Understanding Dimension, Shape, Axis, Broadcasting and Vectorization

1. Dimension (ndim)

- **Definition:** Number of axes (or levels of nesting) in a NumPy array.
- **Examples:**
  - Scalar → 0-D (No dimension)
  - Vector → 1-D
  - Matrix → 2-D
  - Tensor → 3-D or higher

Example Array	ndim (Dimension)
`np.array(5)`	0-D
`np.array([1,2,3])`	1-D
`np.array([[1,2,3]])`	2-D
`np.zeros((2,3,4))`	3-D

---

2. Shape

- **Definition:** A tuple representing the **size of the array along each dimension**.
- **Examples:**
  - A 1-D array of 5 elements → `(5,)`
  - A 2-D array with 3 rows and 4 columns → `(3, 4)`
  - A 3-D array with shape `(2,3,4)` → 2 blocks, 3 rows, 4 columns each

---

3. Axis

- **Definition:** A particular **dimension along which operations are performed**.

Rule of Axes in NumPy

Axis number increases with depth of dimensions:

- Axis 0 → First dimension (outermost)
- Axis 1 → Second dimension
- Axis 2 → Third dimension ...and so on.

Operations collapse the given axis, meaning elements along that axis are combined, and the axis disappears in the result.

- **Example:**
  - Summing along `axis=0` → Collapse first dimension
  - Summing along `axis=1` → Collapse second dimension

1. 2D Array Case

In [4]:
```
import numpy as np
arr = np.array([[1, 2, 3],
                [4, 5, 6]])
```

In [5]:
```
np.sum(arr, axis=0) # Sum along axis 0 (Column-wise)
```

Out[5]:
```
array([5, 7, 9])
```

In [6]:
```
np.sum(arr, axis=1) # Sum along axis 1 (Row-wise)
```

Out[6]:
```
array([ 6, 15])
```

2. 3D Array Case

In [8]:
```
import numpy as np

arr3d = np.array([
    [[ 1,  2,  3],
     [ 4,  5,  6]],

    [[ 7,  8,  9],
     [10, 11, 12]]
])

arr3d
```

Out[8]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In [9]:
```
print(arr3d.shape)
```

```
(2, 2, 3)
```

Shape (2, 2, 3) means:

- 2 blocks
- 2 rows per block
- 3 columns per row

In [11]:
```python
np.sum(arr3d, axis=0)
```

Out[11]:
```
array([[ 8, 10, 12],
       [14, 16, 18]])
```

In [14]:
```python
np.sum(arr3d, axis=0).shape
```

Out[14]:
```
(2, 3)
```

In [12]:
```python
np.sum(arr3d, axis=1)
```

Out[12]:
```
array([[ 5,  7,  9],
       [17, 19, 21]])
```

In [13]:
```python
np.sum(arr3d, axis=1).shape
```

Out[13]:
```
(2, 3)
```

In [15]:
```python
np.sum(arr3d, axis=2)
```

Out[15]:
```
array([[ 6, 15],
       [24, 33]])
```

In [16]:
```python
np.sum(arr3d, axis=2).shape
```

Out[16]:
```
(2, 2)
```

In Short:

- **Dimension (ndim)** → How many axes?
- **Shape** → Size along each axis.
- **Axis** → Direction along which operations are applied.

---

4. Broadcasting

Broadcasting Rules

1. Compare shapes from the last dimension to the first.
   - If one shape has fewer dimensions, leading 1s are added automatically to the smaller shape.
2. Dimensions are compatible if:
   - They are equal, or
   - One of them is 1
3. Resulting shape is the maximum in each dimension after broadcasting.

In [19]:
```
#Simple Example (1D + Scalar)

arr = np.array([1, 2, 3])
print(arr + 5)  # Broadcasts 5 to [5, 5, 5]
```

```
[6 7 8]
```

In [20]:
```
# 1D and 2D broadcasting

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

b = np.array([10, 20, 30])   # Shape (3,)

C = A + b
print(C)
```

```
[[11 22 33]
 [14 25 36]]
```

In [21]:
```
np.array([1,2,3]).shape
```

Out[21]:
```
(3,)
```

In [22]:
```
np.array([[1],[2]]).shape
```

Out[22]:
```
(2, 1)
```

In [23]:
```
np.array([1,2,3]) + np.array([[1],[2]])
```

Out[23]:
```
array([[2, 3, 4],
       [3, 4, 5]])
```

Broadcasting Fails When Shapes are Incompatible

In [26]:
```
A = np.array([[1, 2, 3],
              [4, 5, 6]])   # Shape (2,3)

B = np.array([[10, 20],
              [30, 40]])    # Shape (2,2)

# A+B
```

Broadcasting Internals Explained with Examples

1. Broadcasting in NumPy is implemented in C under the hood for speed.
2. Align shapes: `(2,3)` and `(1,3)`
3. Strides for B (row stride = 0) → Reuse the same row twice
4. Compute element-wise sum in a new `(2,3)` array - using Vectorization

Advantages of Broadcasting

1. Memory efficient (no actual data replication).

2. Faster computations using vectorization instead of loops.

---

5. Vectorization

- Vectorization means performing operations on entire arrays at once instead of iterating element by element in Python.
- NumPy operations are implemented in C, which avoids Python-level loops and interpreter overhead.

In [31]:
```
A = np.array([1, 2, 3])
B = np.array([2, 3, 4])
print(A,B)
```

```
[1 2 3] [2 3 4]
```

In [32]:
```
A+B
```

Out[32]:
```
array([3, 5, 7])
```

In [41]:
```
[int(x+y) for x, y in zip(A, B)]
```

Out[41]:
```
[3, 5, 7]
```

When you do A + B:

- NumPy does not loop in Python.
- It calls optimized C functions that perform element-wise addition directly on the memory buffers.

Why Vectorization is Faster?:

- **C-level implementation**: NumPy's loops are in compiled C, much faster than Python loops.
- **Fewer Python instructions**: No per-element Python function call overhead.
- **SIMD optimizations**: Uses CPU vectorized instructions for multiple elements at once using wide vector registers (SSE, AVX registers).
- **Memory efficiency**: Works with contiguous arrays and broadcasting without creating many intermediate Python objects.

Performance Comparison - Vectorization Vs Loops

In [ ]:
```
# Example 1: Squaring 10 million numbers
```

In [43]:
```
import numpy as np
import time
```

```
N = 10000000
arr = np.arange(N)

arr.shape
```

Out[43]: (10000000,)

In [44]:
```
# Python Loop Implementation
start = time.time()
result_loop = [x**2 for x in arr]  # Python Loop
print("Python loop time:", time.time() - start)
```

```
Python loop time: 1.0810484886169434
```

In [45]:
```
# NumPy Vectorized Operation (Fast)
start = time.time()
result_vec = arr**2  # Vectorized operation
print("Vectorized time:", time.time() - start)
```

```
Vectorized time: 0.018139839172363281
```

In [46]:
```
1.0810484886169434// 0.018139839172363281
```

Out[46]: 59.0

⚡ Vectorized code is **50x+ faster** because it runs in compiled C loops internally.