

Python Summer Course

Course 4: Numpy

Théophile Gentilhomme

August 4, 2025



Numpy

Downloading package: micropip

Why Use NumPy?

The Problem with Native Python Loops:

- Native Python lists are **flexible but slow**
- Loops in Python are **interpreted line-by-line: not efficient for large data**
- For numerical tasks, we often want to apply the same operation to **millions of elements**

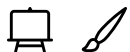


Numpy

Downloading package: ipython

NumPy to the Rescue

- NumPy provides **efficient, fixed-type arrays**
- Uses **vectorized operations** (implemented in C): much faster than Python loops!
- Reduces code size and boosts performance
- Provide a lot of useful functionalities



Example

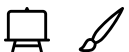
Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1  # Import numpy
2  import numpy as np
3  # To measure time
4  import time
5  import math
6
7  # Using Python list
8  py_list = list(range(1, 1000000))
9  start = time.time()
10 py_result = []
11 for x in py_list:
12     py_result.append(math.log(2 * math.sqrt(x)))
13 print("Python time:", round(time.time() - start, 5), "s")
14
15 # Using NumPy array
16 np_array = np.arange(1000000)
17 start = time.time()
18 np_result = np.log(2 * np.sqrt(np_array))
19 print("NumPy time:", round(time.time() - start, 5), "s")
```

[BACK](#) main

Numpy



! Important:

- **NumPy is 10–100x faster** for large data and avoids explicit `for` loops.
- Whenever possible, do not use loop on arrays! Use vector operations
- Numpy is used in **Pandas**
- If you know **Numpy**, you know **Pytorch** (for ML/DL, GPU accelerated library)

Creating NumPy Arrays

NumPy arrays are fixed-type, fast containers for numerical data.

You can create them from list, tuples, list of tuples, etc.

Python Code

↺ Start Over

▶ Run Code

```
1 import numpy as np
2
3 a = np.array([1, 2, 3])
4 print(a)
5 print(type(a))
6 # Type is automatically assigned based on the content of the array
7 print(a.dtype)
8 # We can also force a type
9 a = np.array([1, 2, 3], dtype=np.float)
10 print(a.dtype)
```



Numpy

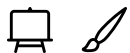
Special arrays

Commun functions to create arrays

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 print(np.zeros(5))      # full of zeros
2 print(np.ones(3))       # full of ones
3 print(np.full(4, 7))    # full of a given number
4 print(np.arange(0, 10, 2)) # similar to range()
5 print(np.linspace(0, 1, 5)) # 5 values from 0 to 1, evenly spaced
```

[↩ main](#)

Numpy

Basic Operations on Arrays

NumPy applies operations **elementwise** (vectorized) – no loops needed!

Python Code

↺ Start Over

▶ Run Code

```
1 a = np.array([1, 2, 3])
2 b = np.array([10, 20, 30])
3
4 print(a + b)
5 print(a * 2)
6 print(a ** 2)
```



Numpy

Math functions

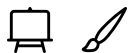
Function are applied on each element in parallel

Python Code

↺ Start Over

▶ Run Code

```
1 x = np.linspace(0, np.pi, 3)
2 print(np.sin(x))
```



← main

Numpy

Automatic broadcasting

Operation with a scalar is also automatically applied to each element.

Python Code

↺ Start Over

▶ Run Code

```
1 a = np.array([1, 2, 3])  
2 print(a + 10)
```

Broadcasting refers to an automatic expansion of an array to match the shape of a larger array. We will see that later in this course.

Multidimensional Arrays

NumPy supports arrays with any number of dimensions: 1D (vector), 2D (matrix), 3D+, etc.

