14. Sorting & Searching (NumPy)

1. Topic Overview

What this topic is

Sorting and searching in NumPy means:

- Reordering array elements.
- Finding positions of elements.
- Finding top, bottom, or specific values.

Why it exists

Data is rarely in the order you need.
Models and analysis often need:

- Sorted values.
- Indices of important values.
- Fast lookup.

Real-world analogy (ONE)

Think of exam marks in a class.
You sort marks.
You search for the topper.
You search for a specific student’s rank.

2. Core Theory (Deep but Clear)

NumPy sorting and searching works on **ndarrays**.
NumPy always thinks in:

- **shape**: dimensions of data
- **dtype**: type of values
- **memory**: contiguous blocks

Main Sub-Topics

1. `np.sort`
2. `np.argsort`
3. `np.lexsort`
4. `np.searchsorted`
5. `np.where`
6. `np.argmax` / `np.argmin`
7. `np.partition` / `np.argpartition`

2.1 `np.sort`

- Returns a **sorted copy**.
- Original array is unchanged.
- Sorting is done along an axis.

Internally:

- NumPy sorts values **inside contiguous memory blocks**.
- Axis decides which block is sorted.

2.2 `np.argsort`

- Returns **indices** that would sort the array.
- Actual values are not returned.

Internally:

- NumPy sorts indices based on value comparison.
- Very important for labels and features.

2.3 `np.lexsort`

- Sorts using **multiple keys**.
- Last key has highest priority.

Internally:

- Think column-wise sorting.
- Uses stable sorting.

2.4 `np.searchsorted`

- Finds insertion index in a **sorted array**.
- Does NOT sort the array.

Internally:

- Uses binary search.
- Very fast.

2.5 `np.where`

- Conditional search.
- Returns indices where condition is True.

Internally:

- Boolean mask + index extraction.

2.6 `np.argmax / np.argmin`

- Returns index of max or min value.
- Flattens array if axis not given.

Internally:

- Linear scan over memory.

2.7 `np.partition` / `np.argpartition`

- Partial sorting.
- Only important elements are placed correctly.
- Faster than full sort.

Internally:

- Does not guarantee full order.
- Uses selection algorithms.

3. Syntax & Examples

3.1 `np.sort`

Syntax

```
np.sort(arr, axis=-1)
```

Example 1

```
import numpy as np

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

Output

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

Explanation

- Array is copied.
- Values are sorted in ascending order.

Example 2 (2D)

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

print(np.sort(b, axis=0))
```

Output

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

Explanation

- Sorting happens column-wise.
- Each column sorted independently.

3.2 `np.argsort`

Syntax

```
np.argsort(arr)
```

Example

```
a = np.array([50, 10, 40])
print(np.argsort(a))
```

Output

```
[1 2 0]
```

Explanation

- Index 1 has smallest value (10).
- Index 2 has next (40).
- Index 0 has largest (50).

3.3 `np.lexsort`

Syntax

```
np.lexsort((key2, key1))
```

Example

```
names = np.array(['A', 'B', 'A'])
scores = np.array([90, 85, 88])

idx = np.lexsort((scores, names))
print(idx)
```

Output

```
[0 2 1]
```

Explanation

- Primary sort: names
- Secondary sort: scores

3.4 `np.searchsorted`

Syntax

```
np.searchsorted(sorted_arr, value)
```

Example

```
a = np.array([10, 20, 30, 40])
print(np.searchsorted(a, 25))
```

Output

```
2
```

Explanation

- 25 should be inserted at index 2.
- Array remains sorted.

3.5 `np.where`

Syntax

```
np.where(condition)
```

Example

```
a = np.array([5, 10, 15, 20])
print(np.where(a > 10))
```

Output

```
(array([2, 3]),)
```

Explanation

- Values greater than 10 are at indices 2 and 3.

3.6 `np.argmax / np.argmin`

Example

```
a = np.array([3, 7, 1, 9])
print(np.argmax(a))
print(np.argmin(a))
```

Output

```
3
2
```

Explanation

- Max value 9 at index 3.
- Min value 1 at index 2.

3.7 `np.partition`

Syntax

```
np.partition(arr, k)
```

Example

```
a = np.array([9, 1, 5, 3, 7])
print(np.partition(a, 2))
```

Output

```
[3 1 5 9 7]
```

Explanation

- Element at index 2 is correct.
- Left side smaller.
- Right side larger.
- Order is not sorted.

