Python Summer Course

Course 4: Numpy

Théophile Gentilhomme

July 11, 2025

Downloading Pyodide

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 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

```python
# Import numpy
import numpy as np
# To measure time
import time
import math

# Using Python list
py_list = list(range(1, 1000000))
start = time.time()
py_result = []
for x in py_list:
    py_result.append(math.log(2 * math.sqrt(x)))
print("Python time:", round(time.time() - start, 5), "s")

# Using NumPy array
np_array = np.arange(1000000)
start = time.time()
np_result = np.log(2 * np.sqrt(np_array))
print("NumPy time:", round(time.time() - start, 5), "s")
```

⚠️ 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 accelarated 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

```python
import numpy as np

a = np.array([1, 2, 3])
print(a)
print(type(a))
# Type is automatically assigned based on the content of the array
print(a.dtype)
# We can also force a type
a = np.array([1, 2, 3], dtype=np.float)
print(a.dtype)
```

Special arrays

Commun functions to create arrays

Python Code Start Over Run Code

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

Basic Operations on Arrays

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

Python Code | Start Over | Run Code

```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])

print(a + b)
print(a * 2)
print(a ** 2)
```

Math functions

Function are applied on each element in parallel

Python Code Start Over Run Code

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

Automatic broadcasting

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

Python Code Start Over Run Code

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

```python
import numpy as np

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

print(a)
```

Inspect the array

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

Python Code Start Over Run Code

```python
print("Dimensions:", a.ndim)    # number of dimensions or axes
print("Shape:", a.shape)        # shape
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

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

Flattening with `ravel()`

Python Code | Start Over | Run Code

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

print(b.ravel())  # [1 2 3 4 5 6]
```

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

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

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

Statistical Operations on Arrays

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

Python Code | Start Over | Run Code

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

Basic Statistics

Python Code Start Over Run Code

```python
print("Sum:", a.sum())
print("Min:", a.min())
print("Max:", a.max())
print("Mean:", a.mean())
print("Standard deviation:", a.std())
print("Median:", np.median(a))
```

Indexing and Slicing Arrays

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

This is simular to list manipulation seen before.

Python Code   Start Over   Run Code

```python
import numpy as np

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

print(a[0])      # access by index
print(a[-1])     # can index backward (starting from the end, i.e. -1 (last), -2 (
print(a[-2])
print(a[1:4])    # Slicing: include the first but not the last!
```

Multidimensionnal Indexing

Python Code Start Over Run Code

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

print(b[0, 1])    # row 0, column 1
print(b[1])       # entire second row
print(b[:, 0])    # all rows, column 0
```

Python Code Start Over Run Code

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

```python
import numpy as np

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

Python Code Start Over Run Code

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

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

```python
import numpy as np

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

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

Concatenate 2D Arrays

Python Code Start Over Run Code

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

y = np.array([[5, 6]])

# Along axis 0 (rows)
print(np.concatenate([x, y], axis=0))
```

Python Code Start Over Run Code

```
z = np.array([[7],
              [8]])

# Along axis 1 (columns)
print(np.concatenate([x, z], axis=1))
```

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

Broadcasting

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

Python Code Start Over Run Code

```python
import numpy as np

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

b = np.array([10, 20, 30])
print(b.shape)

# b is broadcast across rows
print(A + b)
```

NumPy treats `b` as:

```
[[10, 20, 30],
```

⚠️ Important:

- Dimensions (axis lenghts) 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

```python
import numpy as np

a = np.array([10, 20, 30, 40, 50])
mask = a > 25

print(mask)       # [False False  True  True  True]
print(a[mask])    # [30 40 50]
```

⚠️ Important:

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

Index Arrays and Fancy Indexing

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

Python Code   Start Over   Run Code

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

2D Fancy Indexing

Python Code Start Over Run Code

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

rows = [0, 2]
cols = [1, 0]
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

```
import numpy as np

a = np.array([[10, 20, 30],
              [40, 50, 60],
              [70, 80, 90]])

rows = [0, 2]
print(a[rows, :])
```

Python Code Start Over Run Code

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

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

Python Code · Start Over · Run Code

```python

import numpy as np

# Simulate 10 sensors over 6 timepoints (values from 0 to 100)

print("Step 1")
data = np.random.randint(0, 101, size=(10, 6))
print(data)
print("-----------------")

# Statistics

print("Step 2")
mean_per_sensor = data.mean(axis=1)
print(mean_per_sensor)
print(data.max(axis=1))
print(data.argmax(axis=1))
print("-----------------")

```