

# 4. NumPy Indexing & Slicing

## 1. Topic Overview

### What this topic is

Indexing and slicing mean selecting parts of a NumPy array.

### Why it exists

Data is large. You never use all data at once.

You must pick exact values, rows, or columns.

### One real-world analogy

Think of an Excel sheet.

Row number + column letter tells you one cell.

Indexing works the same way.

## 2. Core Theory (Deep but Clear)

### How NumPy thinks internally

- NumPy array is:
  - Continuous memory block
  - Fixed data type
  - Known shape

Example:

```
shape = (rows, columns)
```

Indexing is **position-based**, not label-based.

Index always starts from **0**.

## Sub-topic 1: 1D Indexing

- Used for vectors
- Array has only one axis
- Axis = 0

Example shape:

(5, )

Index points directly to memory location.

## Sub-topic 2: 2D Indexing [row, column]

- Used for matrices
- Two axes:
  - Axis 0 → rows
  - Axis 1 → columns

Example shape:

(3, 4)

Syntax:

```
array[row_index, column_index]
```

NumPy first moves to row, then column.

## Sub-topic 3: Row Slicing

- Select multiple rows
- Column remains fixed or all

Slice format:

```
start : stop
```

Stop is **excluded**.

## Sub-topic 4: Column Slicing

- Select columns
- Rows remain fixed or all

Use `:` for rows.

## Sub-topic 5: Step Slicing

- Skip values
- Pattern selection

Format:

```
start : stop : step
```

## Sub-topic 6: Negative Indexing

- Index from end
- `-1` means last element

Works in all dimensions.

## Sub-topic 7: Relation to Pandas `.loc` and `.iloc`

- NumPy indexing = Pandas `.iloc`
- Both are:
  - Integer based

- Position based

Difference:

- NumPy has no labels
- Pandas `.loc` uses labels

## 3. Syntax & Examples

### 1D Indexing

#### Syntax

```
arr[index]
```

#### Example 1

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])
print(arr[2])
```

#### Output

```
30
```

#### Explanation

- Index 2 → third element
- Indexing starts from 0

#### Example 2

```
print(arr[-1])
```

## Output

50

## Explanation

- $-1 \rightarrow$  last element

# 2D Indexing [row, column]

## Syntax

`arr[row, col]`

## Example 1

```
mat = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
print(mat[1, 2])
```

## Output

6

## Explanation

- Row index 1  $\rightarrow$  second row
- Column index 2  $\rightarrow$  third column

## Example 2

```
print(mat[0, 0])
```

### Output

1

## Row Slicing

### Syntax

```
arr[start_row:stop_row, :]
```

### Example

```
print(mat[0:2, :])
```

### Output

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

### Explanation

- Rows 0 and 1 selected
- : means all columns

## Column Slicing

### Syntax

```
arr[:, start_col:stop_col]
```

## Example

```
print(mat[:, 1:3])
```

## Output

```
[[2 3]
 [5 6]
 [8 9]]
```

## Explanation

- All rows
- Columns 1 and 2

# Step Slicing

## Syntax

```
arr[start:stop:step]
```

## Example

```
x = np.array([0,1,2,3,4,5,6,7,8,9])
print(x[0:10:2])
```

## Output

```
[0 2 4 6 8]
```

## Explanation

- Step = 2
- Every second element

# Negative Indexing

## Example

```
print(mat[-1, :])
```

## Output

```
[7 8 9]
```

## Explanation

- Last row selected

# 4. Why This Matters in Data Science

## Data Cleaning

- Remove bad rows
- Select valid columns
- Filter ranges

## Feature Engineering

- Select feature columns
- Drop target column
- Create subsets

## Model Input Preparation

- X and y split
- Train-test slicing
- Batch creation

## ML / DL Pipelines

- Mini-batch indexing

- Time series windows
- Masking operations

### What breaks if you don't know this

- Wrong data goes into model
- Shape mismatch errors
- Silent bugs
- Wrong predictions

## 5. Common Mistakes (VERY IMPORTANT)

1. Confusing row and column order
  - NumPy uses [row, column]
2. Forgetting zero-based indexing
  - First element is index 0
3. Assuming slice includes stop value
  - Stop is always excluded
4. Mixing NumPy indexing with Pandas .loc
  - .loc is label based
5. Shape collapse when selecting single row
  - Use slicing to preserve shape

## 6. Performance & Best Practices

- Indexing is fast
- Slicing returns **views**, not copies
- Modifying slice changes original array

Be careful:

- Fancy indexing creates copies
- Large slices increase memory use

Always check:

```
arr.shape
```

## 7. Practice Problems (NO SOLUTIONS)

### Easy (5)

1. Select the 3rd element from a 1D array
2. Get last element using negative indexing
3. Select first row from 2D array
4. Select second column from matrix
5. Slice first 5 elements from array

### Medium (7)

6. Select rows 2 to 5 from dataset matrix
7. Extract feature columns except target
8. Select every alternate value from signal data
9. Get last 3 rows from time-series array
10. Select middle column from odd-column matrix
11. Slice training data using index range
12. Remove first column from feature matrix

### Hard (5)

13. Create sliding window using slicing
14. Extract diagonal block from matrix
15. Split dataset into X and y using indexing
16. Batch data using step slicing
17. Reverse array using slicing

## Industry-Level Tasks (3)

18. Select last 30 days from stock price matrix
19. Extract sensor channels from IoT data array
20. Prepare mini-batches for neural network input

## 8. Mini Checklist

- Index starts from 0
- Shape decides valid indexing
- [row, column] order
- Stop index is excluded
- Slicing returns views
- NumPy indexing = Pandas .iloc

# Advanced NumPy Indexing & Slicing

## 1. Topic Overview

### What this topic is

Advanced ways to select data from NumPy arrays using conditions, index arrays, and masks.

### Why it exists

Real datasets are not clean ranges.

You select data based on rules, not positions.

### One real-world analogy

Filtering Excel rows using conditions like

`salary > 50000 and age < 30 .`

## 2. Core Theory (Deep but Clear)

### How NumPy thinks internally

- Array is stored in continuous memory
- Indexing decides:
  - which memory locations to read
  - whether data is a view or a copy

Advanced indexing usually creates **copies**, not views.

### Sub-topic 1: Boolean Indexing (MOST IMPORTANT)

#### What it is

- Indexing using `True / False` mask
- Mask length must match axis length

#### Internal logic

- NumPy converts condition into boolean array
- `True` → keep value
- `False` → drop value

### Sub-topic 2: Masking with Conditions

- Boolean indexing applied on conditions
- Used heavily in data cleaning

### Sub-topic 3: Fancy Indexing (Index Arrays)

- Index using list or array of indices

- Order is controlled manually

Creates a **copy**, not view.

## **Sub-topic 4: np.where() Indexing**

- Conditional selection
- Vectorized if-else

Returns:

- indices
- or values based on usage

## **Sub-topic 5: np.take() and np.take\_along\_axis()**

- Safe indexing along axis
- Used in pipelines and production code

## **Sub-topic 6: np.ix\_() for Cartesian Indexing**

- Used to select row-column combinations
- Prevents shape mismatch

## **Sub-topic 7: Ellipsis ( ...) Indexing**

- Used for high-dimensional arrays
- Replaces multiple : symbols

# 3. Syntax & Examples

## Boolean Indexing

### Syntax

```
arr[condition]
```

### Example 1

```
import numpy as np

x = np.array([10, 25, 30, 5, 8])
print(x[x > 10])
```

### Output

```
[25 30]
```

### Explanation

- Condition creates boolean mask
- Only True values selected

### Example 2

```
print(x[x % 2 == 0])
```

### Output

```
[10 30 8]
```

# Masking with 2D Arrays

```
data = np.array([
    [100, 200],
    [50, 300],
    [400, 150]
])

print(data[data[:, 0] > 100])
```

## Output

```
[[400 150]]
```

## Explanation

- Condition applied on first column
- Full rows selected

# Fancy Indexing

## Syntax

```
arr[[i1, i2, i3]]
```

## Example

```
x = np.array([10, 20, 30, 40])
print(x[[0, 3, 1]])
```

## Output

```
[10 40 20]
```

## Explanation

- Order follows index list

- Creates copy

## np.where()

### Syntax

```
np.where(condition, value_if_true, value_if_false)
```

### Example

```
x = np.array([5, 15, 25])
y = np.where(x > 10, 1, 0)
print(y)
```

### Output

