2. Creating NumPy Arrays

This topic is **foundational**. If you mess this up, everything later breaks silently. Shapes go wrong, models fail, performance tanks. So we go slow and precise.

I’ll cover **each function one by one**, but first you need the big picture.

Why array creation matters (don’t skip this)

In data science and ML, arrays are used for:

- Raw data storage
- Feature matrices (X)
- Labels (y)
- Model weights
- Intermediate computations

1. Raw Data Storage

What it means

This is data **exactly as you receive it**, before cleaning or processing.

- From CSV files
- From databases
- From APIs
- From sensors or logs

No ML thinking yet. Just storage in array form.

Example

Imagine you read height and weight data from a file.

```python
import numpy as np

raw_data = np.array([
    [180, 75],
    [165, 60],
    [170, 68]
])
```

- Shape: `(3, 2)`
- Each row = one person
- Each column = raw measurement

Common mistake

Thinking raw data is already “model-ready”. It is not. Raw data is usually dirty, unscaled, and inconsistent.

2. Feature Matrix (X)

What it means

Features are **inputs to the model**.

Feature matrix `X` contains:

- Rows → samples
- Columns → features

This is the most important NumPy concept for ML.

Example

From the same data, height and weight are features.

```
X = np.array([
    [180, 75],
    [165, 60],
    [170, 68]
])
```

- Shape: `(n_samples, n_features)`
- Here: `(3, 2)`

Interview rule (remember this)

> X is **always 2D** in classical ML.

Even if you have one feature, shape should be `(n, 1)` , not `(n,)` .

Common mistake

Using shape `(n,)` instead of `(n, 1)` and breaking matrix math.

3. Labels (y)

What it means

Labels are **outputs or answers** the model must learn to predict.

- Classification → class labels
- Regression → numeric values

Example

Suppose we predict if a person is “fit” (1) or “not fit” (0).

```
y = np.array([1, 0, 1])
```

- Shape: `(3,)`
- One label per row in X

Relationship rule (very important)

```
Number of rows in X == length of y
```

If this breaks, your model is invalid.

Common mistake

- Misaligned rows
- Extra or missing labels
- Treating y as 2D when not required

4. Model Weights

What it means

Weights are **parameters learned by the model**.

They control:

- How much importance each feature has
- How predictions are computed

Weights are **not data**. They are learned from data.

Example (simple linear model)

```
weights = np.array([0.4, 0.6])
bias = 0.2
```

Prediction formula:

```
prediction = X · weights + bias
```

Here:

- Shape of weights: `(n_features,)`

- Must match number of columns in X

ML reality

Bad weight shape = runtime error or silent wrong output.

Common mistake

Initializing weights with wrong shape and blaming the model.

5. Intermediate Computations

What it means

Temporary arrays created **during processing**, not stored permanently.

Used for:

- Scaling
- Normalization
- Loss calculation
- Gradients
- Predictions

Example

Normalize features:

```
mean = X.mean(axis=0)
std = X.std(axis=0)

X_scaled = (X - mean) / std
```

- `mean` , `std` , `X_scaled` are intermediate arrays
- They exist only to compute something else

Another example

Predictions:

```
y_pred = X @ weights + bias
```

`y_pred` is an intermediate computation.

Common mistake

Saving intermediate arrays as final data and confusing pipelines.

One Clear Mental Picture (important)

```
Raw Data  →  Feature Matrix (X)
                  ↓
               Model Weights
                  ↓
        Intermediate Computations
                  ↓
                Prediction
                  ↓
                Labels (y)
```

Interview-level clarity (don’t miss this)

- Raw data → storage
- X → input to model (2D)
- y → output/target (usually 1D)
- Weights → learned parameters
- Intermediate → temporary math results

If you mix **data** and **parameters** in your explanation, interviewer will stop you immediately.

1. Classification

What it means

Classification = predicting a category or class.

The output is **discrete**, not continuous.
The model chooses from a **fixed set of labels**.

Simple examples

- Email → spam or not spam
- Loan → approved or rejected
- Disease → yes or no

Outputs look like:

```
0, 1
"yes", "no"
"A", "B", "C"
```

NumPy-style example

```
# Features: height, weight
X = np.array([
    [180, 75],
    [165, 60],
    [170, 68]
])

# Labels: fit (1) or not fit (0)
y = np.array([1, 0, 1])
```

Here:

- X → input features
- y → class labels
- y values come from a **small fixed set**

Real-world classification tasks

- Fraud detection
- Face recognition
- Sentiment analysis
- Medical diagnosis

Common beginner mistake

Thinking probability output means regression.

