3. Array Properties

- `shape`
- `ndim`
- `size`
- `dtype`

1. Topic Overview

Array Properties describe the basic facts about a NumPy array.
They tell you **what the array looks like in memory**.

This topic exists because **ML models break silently** if array shape or type is wrong.

Real-world analogy (one only):
Think of an array like a box shipment.

- `shape` = box layout
- `ndim` = number of layers
- `size` = total items
- `dtype` = item type

If any is wrong, delivery fails.

2. Core Theory (Deep but Clear)

We are talking about **NumPy arrays**, not Python lists.

NumPy stores data as:

- **Continuous memory**
- **Fixed data type**
- **Strict shape**

Array properties let you **inspect this internal structure**.

2.1 `shape`

- `shape` tells **rows and columns**
- It is a **tuple**
- It defines how NumPy maps memory to dimensions

Internally:

- NumPy stores data in 1D memory
- `shape` tells how to *view* that memory

2.2 `ndim`

- `ndim` means **number of dimensions**
- It is an integer

Internally:

- Each dimension adds a level of indexing
- ML models expect specific `ndim`

2.3 `size`

- `size` means **total number of elements**
- It is `product(shape)`

Internally:

- NumPy counts raw memory slots
- Independent of how data is grouped

2.4 `dtype`

- `dtype` means **data type**
- Every element has the same type

Internally:

- Determines **memory per element**
- Controls speed and precision

3. Syntax & Examples

We use one array and inspect all properties.

```
import numpy as np

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

3.1 `shape`

```
print(arr.shape)
```

Output

```
(2, 3)
```

Explanation:

- 2 rows
- 3 columns
- Tuple format

Another example:

```
x = np.array([10, 20, 30, 40])
print(x.shape)
```

Output

```
(4,)
```

Explanation:

- 1D array
- Single dimension
- Trailing comma means tuple

3.2 `ndim`

```
print(arr.ndim)
```

Output

```
2
```

Explanation:

- 2 axes
- Row axis + column axis

```
print(x.ndim)
```

Output

```
1
```

Explanation:

- Single axis
- Flat vector

3.3 `size`

```
print(arr.size)
```

Output

```
6
```

Explanation:

- 2 × 3 = 6 elements

```
print(x.size)
```

Output

```
4
```

Explanation:

- Total count
- Independent of shape style

3.4 `dtype`

```
print(arr.dtype)
```

Output

```
int64
```

Explanation:

- 64-bit integer
- Fixed memory per element

```python
y = np.array([1.2, 3.5, 7.8])
print(y.dtype)
```

Output

```
float64
```

Explanation:

- Decimal numbers
- More memory than int

4. Why This Matters in Data Science

Data Cleaning

- Detect unexpected dimensions
- Catch mixed data types early

Feature Engineering

- Ensure features are `(n_samples, n_features)`
- Prevent accidental flattening

Model Input Preparation

- ML models expect exact `shape`
- Wrong `ndim` breaks `.fit()` or `.predict()`

ML / DL Pipelines

- Broadcasting depends on `shape`
- GPU memory depends on `dtype`

What breaks if you don't know this

- Silent training bugs
- Wrong predictions

- Runtime errors during deployment

5. Common Mistakes (VERY IMPORTANT)

1. **Confusing `(n,)` with `(n,1)`**
   Happens due to weak shape understanding
   Fix: Always print `shape`
2. **Passing 1D array to model expecting 2D**
   Happens in sklearn
   Fix: Check `ndim`
3. **Ignoring dtype during division**
   Integer division truncates
   Fix: Convert to float
4. **Assuming size equals length**
   Wrong for multidimensional arrays
   Fix: Use `shape` for structure
5. **Using Python list assumptions**
   Lists allow mixed types
   NumPy does not
   Fix: Always inspect `dtype`

6. Performance & Best Practices

Fast when

- dtype is numeric
- memory is contiguous
- shape matches vectorized ops

Slow when

- dtype is `object`
- frequent reshaping without understanding

Warnings

- Changing dtype increases memory

- Wrong shape causes broadcasting bugs

Best practice:

- Always log `shape`, `ndim`, `dtype`
- Before training
- Before saving models

7. 20 Practice Problems (MANDATORY)

Easy (5)

1. Create a 1D array of 10 integers. Print all properties.
2. Convert a list of floats to array. Check dtype.
3. Create a 3×4 array. Print size.
4. Create a single value array. Check ndim.
5. Compare shape of `[1,2,3]` and `[[1,2,3]]`.

Medium (7)

6. Load CSV numeric data into array. Validate shape.
7. Check if feature matrix is 2D before training.
8. Detect wrong dtype after division.
9. Reshape array and verify size unchanged.
10. Identify samples vs features using shape.
11. Convert int array to float for ML.
12. Debug broadcasting error using ndim.

Hard (5)

13. Validate input for linear regression model.
14. Detect memory waste due to wrong dtype.
15. Ensure batch input shape for neural network.
16. Debug silent model failure using shape logs.
17. Compare performance using int32 vs float64.

Industry-Level (3)

18. Build input validation for ML pipeline.
19. Detect corrupted feature array in production.
20. Optimize memory for large dataset using dtype.

8. Mini Checklist

- `shape` defines structure
- `ndim` defines model compatibility
- `size` counts raw elements
- `dtype` controls memory and speed
- Always print properties before ML steps

Additional Mandatory NumPy Array Properties for Data Science

Covered Properties

- `itemsize`
- `nbytes`
- `T` (transpose)
- `strides`
- `flags`

All belong to **NumPy arrays**.

1. Topic Overview

These properties describe **memory layout and data movement**, not just shape.

They exist because:

- ML systems deal with **large data**
- Memory misuse causes **slow training and crashes**

Real-world analogy (one only):
Think of a warehouse.

- `itemsize` = size of one box
- `nbytes` = total storage used
- `T` = shelf rotation
- `strides` = walking steps between boxes
- `flags` = warehouse rules

2. Core Theory (Deep but Clear)

NumPy arrays are:

- Fixed dtype
- Continuous memory (usually)
- Viewed through metadata

These properties expose **how NumPy sees memory internally**.

2.1 `itemsize`

- Size of **one element**
- Measured in **bytes**

Internally:

- Depends on `dtype`
- Affects cache and speed

2.2 `nbytes`

- Total memory used by array

- `size × itemsize`

Internally:

- Direct RAM consumption
- Critical for large datasets

2.3 `T` (Transpose)

- Swaps axes
- Returns a **view**, not copy (usually)

Internally:

- Changes how memory is interpreted
- Does not move data physically

2.4 `strides`

- Steps (in bytes) to move between elements
- Tuple per dimension

Internally:

- Controls how NumPy jumps in memory
- Key for slicing and views

2.5 `flags`

- Metadata about memory behavior
- Read-only, contiguous, writable

Internally:

- Used to protect memory
- Used by NumPy optimizers

3. Syntax & Examples

We reuse one array.

