# Part 4:

## Data Science and Machine Learning With Amin.

## Foundations of Data Analyis with Pandas.



Amin Hydar Ali

# Table of Contents

**The Source Code for this is available in my Github repository:**

**https://github.com/alaminhydar/Data-Science-and-Machine-Learning-With-Amin**

# Before Pandas: Understanding What It Is & Why You Need It

Before installing Pandas and jumping into code, it's important to understand **what it is**, **why it exists**, and **how it makes your life easier when working with data**.

Many beginners install tools without understanding *why* they are using them and that's the fastest way to get confused later. So let's start with clarity.

## 🐼 What is Pandas?

**Pandas** is a Python library designed for **working with data easily, efficiently, and in a structured way**.
If Python is a language, **Pandas is like its "Excel brain",** but far more powerful.

Think of Pandas as a supercharged tool that allows you to:

✅ Read data (CSV, Excel, JSON, SQL, etc.)
✅ Clean messy data
✅ Explore and analyze data
✅ Transform and prepare data for machine learning
✅ Summarize and visualize insights

Pandas turns raw data into **tables** that look and behave like Excel spreadsheets but with the power of programming.

## Why Was Pandas Created? *(The Problem It Solves)*

Before Pandas existed, handling data in Python was **hard and slow**.
People had to write long, complicated code just to do simple things like:

- Load a CSV
- Filter rows
- Remove missing values
- Calculate averages or trends

Data existed everywhere (banks, hospitals, social media, companies), but **there was no easy way to work with it in Python**.

Pandas was created to solve 3 main problems:

| Problem Before Pandas | How Pandas Solved It |
| --- | --- |
| Data was hard to load and clean | Simple functions like **read_csv()** and **.dropna()** |
| Python lacked Excel-like table structures | Introduced **Series** & **DataFrame** objects |
| Data analysis was slow and manual | Vectorized, optimized operations (very fast) |

In short: **Pandas brought simplicity, speed, and structure to data work in Python.**

# 🚀 Why Should You Learn Pandas?

Whether you want to become a data scientist, AI engineer, analyst, or even build ML models, Pandas is unavoidable.

Here's why:

## 1️⃣ It is the Foundation of Data Analysis & Machine Learning

You cannot build models if your data is dirty, incomplete, or unstructured.
**80% of Data Science work is data cleaning,** Pandas is the tool for that.

## 2️⃣ It Replaces Manual Excel Work with Instant Code

Anything you can do in Excel, filtering, merging sheets, pivot tables, Pandas can do **faster**, **more accurately**, and **without errors**.

## 3️⃣ Industry Uses Pandas Everywhere

Used by: Google, Netflix, NASA, Amazon, Meta, Spotify, Banks, Research Labs, Healthcare, Finance, and more.

## 4️⃣ It Connects Directly to AI Frameworks

Tools like TensorFlow and PyTorch expect clean structured data, often prepared with Pandas first.

If AI is the brain, **Pandas prepares the food** so the brain can function.

# When Should You Use Pandas?

Use Pandas when working with any structured data, such as tables or datasets.

### ✅ Use Pandas when:

- You have data in rows and columns (like Excel, CSV, database results)
- You need to clean or fix messy data
- You want to analyze trends or patterns
- You need to prepare data for machine learning
- You want to combine, filter, sort, or transform datasets

### ❌ Pandas is NOT for:

| Task | Better Tool |
|---|---|
| Huge Big Data (GB–TB) | Spark / Dask |
| Deep Learning training | PyTorch / TensorFlow |
| Working with images directly | NumPy, OpenCV, PIL |
| Working with unstructured text | NLTK, spaCy (but data still enters through Pandas) |

# 🔧 Setting Up to Learn Pandas

Before we start coding, you need the right tools. Here's your setup:

## ✅ Install Pandas

Choose one:

**Option 1 - Using pip:**

```
1. pip install pandas
```

**Option 2 - Install Anaconda (includes Pandas + Jupyter Notebook):**
Recommended for beginners because it comes with everything pre-installed.

## 🧰 Tools You Can Use with Pandas

| Tool | Why Use It |
|---|---|
| **Jupyter Notebook** | Best for learning & testing code step-by-step |
| **Google Colab** | Runs on the cloud; great if laptop is low spec |
| **VS Code** | Best for project-style coding and long scripts |

As a beginner, **start with Jupyter Notebook or Google Colab,** they allow small code execution in chunks, which helps you learn gradually.

## 📂 Files You Should Practice With

Start with **small datasets** so you can see changes clearly.

Examples to begin with:

| File Type | Example |
|---|---|
| CSV | sales.csv, students.csv |
| Excel | products.xlsx, employees.xlsx |
| JSON | API response, small structured JSON |
| Sample datasets | Iris, Titanic, FIFA, Netflix shows |

You'll gradually move to larger real-world datasets as your skills grow.

> **Beginners' Tip:** Start with 10–100 rows.
> If your first dataset has 30,000 rows, you'll get overwhelmed and confused.

## 💡 Mini-Starter Exercise (No coding yet)

Before next module, do this:

1. Go to Kaggle or download a **small CSV** (30–100 rows)
2. Open it in Excel or Google Sheets
3. Observe:
    - columns
    - rows
    - missing values
    - strange data formats
4. Ask yourself: *"If I wanted to clean or analyze this, how would I do it manually?"* (Soon, Pandas will do it in seconds.)

