
🔢 NumPy Array vs Python List

⚖️ Comparison: Memory Usage & Speed Performance

🔹 1. Memory Usage

NumPy arrays use less memory than Python lists because they store data more efficiently using fixed-size data types.

✅ **Example Code (Memory Size):**

```
import numpy as np

import sys


py_list = list(range(1000))

np_array = np.arange(1000)


print("Python List size:", sys.getsizeof(py_list))      # Size of list container only

print("NumPy Array size:", np_array.nbytes)          # Total size in bytes
```

✅ **Output (approximate):**

Python List size: 9000+ bytes

NumPy Array size: 8000 bytes

📌 **Why?**

- Python lists store references (pointers) to each element.
- NumPy arrays store data in **contiguous memory blocks** of a fixed type (int32, float64, etc.).

🔹 2. Speed (Performance)

NumPy arrays are much faster than Python lists for numerical operations due to internal implementation in C and use of vectorization.

✅ **Example Code (Execution Time):**

```python
import time

import numpy as np


size = 1000000


# Python list

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)]

print("Python List Time:", time.time() - start)


# NumPy array

np_array1 = np.arange(size)

np_array2 = np.arange(size)


start = time.time()

result = np_array1 + np_array2

print("NumPy Array Time:", time.time() - start)
```

✅ Output (approximate):

Python List Time: 0.25 seconds

NumPy Array Time: 0.01 seconds

📌 Why?

- NumPy uses **vectorized operations** and **precompiled C code**.
- Python lists use a **loop in Python**, which is slower.

🔚 Summary Table:

Feature	Python List	NumPy Array
Memory Usage	More (dynamic & pointers)	Less (fixed-type, efficient)
Speed	Slower (loops)	Faster (vectorized ops)
Data Type	Mixed types allowed	Same data type required
Functionality	General-purpose	Scientific & numeric

🗂️ NumPy Advanced Indexing

=================================

NumPy provides two powerful techniques for advanced data selection:

✅ **Fancy Indexing**
✅ **Boolean Indexing**

🔹 **1. Fancy Indexing**

Fancy indexing allows you to pass **a list or array of indices** to access multiple elements at once.

✅ **Example 1: 1D Array**

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

indices = [0, 2, 4]

result = arr[indices]

Output:

[10, 30, 50]

You can also use a NumPy array of indices:

arr[np.array([1, 3])]

→ [20, 40]

✅ **Example 2: 2D Array (Rows and Columns)**

arr = np.array([[10, 11], [20, 21], [30, 31]])

rows = [0, 1, 2]

cols = [1, 0, 1]

result = arr[rows, cols]

Output:

[11, 20, 31]

It selects:

- (0,1) → 11
- (1,0) → 20
- (2,1) → 31

🔹 2. Boolean Indexing

Use a **Boolean array** (same shape) to filter values based on a condition.

✅ **Example: 1D Array**

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

mask = arr > 10

result = arr[mask]

Output:

[15, 20]

Can also be written in one line:

arr[arr > 10] → [15, 20]

✅ **Example: 2D Array**

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

result = arr[arr > 3]

Output:

[4, 5, 6]

Returns a **flattened 1D array** of all elements greater than 3.

✅ **Combine Conditions with Boolean Operators:**

arr[(arr > 3) & (arr < 6)] → [4, 5]

arr[(arr == 2) | (arr == 6)] → [2, 6]

Use **& (and), | (or), ~ (not)** with parentheses.

🔚 **Summary:**

Feature	**Fancy Indexing**	**Boolean Indexing**
Type of index	List/array of integers	Boolean array or condition result
Output shape	Depends on indices	1D array of matching elements
Usage	Select specific positions	Filter based on condition

🗞 NumPy Broadcasting

================================

🔹 **What is Broadcasting?**

Broadcasting is a feature in NumPy that allows **arithmetic operations** between arrays of **different shapes** without explicitly reshaping or replicating data.

It automatically **expands smaller arrays** so they match the shape of larger arrays, **without copying data**.

🔹 **Why Use Broadcasting?**

✅ Avoids explicit looping or reshaping
✅ Saves memory and improves performance
✅ Makes code cleaner and shorter

🔹 **Broadcasting Example:**

a = np.array([1, 2, 3])   # Shape (3,)

b = np.array([[10], [20]])   # Shape (2, 1)

result = b + a

Output:

[[11, 12, 13],

 [21, 22, 23]]

Here:

- a becomes shape (1, 3)
- b becomes shape (2, 1)
- Result shape → (2, 3)

🔹 Broadcasting Rules

To apply broadcasting, NumPy compares the **shapes** of the arrays **from right to left** (trailing dimensions).

✅ **Rule 1:**

If the two dimensions are **equal**, they're compatible.

✅ **Rule 2:**

If one of the dimensions is **1**, it's stretched to match the other.

✅ **Rule 3:**

If the dimensions are **not equal and neither is 1**, broadcasting **fails**.