Python Code

↺ Start Over

▶ Run Code

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



Numpy

Inspect the array

- Dimensions: number of axes
- Shape: number of elements for each axis
- Size: total number of elements

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 print("Dimensions:", a.ndim)    # number of dimensions or axes
2 print("Shape:", a.shape)        # shape
3 print("Size:", a.size)          # 6
```

`.ndim`, `.shape`, and `.size` help describe the structure of any array.

Reshaping Arrays

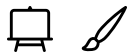
Use `reshape()` to change the shape of an array without changing its data.

Python Code

↺ Start Over

▶ Run Code

```
1 a = np.arange(12)
2 print(a.reshape(3, 4))
```



Numpy

Flattening with `ravel()`

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 v b = np.array([[1, 2, 3],  
2               [4, 5, 6]])  
3  
4 print(b.ravel()) # [1 2 3 4 5 6]
```

Can use `flatten()`, but will make a copy.



Numpy

Notes

- You can reshape to any shape that **preserves total number of elements**
- Use `-1` to let NumPy infer one dimension:

Python Code ↺ Start Over ▶ Run Code

```
1 a.reshape(2, -1)
2 print(shape)
```

Statistical Operations on Arrays

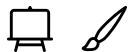
NumPy provides fast, vectorized functions to compute common statistics on arrays.

Python Code

↺ Start Over

▶ Run Code

```
1 import numpy as np
2 a = np.array([3, 7, 1, 9, 2])
3 print(np.mean(a))
```



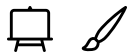
Numpy

Basic Statistics

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 print("Sum:", a.sum())
2 print("Min:", a.min())
3 print("Max:", a.max())
4 print("Mean:", a.mean())
5 print("Standard deviation:", a.std())
6 print("Median:", np.median(a))
```

[↺ main](#)

Numpy

Indexing and Slicing Arrays

NumPy arrays support fast access to elements and subarrays using indexing.

This is similar to list manipulation seen before.

Python Code

↺ Start Over

▶ Run Code

```
1 import numpy as np
2
3 a = np.array([10, 20, 30, 40, 50])
4
5 print(a[0])      # access by index
6 print(a[-1])     # can index backward (starting from the end, i.e. -1 (last), -2 (
7 print(a[-2])
8 print(a[1:4])    # Slicing: include the first but not the last!
```



← main

Numpy

Multidimensionnal Indexing

Python Code

[↺ Start Over](#)[▶ Run Code](#)

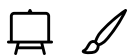
```
1 b = np.array([[1, 2, 3],  
2               [4, 5, 6]])  
3  
4 print(b[0, 1])    # row 0, column 1  
5 print(b[1])       # entire second row  
6 print(b[:, 0])    # all rows, column 0
```

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 b = np.arange(16).reshape(-1, 2, 2)  
2 print(b[3, 1, 0])  
3 print(b[1]) # equivalent to b[1, :, :]  
4 print(b[1].shape)  
5 print(b[:, 0]) # equivalent to b[:, 0, :]  
6 print(b[:, 0].shape)
```

Use `:` to select all elements along a dimension



Applying Functions Along an Axis

You can use NumPy functions (like `sum`, `mean`, `max`) with the `axis` argument to apply them row-wise or column-wise.

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 import numpy as np
2
3 a = np.array([[5, 2, 9],
4               [4, 7, 1],
5               [6, 3, 8]])
6 print(a.shape)
```

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 print("Max per row:", a.max(axis=1))    # [9 7 8]
2 print("Mean per column:", a.mean(axis=0)) # [5. 4. 6.]
```

[main](#)

Numpy

Notes

- Works with: `np.sum`, `np.mean`, `np.std`, `np.max`, `np.min`, etc.
- Default behavior “remove” the axis where to operation is applied. It can be useful to keep the axis using `keepdims=True` (keep axis, but its length becomes 1)
- **Custom function** can be applied using `np.apply_along_axis()`

Concatenating Arrays in NumPy

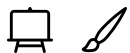
You can combine multiple arrays using `np.concatenate()` or `np.vstack()` / `np.hstack()`.