```
import numpy as np

arr = np.array([[1, 2, 3],
                [4, 5, 6]], dtype=np.int32)
```

3.1 `itemsize`

```
print(arr.itemsize)
```

Output

```
4
```

Explanation:

- int32 = 4 bytes
- Each element uses 4 bytes

Another example:

```
x = np.array([1.5, 2.5])
print(x.itemsize)
```

Output

```
8
```

Explanation:

- float64
- Higher precision, more memory

3.2 `nbytes`

```
print(arr.nbytes)
```

Output

```
24
```

Explanation:

- 6 elements
- 6 × 4 bytes = 24

```
print(x.nbytes)
```

Output

```
16
```

Explanation:

- 2 × 8 bytes

3.3 `T` (Transpose)

```
print(arr.T)
```

Output

```
[[1 4]
 [2 5]
 [3 6]]
```

Explanation:

- Shape changed from (2,3) to (3,2)
- Data not copied

```python
print(arr.shape, arr.T.shape)
```

Output

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

Explanation:

- Axes swapped

3.4 `strides`

```python
print(arr.strides)
```

Output

```
(12, 4)
```

Explanation:

- Move 12 bytes to next row
- Move 4 bytes to next column

```
print(arr.T.strides)
```

Output

```
(4, 12)
```

Explanation:

- Memory steps reversed
- Shows view behavior

3.5 `flags`

```
print(arr.flags)
```

Output (simplified)

```
C_CONTIGUOUS : True
WRITEABLE    : True
OWNDATA      : True
```

Explanation:

- Stored row-wise
- Can be modified
- Owns its memory

4. Why This Matters in Data Science

Data Cleaning

- Detect unnecessary memory copies
- Reduce memory usage

Feature Engineering

- Transpose features safely
- Avoid copying huge arrays

Model Input Preparation

- DL frameworks check contiguity
- Wrong strides slow GPU transfer

ML / DL Pipelines

- Batch creation depends on strides
- Memory flags affect multiprocessing

What breaks if you don’t know this

- Out-of-memory errors
- Very slow training
- Hidden performance drops

5. Common Mistakes (VERY IMPORTANT)

1. **Ignoring nbytes for large data**
   Causes RAM crash
   Fix: Always check memory usage
2. **Assuming transpose copies data**
   Leads to unsafe modifications
   Fix: Check `flags`
3. **Using float64 everywhere**
   Wastes memory
   Fix: Use float32 when possible
4. **Not understanding strides**
   Leads to slow slicing
   Fix: Learn view vs copy
5. **Modifying non-writeable arrays**
   Causes runtime errors
   Fix: Check `flags.writeable`

6. Performance & Best Practices

Fast when

- C_CONTIGUOUS is True
- dtype is numeric
- strides are simple

Slow when

- Non-contiguous views
- Large transpose chains

Warnings

- Transpose before training carefully
- Copy only when required

Best practice:

- Log `nbytes` for big arrays
- Use float32 for ML
- Avoid unnecessary copies

7. 20 Practice Problems (MANDATORY)

Easy (5)

1. Check itemsize for int32 vs float64.
2. Calculate nbytes manually and verify.
3. Transpose a 2×3 array and print shape.
4. Inspect strides of 1D array.
5. Check flags of a sliced array.

Medium (7)

6. Reduce memory by changing dtype.

7. Compare nbytes before and after type cast.
8. Detect non-contiguous array.
9. Check if transpose shares memory.
10. Debug slow slicing using strides.
11. Validate array before GPU transfer.
12. Identify write-protected arrays.

Hard (5)

13. Optimize memory for million-row dataset.
14. Ensure contiguous input for DL model.
15. Debug model slowdown due to views.
16. Track memory spike in pipeline.
17. Compare copy vs view cost.

Industry-Level (3)

18. Prevent OOM crash in production ML job.
19. Optimize batch pipeline memory.
20. Build array validator for performance.

8. Mini Checklist

- `itemsize` = bytes per element
- `nbytes` = total memory
- `T` usually returns view
- `strides` control memory jumps
- `flags` reveal safety and layout