np.prod()`** – Compute the **product** of all elements in an array.

```
np.prod(arr)
```

8. **`np.median()`** – Compute the **median** of an array.

```
np.median(arr)
```

9. **`np.percentile()`** – Compute the **percentile** of an array.

```
np.percentile(arr, 50)  # For the 50th percentile (median)
```

10. **`np.argmin()`** – Return the **index of the minimum** value in an array.

```
np.argmin(arr)
```

11. **`np.argmax()`** – Return the **index of the maximum** value in an array.

```
np.argmax(arr)
```

12. `np.corrcoef()` – Compute the **correlation coefficient** matrix of two arrays.

```
np.corrcoef(arr1, arr2)
```

13. `np.unique()` – Find the **unique elements** of an array.

```
np.unique(arr)
```

14. `np.diff()` – Compute the **n-th differences** of an array.

```
np.diff(arr)
```

15. `np.cumsum()` – Compute the **cumulative sum** of an array.

```
np.cumsum(arr)
```

16. `np.linspace()` – Create an array with **evenly spaced numbers** over a specified interval.

```
np.linspace(0, 10, 5)  # 5 numbers from 0 to 10
```

17. `np.log()` – Compute the **natural logarithm** of an array.

```
np.log(arr)
```

18. `np.exp()` – Compute the **exponential** of an array.

```
np.exp(arr)
```

These methods are used for performing mathematical and statistical operations with NumPy