NUMPY

WHY NUMPY?

```
import numpy as np
```

Python is slow for numerical computing

•Python itself is an interpreted language — it processes one instruction at a time.

•Doing heavy number crunching (e.g., multiplying millions of numbers) in plain Python loops is **very slow**.

That's where **NumPy** comes in.

NumPy is mostly written in C

•The **core of NumPy** (**ndarray objects and mathematical operations**) is implemented in **C** (and some Fortran).

•C is a **compiled language**, meaning the code runs directly on your machine's processor → **much faster than Python loops**.

CREATING ARRAYS

```
a = np.array([1, 2, 3])          # 1D
M = np.array([[1,2,3],[4,5,6]])  # 2D
zeros = np.zeros((3,3))
ones  = np.ones(4)
r = np.arange(0,10,2)            # like range()
lin = np.linspace(0,1,5)         # evenly spaced
I = np.eye(3)                    # identity
```

Exercise:
Create a 4x2 array of ones and a vector from 10 down to 1.

ARRAY ATTRIBUTES

```
a.shape    # tuple of dims
a.ndim     # number of dimensions
a.size     # total elements
a.dtype    # data type
```

INDEXING AND SLICING

Basic Indexing

```
v = np.array([10,20,30,40])
v[0]     # first
v[-1]    # last
v[1:3]   # slice (view)
```

2D Indexing

```
M[1,2]   # row 1 col 2
M[:,1]   # all rows, col 1
M[0]     # first row
```

Copying an array

```
s = v[1:3]
s[0] = 999     # changes original v
copy = v[1:3].copy()  # make a copy instead
```

Exercise: slice a 3×4 matrix to get the middle two columns and modify one value.

VECTORIZED OPERATIONS

```
a = np.array([1,2,3])
b = np.array([4,5,6])
a + b          # [5,7,9]
a * 2          # [2,4,6]
a * b          # elementwise [4,10,18]
a @ b          # dot product (or np.dot(a,b))
```

- Much faster than Python loops.
  Exercise: compute column-wise mean of a 2D array using vectorized ops.

BROADCASTING

Compatibility rules:

- dimensions either equal or one is 1;
- NumPy aligns trailing dims.

```
A = np.ones((2,3))
v = np.array([1,2,3])    # shape (3,)
A + v   # v broadcasts across rows => each row + v
```

```
import numpy as np

arr = np.array([1, 2, 3, 4])
print(arr + 10)    # [11 12 13 14]
```

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

print(a + b)
```

Exercise: add a (3,1) column vector to a (3,4) matrix

AGGREGATION AND AXIS

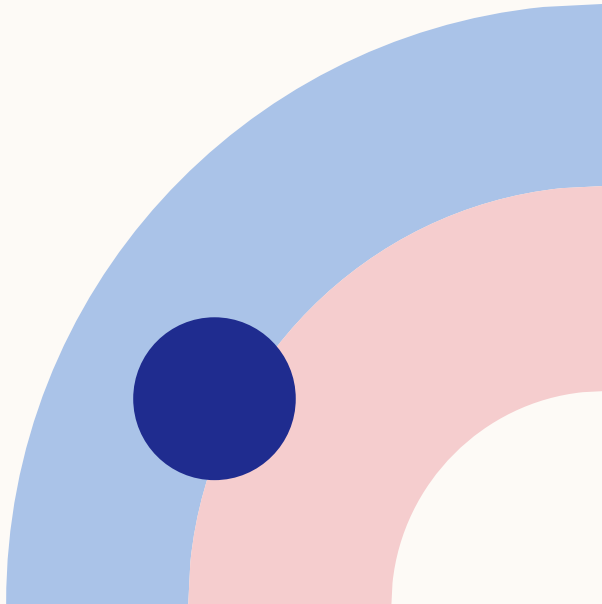

```
data = np.array([[1, 2, 3],
                 [4, 5, 6]])

print(data.sum())           # 21  (all elements)
print(data.sum(axis=0))     # [5 7 9]  (column-wise)
print(data.sum(axis=1))     # [6 15]   (row-wise)

print(data.mean())          # 3.5
print(data.min(axis=1))     # [1 4]
print(data.max(axis=0))     # [4 5 6]
```

Exercise: compute average per row and per column for a sample matrix.

LINEAR ALGEBRA

```
A = np.array([[1,2],[3,4]])
np.linalg.inv(A)
np.linalg.det(A)
np.linalg.eig(A)
```

Exercise: compute eigenvalues of a simple matrix.



PERFORMANCE COMPARISON PYTHON LIST VS. NUMPY ARRAY

Python talks to C via a "wrapper"

- NumPy provides a Python API (functions like np.array, np.dot, etc.).
- When you call a NumPy function in Python, under the hood it calls optimized C functions.

- This is possible because:
  - Python allows C extensions (modules written in C can be imported in Python).
  - NumPy is such a C extension.

```
import time
n = 1_000_000
L = list(range(n))
start = time.time()
[s*2 for s in L]
print("list time", time.time()-start)

import numpy as np
a = np.arange(n)
start = time.time()
a*2
print("numpy time", time.time()-start)
```

PERFORMANCE COMPARISON PYTHON LIST VS. NUMPY ARRAY

Vectorization vs SIMD

•NumPy's core vectorization = **looping in C over raw memory buffers** (fast).

•Some NumPy builds (e.g., with Intel MKL, OpenBLAS, or SIMD instructions like AVX) **use CPU vector instructions**.

- This means the CPU itself processes multiple elements in one instruction.
- Example: adding 4 doubles in one CPU cycle instead of one at a time.

•That's why performance can differ depending on your NumPy build.