Wrong.
Even if the model outputs `0.92` , it is still **classification** if that number represents confidence for a class.

2. Regression

What it means

Regression = predicting a continuous numeric value.

The output is a **number on a scale**, not a category.

Simple examples

- House price → ₹45,00,000
- Temperature → 32.5°C
- Salary → ₹8.7 LPA

Outputs look like:

```
12.3
4500000
-7.8
```

NumPy-style example

```
# Features: size (sqft), number of rooms
X = np.array([
    [1000, 2],
    [1500, 3],
    [2000, 4]
])

# Labels: house prices
y = np.array([30, 45, 65])  # in lakhs
```

Here:

- y is **continuous**
- There is no fixed list of possible outputs

Real-world regression tasks

- Stock price prediction
- Demand forecasting
- Risk scoring
- Energy consumption prediction

Common beginner mistake

Converting numeric values into buckets and calling it regression.

If you do:

```
price < 50 → cheap
price >= 50 → expensive
```

You just turned regression into classification.

The cleanest comparison (memorize this)

Aspect	Classification	Regression
Output type	Category	Number
Output nature	Discrete	Continuous
Example output	0 / 1	42.7
Typical loss	Cross-entropy	MSE
Question type	“Which one?”	“How much?”

Interview traps (important)

Trap 1

“Binary classification is just regression with two values.”

Wrong.
Binary classification predicts **class probability**, not a raw numeric value.

Trap 2

“If output is a number, it’s regression.”

Wrong.
If the number represents a **class ID**, it is still classification.

Trap 3

“Logistic regression is regression.”

Wrong by name.
Logistic regression is a **classification algorithm**.

If you say otherwise, interview is over.

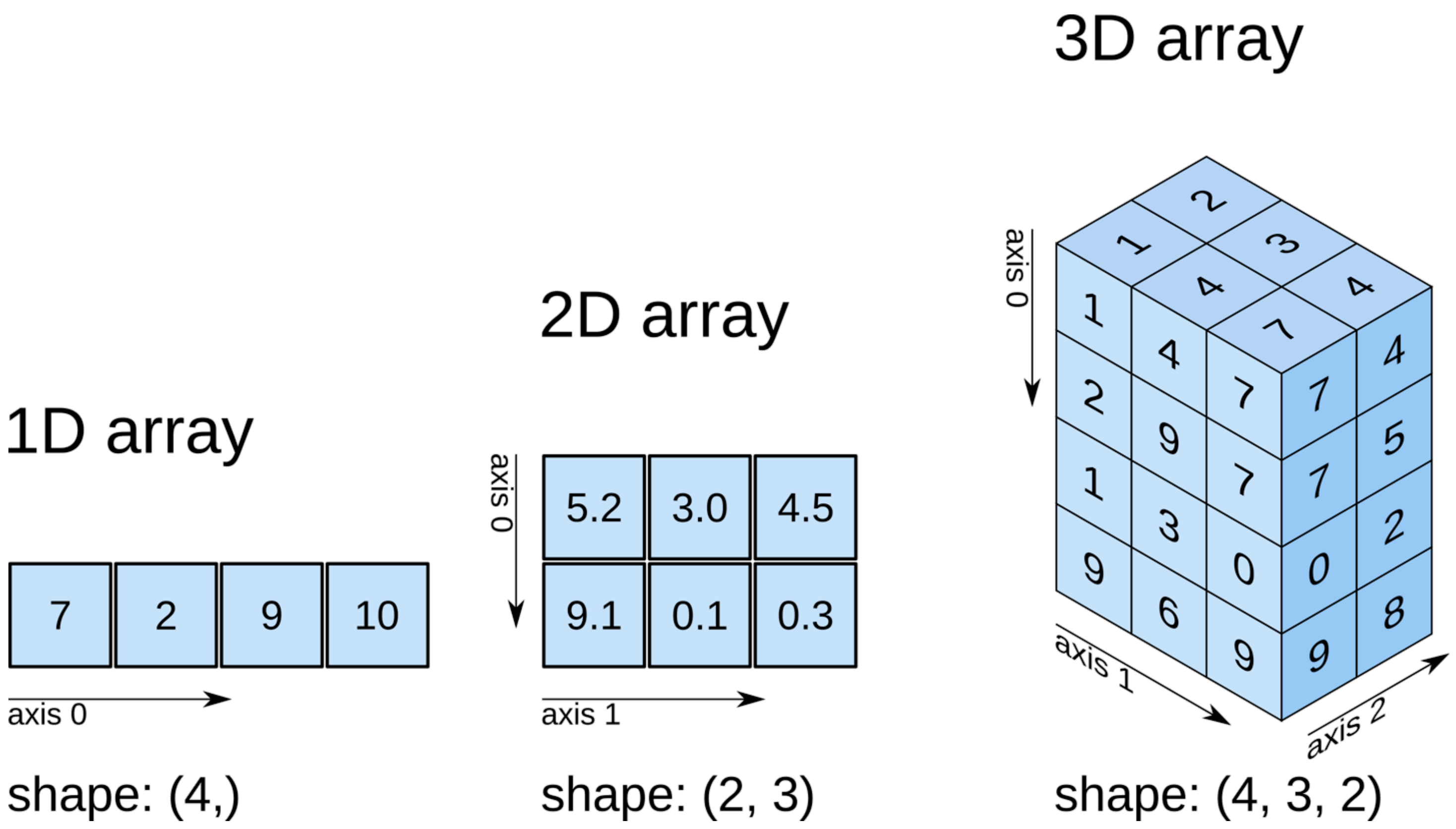
One-line intuition (this sticks)

