10. Shape Manipulation (NumPy)

1. Topic Overview

What this topic is

Shape manipulation means changing how a NumPy array is structured.
The data does not change.
Only the arrangement changes.

Why it exists

Data rarely comes in the shape models need.
NumPy lets you reshape data without copying it.

One real-world analogy

Same LEGO blocks.
You rebuild them into a different shape.

2. Core Theory (Deep but Clear)

NumPy arrays have:

- **shape** → how data is arranged
- **dtype** → data type
- **memory buffer** → actual stored values

Shape manipulation only changes **how NumPy views the memory**.

Sub-topics Covered

1. `reshape`

2. `flatten`
3. `ravel`
4. `transpose` / `.T`
5. `swapaxes`
6. `expand_dims`
7. `squeeze`

How NumPy Thinks Internally

- Data is stored in **1D contiguous memory**
- Shape is just **metadata**
- Many operations return **views**, not copies
- Wrong shapes break ML pipelines

3. Syntax & Examples

3.1 `reshape`

Purpose

Change array shape without changing data.

Rule

Total elements must stay the same.

Syntax

```
np.reshape(array, new_shape)
array.reshape(new_shape)
```

Example 1

```
import numpy as np

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

Output

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

Explanation

- Original shape: `(6,)`
- New shape: `(2, 3)`
- 2 × 3 = 6 elements

Example 2 (Using `-1`)

```
c = a.reshape((3, -1))
print(c)
```

Output

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

Explanation

- `-1` tells NumPy to auto-calculate
- Useful when size is unknown

3.2 `flatten`

Purpose

Convert array to 1D **copy**.

Syntax

```
array.flatten()
```

Example

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

Output

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

Explanation

- Always returns a new array
- Safe but uses more memory

3.3 `ravel`

Purpose

Convert array to 1D **view when possible**.

Syntax

```
array.ravel()
```

Example

```
z = x.ravel()
print(z)
```

Output

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

Explanation

- Faster than `flatten`
- May share memory with original array

3.4 `transpose / .T`

Purpose

Swap rows and columns.

Syntax

```
array.T
np.transpose(array)
```

Example

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

Output

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

Explanation

- Shape changes from `(2, 3)` to `(3, 2)`
- Used heavily in linear algebra

3.5 `swapaxes`

Purpose

Swap any two axes.

Syntax

```
np.swapaxes(array, axis1, axis2)
```

Example

```
a = np.zeros((2, 3, 4))
b = np.swapaxes(a, 0, 2)
print(b.shape)
```

Output

```
(4, 3, 2)
```

Explanation

- Needed for deep learning tensors

3.6 `expand_dims`

Purpose

Add a new axis.

Syntax

```
np.expand_dims(array, axis)
```

Example

```
v = np.array([1, 2, 3])
print(np.expand_dims(v, axis=0).shape)
print(np.expand_dims(v, axis=1).shape)
```

Output

```
(1, 3)
(3, 1)
```

Explanation

- Required for batch dimensions

3.7 `squeeze`

Purpose

Remove axes of size 1.

Syntax

```
np.squeeze(array)
```

Example

```
s = np.array([[[5, 6, 7]]])
print(s.shape)
print(np.squeeze(s).shape)
```

Output

```
(1, 1, 3)
(3,)
```

Explanation

- Dangerous if you remove wrong axis
- Always check shape

4. Why This Matters in Data Science

Data Cleaning

- Fix wrong column or row orientation
- Flatten extracted features

Feature Engineering

- Convert features to `(n_samples, n_features)`
- Add batch dimension

Model Input Preparation

- ML models expect strict shapes
- DL models require exact tensor rank

ML / DL Pipelines

- Shape mismatch causes runtime errors
- Silent bugs ruin training

If you don’t understand this

- Models crash
- Training fails
- Wrong predictions with no error

5. Common Mistakes (VERY IMPORTANT)

1. Reshaping to incompatible size
   Happens when element count mismatches
   Always check `array.size`

2. Confusing `flatten` and `ravel`
   Leads to unintended data changes
   Use `flatten` when safety matters
3. Using `.T` on 1D arrays
   No effect
   Convert to 2D first
4. Removing wrong axis with `squeeze`
   Breaks batch dimension
   Use `axis` argument when possible
5. Forgetting batch dimension
   Models expect `(n, features)`
   Use `expand_dims`

6. Performance & Best Practices

- `reshape`, `transpose`, `ravel` are **fast**
- They return **views** when possible
- `flatten` copies data. Slower.
- Avoid reshaping inside training loops
- Always print `.shape` during debugging

7. 20 Practice Problems (NO SOLUTIONS)

Easy (5)

1. Reshape a 1D array of 12 elements into `(3, 4)`
2. Convert a 2D array into 1D using `ravel`
3. Add a batch dimension to `(10,)`
4. Transpose a `(5, 2)` array
5. Remove size-1 dimensions from `(1, 3, 1)`

Medium (7)

6. Prepare feature matrix from raw sensor data
7. Convert `(100, 28, 28)` images to `(100, 784)`
8. Swap channels axis in image tensor
9. Fix wrongly oriented CSV-loaded array
10. Add channel dimension to grayscale images
11. Flatten model output safely
12. Debug a shape mismatch error

Hard (5)

13. Reshape time-series data for LSTM
14. Convert predictions back to original shape
15. Handle variable batch sizes
16. Prevent accidental data copy
17. Align matrix shapes for dot product

Industry-Level (3)

18. Prepare data for CNN input pipeline
19. Fix broken production model due to shape drift
20. Optimize memory usage in large reshapes

8. Mini Checklist

- Shape is metadata, not data
- Total elements must match
- Use `-1` wisely
- Prefer views over copies
- Always print `.shape`
- Models are strict about dimensions

10. Shape Manipulation (Additional – MUST KNOW)

These are used daily in real pipelines.
Skipping them will break feature assembly and batching logic.

1. Topic Overview

What this part is

These operations **combine, split, repeat, or align arrays by shape**.
They do not change values.
They change how arrays fit together.

Why it exists

Real datasets come from multiple sources.
Models need one clean, aligned array.

One real-world analogy

Combining multiple Excel sheets into one table.

2. Core Theory (Deep but Clear)

NumPy works with:

- **axis** → direction of operation
- **shape compatibility** → required for joining
- **memory layout** → affects speed

Most errors here are **axis mistakes**.

Sub-topics Covered (NEW ONLY)

1. `concatenate`
2. `stack`
3. `hstack` , `vstack`
4. `split` , `hsplit` , `vsplit`
5. `tile`
6. `repeat`
7. `broadcast_to`
8. `atleast_1d` , `atleast_2d` , `atleast_3d`

3. Syntax & Examples

3.1 `concatenate`

Purpose

Join arrays along an existing axis.

Syntax