4. Why This Matters in Data Science

Data Cleaning

- Detect min/max values.
- Find outliers using sorting.

Feature Engineering

- Rank features.
- Select top-k values.

Model Input Preparation

- Align features with labels using `argsort`.
- Sort time-series data.

ML / DL Pipelines

- Top-k predictions.
- Threshold based decisions.

What breaks if you don’t understand this

- Labels get mismatched.
- Wrong samples selected.
- Performance becomes slow.
- Models train on wrong data.

5. Common Mistakes (VERY IMPORTANT)

1. Sorting labels instead of indices
   Reason: Using `sort` instead of `argsort`
   Fix: Use `argsort` to reorder related arrays.
2. Using `searchsorted` on unsorted array
   Reason: Forgetting it assumes sorted input
   Fix: Always sort first.

3. Assuming `partition` fully sorts data
   Reason: Misunderstanding partial sort
   Fix: Use `sort` when full order is needed.
4. Ignoring axis in 2D arrays
   Reason: Default axis confusion
   Fix: Always specify axis explicitly.
5. Using Python `sorted()` on NumPy arrays
   Reason: Habit from lists
   Fix: Use NumPy functions only.

6. Performance & Best Practices

- `np.sort` is fast for full ordering.
- `np.partition` is faster for top-k tasks.
- `searchsorted` is very fast for lookup.
- Sorting large arrays is memory heavy.
- Avoid sorting inside loops.
- Prefer vectorized sorting.

7. 20 Practice Problems (NO SOLUTIONS)

Easy (5)

1. Sort a 1D array of integers.
2. Sort each column of a 2D array.
3. Find index of maximum value.
4. Find indices where values are negative.
5. Insert a value in a sorted array using search index.

Medium (7)

6. Sort feature values and reorder labels.
7. Find top 3 highest values using partition.

8. Rank students based on marks.
9. Sort time-series data by timestamps.
10. Find positions of missing values.
11. Sort rows based on second column.
12. Use lexsort for multi-column data.

Hard (5)

13. Select top-k probabilities from model output.
14. Find bottom-k loss values.
15. Align shuffled predictions with true labels.
16. Find percentile thresholds using sorting.
17. Detect outliers using sorted distances.

Industry-Level Tasks (3)

18. Rank users by engagement score.
19. Select top products per category.
20. Insert streaming data into sorted array efficiently.

8. Mini Checklist

- `sort` returns values.
- `argsort` returns indices.
- `lexsort` handles multiple keys.
- `searchsorted` needs sorted input.
- `partition` is not full sort.
- Always think about axis and shape.
- Sorting affects performance.

These are not “extra for fun”.
These are used in **EDA, feature engineering, ML pipelines, and production code**.

1. Topic Overview

What this topic is

These tools help you:

- Find unique values
- Count occurrences
- Check membership
- Detect duplicates
- Compare arrays efficiently

Why it exists

Real data is:

- Repetitive
- Categorical
- Noisy
- Large

You must analyze value distribution fast.

Real-world analogy (ONE)

Think of customer IDs.
You need:

- Unique customers
- How many times each appears
- Whether a customer exists in a list

2. Core Theory (Deep but Clear)

NumPy provides **search-style utilities** that work on:

- Value comparison
- Hash-like grouping
- Vectorized boolean logic

All of them operate on:

- **ndarray**
- **dtype-aware comparisons**
- **contiguous memory scans**

Mandatory Sub-Topics (New Only)

1. `np.unique`
2. `np.bincount`
3. `np.isin`
4. `np.in1d`
5. `np.setdiff1d`
6. `np.intersect1d`
7. `np.union1d`

2.1 `np.unique`

- Finds unique values.
- Can return:
  - counts
  - inverse mapping
  - indices

Internally:

- Sorts data.
- Then scans sequential memory.
- Very efficient for categorical data.

2.2 `np.bincount`

- Counts frequency of **non-negative integers**.
- Extremely fast.
- Used for labels and classes.

Internally:

- Direct index-based counting.
- No comparisons needed.

2.3 `np.isin`

- Checks membership.
- Vectorized version of `in`.

Internally:

- Uses broadcasting.
- Outputs boolean mask.

2.4 `np.in1d`

- Same logic as `isin`.
- Returns flattened boolean array.

Internally:

- Linear scan.
- Often used in pipelines.