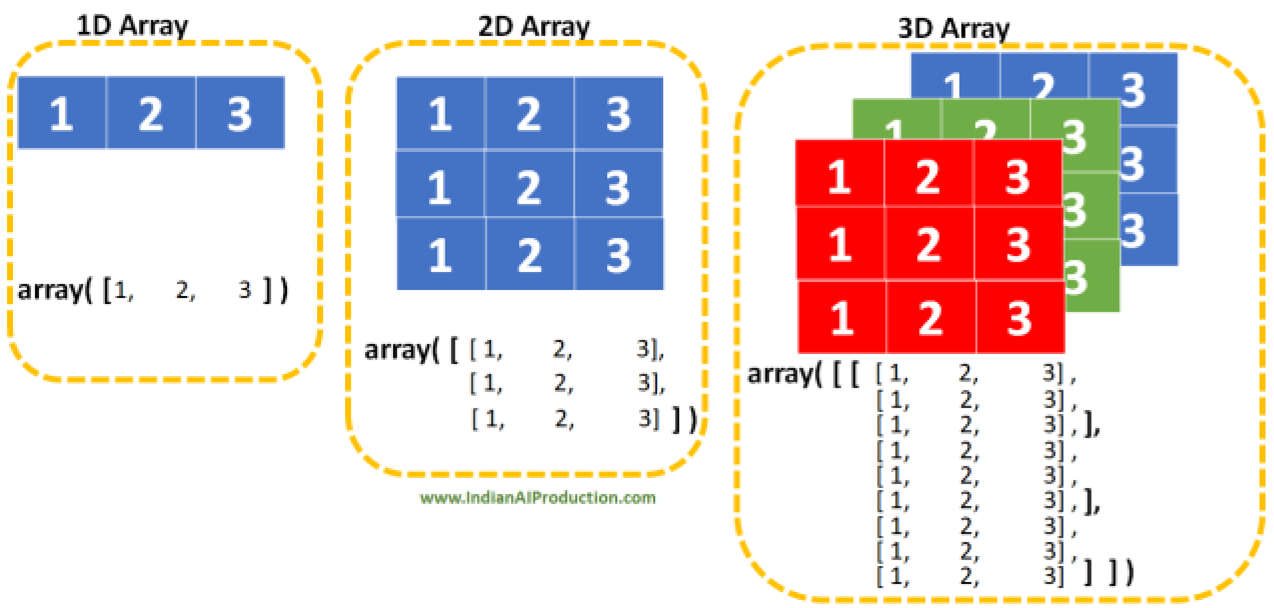
- Classification → **Which bucket?**
- Regression → **How much?**

If you create arrays incorrectly:

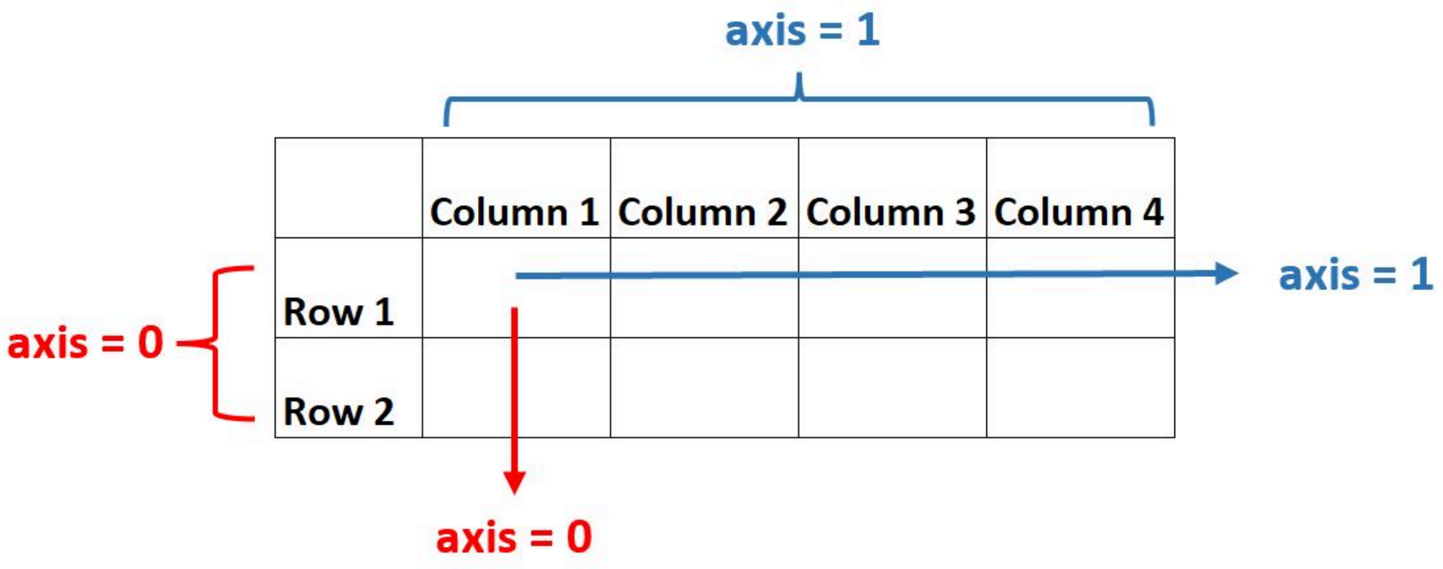
- Your model trains but learns garbage
- Broadcasting gives wrong results without errors
- Memory usage explodes

That’s why this topic matters.





1D Array
1 2 3
array([1, 2, 3])
2D Array
array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
3D Array
array([[[1, 2, 3], [1, 2, 3], [1, 2, 3],], [1, 2, 3], [1, 2, 3], [1, 2, 3],], [1, 2, 3], [1, 2, 3], [1, 2, 3]]])
www.IndianAIProduction.com



1. `np.array()`

What it is

Creates a NumPy array from **existing data** like lists or tuples.

Why it exists

Python lists are slow for math. NumPy arrays are:

- Fixed-type

- Contiguous in memory
- Vectorized

Basic example

```
import numpy as np

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

- Shape: `(3,)`
- Rows: none
- Columns: none
- This is a **1D array**, not a row or column matrix

2D example

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

- Shape: `(2, 3)`
- Rows: 2
- Columns: 3

Real data science usage

```
X = np.array([
    [180, 75],
    [165, 60],
    [170, 68]
])
```

- Each row = one sample
- Each column = one feature

Beginner mistakes

- Thinking `[1,2,3]` is a row vector. It is **not**
- Mixing data types unintentionally

```python
np.array([1, 2, 3.5])  # becomes float array
```

Interview trap

> "Is NumPy array dynamic like list?"

Correct answer: **No. Size is fixed. NumPy may allocate new memory when resizing.**

2. `np.zeros()`

What it is

Creates an array filled with **0s**.

Why it exists

Used for:

- Initialization
- Placeholders
- Accumulators
- ML weights starting point

Example