```
np.concatenate((a, b), axis=0)
```

Example

```
import numpy as np

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

c = np.concatenate((a, b), axis=0)
print(c)
```

Output

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

Explanation

- Axis 0 means row-wise join
- Columns must match

3.2 `stack`

Purpose

Join arrays along a **new axis**.

Syntax

```
np.stack((a, b), axis=0)
```

Example

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

z = np.stack((x, y), axis=0)
print(z)
print(z.shape)
```

Output

```
[[1 2 3]
 [4 5 6]]
(2, 3)
```

Explanation

- New dimension created
- Used for batching

3.3 `hstack` and `vstack`

Purpose

Quick horizontal or vertical stacking.

Syntax

```
np.hstack((a, b))
np.vstack((a, b))
```

Example

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

print(np.hstack((a, b)))
print(np.vstack((a.T, b.T)))
```

Output

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

Explanation

- `hstack` → column-wise
- `vstack` → row-wise

3.4 `split` , `hsplit` , `vsplit`

Purpose

Break array into smaller arrays.

Syntax

```
np.split(array, indices, axis)
```

Example

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

for p in parts:
    print(p)
```

Output

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

Explanation

- Used for train/validation splits

3.5 `tile`

Purpose

Repeat whole array.

Syntax

```
np.tile(array, reps)
```

Example

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

Output

```
[1 2 1 2 1 2]
```

Explanation

- Used in synthetic data creation

3.6 `repeat`

Purpose

Repeat individual elements.

Syntax

```
np.repeat(array, repeats)
```

Example

```
print(np.repeat([1, 2], 3))
```

Output

```
[1 1 1 2 2 2]
```

Explanation

- Different from `tile`
- Element-level repetition

3.7 `broadcast_to`

Purpose

Force array to behave like larger shape.

Syntax

```
np.broadcast_to(array, shape)
```

Example

```
a = np.array([1, 2, 3])
b = np.broadcast_to(a, (4, 3))
print(b)
```

Output

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

Explanation

- No real copy
- Read-only view

3.8 `atleast_1d , atleast_2d , atleast_3d`

Purpose

Guarantee minimum dimensions.

Syntax

```
np.atleast_2d(array)
```

Example

```
x = np.array(5)
print(np.atleast_1d(x).shape)
print(np.atleast_2d(x).shape)
```

Output

```
(1,)
(1, 1)
```

Explanation

- Safe input handling for models

4. Why This Matters in Data Science

Data Cleaning

- Merge multiple data sources
- Split datasets cleanly

Feature Engineering

- Combine feature blocks
- Repeat categorical encodings

Model Input Preparation

- Stack batches
- Align labels with features

ML / DL Pipelines

- Broadcasting saves memory
- Wrong axis breaks training

If you don’t know this

- Silent shape bugs
- Wrong data fed to models
- Training looks fine but fails

5. Common Mistakes (VERY IMPORTANT)

1. Wrong axis in `concatenate`
   Leads to invalid feature matrix
2. Using `stack` instead of `concatenate`
   Adds unwanted dimension
3. Confusing `tile` and `repeat`
   Causes data duplication bugs
4. Broadcasting writable arrays
   Causes runtime errors
5. Unequal shapes during stacking
   Always print shapes before joining

6. Performance & Best Practices

- `broadcast_to` is memory efficient
- `concatenate` copies data
- Avoid stacking inside loops
- Validate shapes before merge
- Prefer vectorized stacking

7. 20 Practice Problems (NO SOLUTIONS)

Easy (5)

1. Concatenate two `(5, 3)` arrays row-wise
2. Stack three 1D arrays into a batch
3. Split array into equal chunks
4. Use `atleast_2d` on scalar input
5. Repeat labels for augmented data

Medium (7)

6. Combine numerical and encoded features
7. Broadcast bias vector to batch
8. Create synthetic samples using `tile`
9. Split dataset into train/val/test
10. Stack image channels correctly
11. Align predictions with samples
12. Fix column mismatch during merge

Hard (5)

13. Assemble batch tensor for training
14. Avoid memory explosion during concat
15. Debug wrong axis error
16. Convert ragged inputs safely
17. Optimize stacking in pipeline

Industry-Level (3)

18. Feature union in production pipeline
19. Batch preparation for GPU training
20. Fix broken model after data source change

8. Mini Checklist

- Use `concatenate` for existing axes
- Use `stack` for new axes
- Axis choice is critical
- Broadcasting saves memory
- Always check final shape