Python Code

↺ Start Over

▶ Run Code

```
1 import numpy as np
2
3 a = np.array([1, 2, 3])
4 b = np.array([4, 5, 6])
5
6 c = np.concatenate([a, b])
7 print(c)  # [1 2 3 4 5 6]
```



Concatenate 2D Arrays

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 ✓ x = np.array([[1, 2],  
2           [3, 4]])  
3  
4 y = np.array([[5, 6]])  
5  
6 # Along axis 0 (rows)  
7 print(np.concatenate([x, y], axis=0))
```

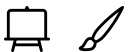
Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 ✓ z = np.array([[7],  
2           [8]])  
3  
4 # Along axis 1 (columns)  
5 print(np.concatenate([x, z], axis=1))
```

 **Important: Arrays must match in shape except along the concatenation**

Numpy

[main](#)

Broadcasting

Broadcasting lets NumPy **automatically expand smaller arrays** to match larger ones **without copying data**.

Python Code

↺ Start Over

▶ Run Code

```

1 import numpy as np
2
3 A = np.array([[1, 2, 3],
4               [4, 5, 6]])
5 print(A.shape)
6
7 b = np.array([10, 20, 30])
8 print(b.shape)
9
10 # b is broadcast across rows
11 print(A + b)
```

NumPy treats `b` as:

[[10, 20, 30],

main

Numpy



Important:

- Dimensions (axis lengths) must be **equal** or **1**
- It acts like it copies the elements to match the shape of the larger array, then apply elementwise operation
- Faster than loops
- Less code, more clarity
- Common in data normalization, scaling, or adding biases

Advanced Indexing: Boolean Masks

Boolean indexing lets you select elements **based on conditions**.

Python Code

↺ Start Over

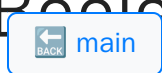
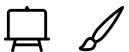
▶ Run Code

```
1 import numpy as np
2
3 a = np.array([10, 20, 30, 40, 50])
4 mask = a > 25
5
6 print(mask)          # [False False  True  True  True]
7 print(a[mask])       # [30 40 50]
```

⚠ Important:

- Boolean masks must be the **same shape** as the array

Numpy



Index Arrays and Fancy Indexing

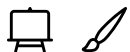
You can index using lists or arrays of positions, even in multiple dimensions.

Python Code

↺ Start Over

▶ Run Code

```
1 a = np.array([100, 200, 300, 400, 500])
2 # Index with lists or arrays of int
3 print(a[[0, 3, 4]])
4 index = np.array((2, 4))
5 print(a[index])
```



2D Fancy Indexing

Python Code

[↺ Start Over](#)[▶ Run Code](#)

```
1 v b = np.array([[10, 20],  
2               [30, 40],  
3               [50, 60]])  
4  
5 rows = [0, 2]  
6 cols = [1, 0]  
7 print(b[rows, cols]) # [20 50]
```

Advanced indexing returns **a copy**, not a view.

Combining Indexing with Slicing (`:`)

You can mix slicing (`:`) with Boolean masks or index arrays to target **specific rows or columns**.

Python Code

↺ Start Over

▶ Run Code

```
1 import numpy as np
2
3 a = np.array([[10, 20, 30],
4               [40, 50, 60],
5               [70, 80, 90]])
6
7 rows = [0, 2]
8 print(a[rows, :])
```

Python Code

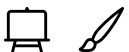
↺ Start Over

▶ Run Code

```
1 cols = [1, 2]
2 print(a[:, cols])
```

BACK main

Numpy



Your turn!

You're working with sensor data collected in a lab: each row represents a different sensor, each column a timepoint.

1. Create the dataset of shape (10, 6) using `np.random.randint()`. Random value must be between 1 and 110.

2. Compute:

- Mean per sensor (using `axis=...`)
- Max value per sensor
- (Optional) Index of the time when max occurs (using `.argmax()`
[<https://numpy.org/doc/stable/reference/generated/numpy.argmax>])

3. Use boolean indexing to select sensors whose **average** reading is above 60
4. Subtract mean from each row (sensor), using broadcasting
5. Simulate a new column of sensor (i.e. use `randint` with correct shape) readings and concatenate it to the existing data.

Solution

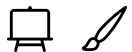
[Show Solution](#)

Numpy

More references

[Python course for data analysis](#)

[Numpy.](#)



Numpy