```python
z = np.zeros((3, 4))
```

- Shape: `(3, 4)`
- 3 rows, 4 columns
- dtype: `float64` by default

ML usage

```python
weights = np.zeros((5,))
```

Used when you want predictable starting values.

Beginner mistakes

- Forgetting shape must be a tuple

```
np.zeros(5)  # WRONG
```

Interview trap

> “Why not use Python lists instead?”

Answer: NumPy arrays are faster and memory-efficient for numerical ops.

3. `np.ones()`

What it is

Creates an array filled with **1s**.

Why it exists

- Bias initialization
- Testing pipelines
- Masking logic

Example

```
o = np.ones((2, 3))
```

ML usage

```
bias = np.ones((1,))
```

Common mistake

Thinking ones are safer defaults. They’re not. In ML, bad initialization can break convergence.

4. `np.empty()`

What it is

Creates an array **without initializing values**.

Important truth

This array contains **garbage values** from memory.

Example

```
e = np.empty((3, 3))
```

Why it exists

- Performance
- You plan to overwrite values immediately

Real usage

```
result = np.empty((1000000,))
# filled later in loop
```

Beginner mistake

Using `np.empty()` and assuming zeros. That's wrong and dangerous.

Interview trap

> "Why is np.empty faster?"

Because it **skips memory initialization**.

5. `np.arange()`

What it is

Creates values in a range, similar to `range()` but returns an array.

Example

```
a = np.arange(0, 10, 2)
```

Result: `[0 2 4 6 8]`

Shape

- `(5,)` → 1D array

Real usage

```
indices = np.arange(len(X))
```

Common mistake

Using floats:

```
np.arange(0, 1, 0.1)  # precision issues
```

Interview trap

> "Why prefer linspace for floats?"

Because arange can skip or duplicate due to floating point precision.

6. `np.linspace()`

What it is

Creates **evenly spaced values** between two limits.

Example

```
l = np.linspace(0, 1, 5)
```

Result:

```
[0. , 0.25, 0.5 , 0.75, 1.]
```

Why it exists

- Mathematical stability
- Plotting
- ML experiments

Real usage

```
learning_rates = np.linspace(0.001, 0.1, 10)
```

Interview trap

> "Difference between arange and linspace?"

Key answer:

- arange → step-based
- linspace → count-based

7. `np.eye()`

What it is

Creates an **identity matrix**.

Example

```
I = np.eye(3)
```

Result:

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

ML usage

- Regularization
- Linear algebra
- Covariance matrices

Beginner mistake

Confusing identity with diagonal matrices.

Math (only what you need)

- Shape = `(rows, columns)`
- Axis 0 → rows
- Axis 1 → columns
- Identity matrix preserves vector values in multiplication

That’s it. No theory dump.

Practice Questions (MANDATORY)

Easy (1–7)

1. What is the shape of `np.array([1,2,3])` ?
2. Difference between Python list and NumPy array?
3. What does `np.zeros((2,2))` return?
4. Default dtype of `np.ones()` ?
5. Is `np.empty()` initialized?
6. Output of `np.arange(1,5)` ?
7. Why must shape be a tuple?

Medium (8–14)

8. Predict shape: `np.array([[1,2],[3,4],[5,6]])`
9. Why is `np.arange(0,1,0.1)` unsafe?
10. When should `np.empty()` be used?
11. Difference between `(5,)` and `(5,1)` ?
12. What happens if mixed datatypes are passed?
13. Why is `np.zeros` common in ML?
14. How many elements in `np.zeros((4,3))` ?

Hard (15–20)

15. Why can bad initialization break gradient descent?
16. Memory difference between list and NumPy array?
17. Why identity matrices matter in linear models?
18. How wrong shapes silently break ML models?
19. When does `np.empty()` cause production bugs?
20. arange vs linspace in numerical stability?

Industry & Practical Usage (real tasks)

1. Creating feature matrices
2. Initializing model weights
3. Creating masks for missing values

4. Generating synthetic data for testing
5. Building confusion matrices
6. Creating batch indices
7. Speed-testing algorithms
8. Debugging shape mismatches
9. Preparing data for vectorized ops
10. Memory-efficient pipelines

Reality check (listen carefully)

If you don’t **visually think in shapes**, NumPy will destroy you in interviews and real work. Memorizing syntax is useless.