2.5 `np.setdiff1d`

- Elements in A but not in B.
- Set-style operation.

Internally:

- Sort + compare.

2.6 `np.intersect1d`

- Common elements between arrays.

Internally:

- Sorted comparison.

2.7 `np.union1d`

- All unique elements from both arrays.

Internally:

- Merge + unique scan.

3. Syntax & Examples

3.1 `np.unique`

Syntax

```
np.unique(arr, return_counts=False)
```

Example

```
import numpy as np

a = np.array([1, 2, 2, 3, 3, 3])
vals, counts = np.unique(a, return_counts=True)

print(vals)
print(counts)
```

Output

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

Explanation

- `1` appears once
- `2` appears twice
- `3` appears three times

3.2 `np.bincount`

Syntax

```
np.bincount(arr)
```

Example

```
labels = np.array([0, 1, 1, 2, 2, 2])
print(np.bincount(labels))
```

Output

```
[1 2 3]
```

Explanation

- Index = label
- Value = count

3.3 `np.isin`

Syntax

```
np.isin(arr, test_elements)
```

Example

```
a = np.array([10, 20, 30, 40])
print(np.isin(a, [20, 40]))
```

Output

```
[False  True False  True]
```

Explanation

- True where value exists in test list

3.4 `np.in1d`

Example

```
a = np.array([1, 2, 3, 4])
print(np.in1d(a, [2, 4]))
```

Output

```
[False  True False  True]
```

Explanation

- Same idea as `isin`
- Always returns 1D result

3.5 `np.setdiff1d`

Example

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

print(np.setdiff1d(a, b))
```

Output

```
[1 3]
```

Explanation

- Elements only in `a`

3.6 `np.intersect1d`

Example

```
print(np.intersect1d(a, b))
```

Output

```
[2 4]
```

3.7 `np.union1d`

Example

```
print(np.union1d(a, b))
```

Output

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

4. Why This Matters in Data Science

Data Cleaning

- Remove duplicates
- Detect unseen categories
- Count missing labels

Feature Engineering

- Encode categorical features
- Class frequency analysis
- Rare category detection

Model Input Preparation

- Validate label ranges
- Check class imbalance
- Align train-test categories

ML / DL Pipelines

- Class weighting
- Sampling strategies
- Monitoring data drift

What breaks if you don’t know this

- Wrong class counts
- Silent data leakage
- Broken encoders
- Invalid model assumptions

5. Common Mistakes (VERY IMPORTANT)

1. Using `bincount` with negative values
   Reason: It only supports non-negative ints
   Fix: Remap labels first
2. Assuming `unique` preserves order
   Reason: It sorts by default
   Fix: Track indices if order matters
3. Using Python `set` instead of NumPy
   Reason: Habit
   Fix: Use NumPy for vectorization
4. Forgetting dtype in membership checks
   Reason: String vs int mismatch
   Fix: Always verify dtype
5. Using `in` inside loops
   Reason: Python thinking
   Fix: Use `isin` or `in1d`

6. Performance & Best Practices

- `bincount` is fastest for label counts
- `unique` is efficient but sorts data
- `isin` scales well for large arrays
- Avoid Python loops for membership checks
- Always prefer vectorized operations

7. 20 Practice Problems (NO SOLUTIONS)

Easy (5)

1. Find unique values in a column.
2. Count class labels.
3. Check if values exist in another array.
4. Remove duplicate IDs.
5. Find common elements between arrays.

Medium (7)

6. Detect unseen categories in test data.
7. Find rare classes below threshold.
8. Validate label encoding.
9. Compare feature sets of two datasets.
10. Count frequency of predicted classes.
11. Filter rows using membership mask.
12. Align categorical levels across datasets.

Hard (5)

13. Detect data drift using category frequency.
14. Identify missing training categories.
15. Build class imbalance report.
16. Filter invalid predictions.
17. Validate multi-class outputs.

Industry-Level Tasks (3)

18. Monitor category explosion in production.
19. Validate real-time incoming labels.
20. Build frequency-based alert system.

8. Mini Checklist

- `unique` finds values and counts
- `bincount` counts labels fast
- `isin` replaces Python `in`
- Set operations are vectorized
- Order may change after unique
- Always think dtype and memory

Hard truth (coach mode)

If you skip these:

- Your EDA will be weak
- Your feature pipelines will break
- Your ML debugging will be slow