```
[0 1 1]
```

### Explanation

- Acts like vectorized if-else

## np.take()

```
x = np.array([100, 200, 300])
print(np.take(x, [2, 0]))
```

### Output

```
[300 100]
```

## `np.ix_()`

```
mat = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

print(mat[np.ix_([0, 2], [1, 2])])
```

### **Output**

```
[[2 3]
 [8 9]]
```

### **Explanation**

- Selects row-column combinations safely

## **Ellipsis ( ... )**

```
x = np.random.rand(2, 3, 4)
print(x[..., 1].shape)
```

### **Output**

```
(2, 3)
```

### **Explanation**

- ... replaces missing axes

## 4. Why This Matters in Data Science

### Data Cleaning

- Remove outliers
- Drop invalid rows
- Replace values conditionally

### Feature Engineering

- Binary flags using conditions
- Feature masking
- Column filtering

### Model Input Preparation

- Create labels
- Balance classes
- Batch filtering

### ML / DL Pipelines

- Mask padding tokens
- Attention masks
- Loss masking

#### If you don't know this

- You will write loops
- Code will be slow
- Bugs will be silent
- Models will learn wrong patterns

## 5. Common Mistakes (VERY IMPORTANT)

1. Mask shape mismatch
2. Expecting view from boolean indexing

3. Using Python loops instead of masks
4. Mixing fancy and slice indexing incorrectly
5. Forgetting axis while masking 2D arrays

## 6. Performance & Best Practices

- Boolean indexing is fast
- Fancy indexing creates copies
- Avoid loops
- Check memory usage
- Prefer vectorized conditions

## 7. Practice Problems (NO SOLUTIONS)

### Easy (5)

1. Select values greater than mean
2. Filter even numbers
3. Mask negative values
4. Select rows where first column > 50
5. Replace values less than 0 with 0

### Medium (7)

6. Create binary target using threshold
7. Filter dataset by multiple conditions
8. Select random rows using fancy indexing
9. Mask missing values
10. Extract rows using index array
11. Convert categorical labels to binary
12. Select top-k values using indexing

## Hard (5)

13. Implement class balancing using masks
14. Create attention mask for sequence data
15. Filter time-series by date index logic
16. Apply conditional scaling to features
17. Combine boolean and fancy indexing

## Industry-Level Tasks (3)

18. Remove outliers using IQR mask
19. Prepare padded batches with masks
20. Build rule-based data filtering pipeline

## 8. Mini Checklist

- Boolean indexing is mandatory
- Mask shape must match
- Fancy indexing creates copies
- `np.where` replaces loops
- `np.ix_` avoids shape bugs
- Ellipsis used in high-dim data

# Pandas vs NumPy Indexing

## 1. Topic Overview

### What this topic is

Comparison of how NumPy and Pandas select data.

## Why it exists

Most real projects use **both** NumPy and Pandas.

Confusing their indexing rules causes wrong data selection.

## One real-world analogy

NumPy is like a numbered locker system.

Pandas is like lockers with **names and numbers**.

## 2. Core Theory (Deep but Clear)

### Fundamental Difference

Aspect	NumPy	Pandas
Index type	Position only	Labels + position
Default index	Implicit	Explicit
Data structure	Array	Series / DataFrame
Focus	Speed, memory	Data meaning

## How NumPy Thinks

- No labels
- Only integer positions
- Fixed shape
- Continuous memory
- Indexing returns:
  - scalar
  - view
  - or copy

NumPy **does not know column names**.

# How Pandas Thinks

- Each axis has an index
- Index has meaning
- Data aligned by labels
- Built on top of NumPy

Pandas **cares about meaning**, not just position.

## 3. Indexing Interfaces

### NumPy

Only one main method:

```
arr[row, column]
```

Everything is position-based.

### Pandas

Three main methods:

Method	Type	Based on
[]	Mixed	Context
.iloc	Integer	Position
.loc	Label	Index name

## 4. Syntax & Examples (Side-by-Side)

# Example Dataset

```
import numpy as np
import pandas as pd

np_arr = np.array([
    [10, 20, 30],
    [40, 50, 60]
])

df = pd.DataFrame(
    np_arr,
    columns=["A", "B", "C"],
    index=["row1", "row2"]
)
```

## Selecting a Single Value

### NumPy