🔹 Broadcasting Examples

✅ **Example 1: Compatible Shapes**

```
a = np.array([1, 2, 3])       # Shape (3,)

b = np.array([[10], [20]])    # Shape (2, 1)
```

→ Result shape: (2, 3)

✅ **Example 2: Scalar and Array**

```
a = np.array([[1, 2], [3, 4]])   # Shape (2, 2)

b = 5                        # Shape () – scalar
```

→ b is broadcast to shape (2, 2)

→ Result = a + b = [[6, 7], [8, 9]]

✅ **Example 3: Fails (Incompatible Shapes)**

a = np.array([1, 2, 3])   # Shape (3,)

b = np.array([[1, 2], [3, 4]]) # Shape (2, 2)

→ Broadcasting fails because (3,) and (2, 2) are not compatible

---

🔹 **Broadcasting Table (Shape Comparison Right to Left)**

Operand A Shape	Operand B Shape	Broadcasted Shape	Valid?
(4, 3)	(3,)	(4, 3)	✅ Yes
(2, 1)	(2, 3)	(2, 3)	✅ Yes
(1, 5)	(4, 1)	(4, 5)	✅ Yes
(3, 4)	(2, 4)	❌ –	❌ No
(1, 1, 3)	(2, 3)	(1, 2, 3)	✅ Yes

🔚 **Summary**

- Broadcasting lets you work with arrays of different shapes.
- Follows specific shape compatibility rules.
- Saves memory and improves code readability.
- If broadcasting fails, use reshape() or expand_dims() to manually align shapes.

🧮 NumPy: Mathematical Functions

==

These functions are commonly used in **machine learning** and **deep learning**, especially for **activation**, **loss calculation**, and **model evaluation**.

🔹 1. Sigmoid Function

The **sigmoid function** is an activation function that maps any real value to the range (0, 1).

✅ **Formula:**

sigmoid(x) = 1 / (1 + exp(-x))

✅ **Example:**

```
import numpy as np


def sigmoid(x):
    return 1 / (1 + np.exp(-x))


x = np.array([-1, 0, 1, 2])
result = sigmoid(x)
```

Output:

[0.268, 0.5, 0.731, 0.881]

🔹 2. Mean Squared Error (MSE)

MSE is a **loss function** used for **regression problems**. It calculates the average of the squared differences between actual and predicted values.

✅ **Formula:**

MSE = mean((y_true - y_pred)^2)

✅ **Example:**

```
y_true = np.array([3, -0.5, 2, 7])
y_pred = np.array([2.5, 0.0, 2, 8])


mse = np.mean((y_true - y_pred) ** 2)
```

Output:

0.375

🔹 3. Binary Cross Entropy (BCE)

BCE is a **loss function** used for **binary classification** problems. It measures the difference between predicted probabilities and actual binary labels.

✅ **Formula:**

BCE = -mean(y_true * log(y_pred) + (1 - y_true) * log(1 - y_pred))

- y_true = true labels (0 or 1)
- y_pred = predicted probabilities (between 0 and 1)

✅ **Example:**

y_true = np.array([1, 0, 1, 0])

y_pred = np.array([0.9, 0.1, 0.8, 0.2])

bce = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

Output:

≈ 0.164

📌 Use np.clip(y_pred, 1e-10, 1 - 1e-10) to avoid log(0) errors in real applications.

🔚 **Summary Table:**

Function	**Used For**	**Output Range**	**Notes**
Sigmoid	Activation (binary)	(0, 1)	Smooth, differentiable
Mean Squared Error	Loss (regression)	≥ 0	Punishes larger errors more
Binary Cross Entropy	Loss (binary classification)	≥ 0	Probabilistic output, more accurate for binary outcomes

---

🧾 NumPy: Handling Missing Values (NaN)

==

🔹 **Detecting Missing Values**

Use np.isnan() to check for NaN (Not a Number) values.

✅ **Example:**

```
import numpy as np


a = np.array([1, 2, np.nan, 4])
np.isnan(a)
```

Output:

[False, False, True, False]

🔹 **Removing Missing Values**

To remove NaN values from the array:

✅ **Use Boolean Masking:**

```
cleaned = a[~np.isnan(a)]
```

Output:

[1. 2. 4.]

- np.isnan(a) → returns True where NaN is present
- ~ → logical NOT, so it selects only non-NaN values

✅ **Summary:**

- np.isnan(array) → detects NaNs
- array[~np.isnan(array)] → removes NaNs from array

📉 Plotting Graphs with NumPy & Matplotlib

==

NumPy is used to generate data
Matplotlib is used to plot the data visually

🔹 **Step 1: Import Libraries**

```
import numpy as np

import matplotlib.pyplot as plt
```

🔹 **Step 2: Generate Data with NumPy**

✅ **Example:**

```
x = np.linspace(0, 10, 100)   # 100 points from 0 to 10

y = np.sin(x)                 # Sine values of x
```

🔹 **Step 3: Plot Graph using plt.plot()**

```
plt.plot(x, y)

plt.title("Sine Wave")

plt.xlabel("X-axis")

plt.ylabel("Y = sin(x)")

plt.grid(True)

plt.show()
```

🔹 **Other Examples:**

✅ **Line Plot:**

```
x = np.arange(0, 5, 0.5)

y = x ** 2

plt.plot(x, y)
```

✅ **Scatter Plot:**

```
plt.scatter(x, y)
```

✅ **Multiple Lines:**

```
plt.plot(x, y, label="x^2")

plt.plot(x, np.sqrt(x), label="sqrt(x)")

plt.legend()
```

✅ **Summary:**

Step	Action
1	Import NumPy & Matplotlib
2	Generate x and y using NumPy
3	Use plt.plot() or plt.scatter()
4	Customize with title, labels, grid
5	Use plt.show() to display