```
np_arr[1, 2]
```

### Output

60

### Explanation

- Row 1
- Column 2

### Pandas .iloc

```
df.iloc[1, 2]
```

### Output

60

## Explanation

- Same as NumPy
- Position based

## Pandas .loc

```
df.loc["row2", "C"]
```

## Output

60

## Explanation

- Label based
- Independent of order

## Row Selection

### NumPy

```
np_arr[0]
```

## Output

```
[10 20 30]
```

## Pandas

```
df.loc["row1"]
```

### Output

```
A    10  
B    20  
C    30
```

### Explanation

- Pandas returns Series
- Index preserved

## Column Selection

### NumPy

```
np_arr[:, 1]
```

### Output

```
[20 50]
```

## Pandas

```
df["B"]
```

### Output

```
row1    20  
row2    50
```

# Slicing Rows

## NumPy

```
np_arr[0:2]
```

Stop index excluded.

## Pandas .iloc

```
df.iloc[0:2]
```

Stop index excluded.

## Pandas .loc

```
df.loc["row1":"row2"]
```

Stop label included.

# 5. Boolean Indexing Comparison

## NumPy

```
np_arr[np_arr > 25]
```

## Output

```
[30 40 50 60]
```

- Returns 1D array
- Shape lost

# Pandas

```
df[df > 25]
```

## Output

```
      A      B      C  
row1 NaN    NaN  30.0  
row2 40.0  50.0  60.0
```

- Shape preserved
- NaN for false values

# 6. Alignment Behavior (CRITICAL DIFFERENCE)

## NumPy

```
a = np.array([1, 2, 3])  
b = np.array([10, 20, 30])  
  
a + b
```

Position-based addition.

# Pandas

```
s1 = pd.Series([1,2,3], index=["a","b","c"])  
s2 = pd.Series([10,20,30], index=["c","b","a"])  
  
s1 + s2
```

## Explanation

- Values aligned by labels
- Order does not matter

This **saves you or destroys you** depending on understanding.

## 7. Why This Matters in Data Science

### Data Cleaning

- Pandas avoids column mixups
- NumPy is faster but blind

### Feature Engineering

- Pandas keeps feature names
- NumPy loses meaning

### Model Input Preparation

- Pandas for exploration
- NumPy for model input

### ML / DL Pipelines

- Pandas → preprocessing
- NumPy → tensors

#### If you confuse them

- Wrong feature goes to model
- No error raised
- Model accuracy drops silently

## 8. Common Mistakes (VERY IMPORTANT)

1. Using `.loc` like `.iloc`

2. Expecting NumPy to understand column names
3. Forgetting `.loc` includes end label
4. Losing shape with NumPy boolean indexing
5. Mixing Pandas Series with NumPy arrays blindly

## 9. Performance & Best Practices

- NumPy indexing is faster
- Pandas indexing is safer
- Convert to NumPy before training
- Always verify shape
- Print `.head()` and `.shape`

Rule:

- **Explore with Pandas**
- **Train with NumPy**

## 10. When to Use What

Task	Use
Data loading	Pandas
Cleaning	Pandas
Feature naming	Pandas
Numerical ops	NumPy
ML model input	NumPy
Speed-critical loops	NumPy

# 11. Practice Problems (NO SOLUTIONS)

## Easy (5)

1. Select same value using NumPy and Pandas
2. Slice rows using `.iloc` and NumPy
3. Select column using Pandas and NumPy
4. Boolean filter values > mean
5. Compare shapes after filtering

## Medium (7)

6. Show label alignment difference
7. Convert Pandas DataFrame to NumPy safely
8. Filter rows using `.loc` condition
9. Create feature matrix X using `.iloc`
10. Compare boolean masking outputs
11. Slice time-series data
12. Drop target column correctly

## Hard (5)

13. Debug silent misalignment bug
14. Preserve column order during slicing
15. Build train-test split manually
16. Detect shape collapse issue
17. Convert Pandas → NumPy for DL

## Industry-Level Tasks (3)

18. Prevent feature shift bug in production
19. Build preprocessing pipeline using Pandas then NumPy
20. Audit model input correctness

## 12. Mini Checklist

- NumPy is position only
- Pandas has labels
- `.iloc`  $\approx$  NumPy
- `.loc` includes stop
- Pandas aligns by labels
- Convert before training