This builds intuition before coding.

# Introduction to Pandas & Its Core Objects: Series and DataFrame

## 🎯 Goal of This Module

By the end of this module, you'll:

- Understand what Pandas objects are (Series & DataFrame)
- Know how to create, explore, and inspect them
- Be comfortable working with small real datasets (like CSVs)
- Be able to explain the difference between Series and DataFrame

## Step 1: What Pandas Actually Works With

Pandas is built around **two main building blocks,** think of them like **bricks** used to build every dataset:

| Object | What it represents | Analogy |
|---|---|---|
| **Series** | 1-dimensional labeled array | A single Excel **column** |
| **DataFrame** | 2-dimensional labeled table | A full **Excel sheet** (rows × columns) |

### 1. Series - One Column of Data

A Series is very similar to a NumPy array (in fact it is built on top of the NumPy array object). What differentiates the NumPy array from a Series, is that a Series can have axis labels, meaning it can be indexed by a label, instead of just a number location. It also doesn't need to hold numeric data, it can hold any arbitrary Python Object.

A **Series** is like a single column in Excel.
It has two parts:

- **Values** (the actual data)
- **Index** (labels that identify each row)

Example:

```
1. import pandas as pd
2.
3. s = pd.Series([10, 20, 30, 40], name="Scores")
4. print(s)
```

```
5. Output:

6.  0    10
7.  1    20
8.  2    30
9.  3    40
10. Name: Scores, dtype: int64
```

Let's break that down:

- 0, 1, 2, 3 ⟶ These are **indexes** (row labels)
- 10, 20, 30, 40 ⟶ These are **values**
- Name: Scores ⟶ Optional column name
- dtype ⟶ The **data type** (here: integers)

You can access a single value:

```
1. print(s[2])   # Output: 30
2. You can also label your own index:

3. s = pd.Series([10, 20, 30], index=['Alice', 'Bob', 'Carol'])
4. print(s)
5. Output:

6. Alice    10
7. Bob      20
8. Carol    30
9. dtype: int64
```

You can convert a list, numpy array, or dictionary to a Series:

```
1. labels = ["a", "b", "c"]
2. my_list = [10, 20, 30]
3. arr = np.array([10, 20, 30])
4. d = {"a":10, "b":20, "c":30}
5.
6. pd.Series(data=my_list, index=labels)
7. pd.Series(my_list, labels)
8. pd.Series(d)
```

A pandas Series can hold variety of object types, Even functions (although unlikely you will use this)

```
pd.Series([sum, print, len])

0 <built-in function sum>
1 <built-in function print>
2 <built-in function len>
dtype: object
```

### Using an Index

The key to using a Series is understanding its index. Pandas makes use of these index names or numbers by allowing for fast look ups of information (works like a hash table or dictionary).

Let's see some examples of how to grab information from a Series. Let us create two series, ser1 and ser2:

```
ser1 = pd.Series([1,2,3,4],index = ['USA', 'Germany','USSR', 'Japan'])
Output:
 USA 1
 Germany 2
 USSR 3
 Japan 4
 dtype: int64

ser2 = pd.Series([1,2,5,4],index = ['USA', 'Germany','Italy', 'Japan'])
Output:
 USA 1
 Germany 2
 Italy 5
 Japan 4
 dtype: int64

ser1['USA']: 1
```

Operations are then also done based off index:

```
ser1 + ser2

Output:
 Germany 4.0
 Italy NaN
 Japan 8.0
```

```
USA 2.0
USSR NaN
dtype: float64
```

**Think of Series as:**

> "A labeled list of data, like one column with row names."

## 2. DataFrame - A Full Table (Rows × Columns)

A **DataFrame** is a collection of multiple Series combined side by side.
Each column in a DataFrame is actually a Series.
DataFrames are the workhorse of pandas and are directly inspired by the R programming language.
We can think of a DataFrame as a bunch of Series objects put together to share the same index.

**Example**:

```
1. data = {
2.     'Name': ['Alice','Bob','Charlie'],
3.     'Age': [24, 27, 22],
4.     'Score': [85, 90, 88]
5. }
6.
7. df = pd.DataFrame(data)
8. print(df)
9. Output:

10.       Name  Age  Score
11. 0    Alice   24     85
12. 1      Bob   27     90
13. 2  Charlie   22     88
```

🧩 Here's what's happening:

- **Columns:** Name, Age, Score
- **Rows:** 0, 1, 2 (these are indexes)
- Each column (like df["Age"]) is actually a Series.

✅ **Think of DataFrame as:**

> "A collection of labeled columns (Series) that share the same row indexes."

## 🔍 Step 2: Exploring DataFrames

Now let's look at some **methods and attributes** that help us *inspect and understand data.*

Here's a sample DataFrame:

```
import pandas as pd

data = {
    'Name': ['Alice','Bob','Charlie','David'],
    'Age': [24, 27, 22, 32],
    'Score': [85, 90, 88, 95]
}
df = pd.DataFrame(data)
```

Let's explore:

| Command | What It Does | Example Output |
|---|---|---|
| **df.shape** | Shows the number of rows and columns | (4, 3) |
| **df.dtypes** | Shows data types of each column | object, int64, int64 |
| **df.columns** | Lists column names | ['Name','Age','Score'] |
| **df.index** | Lists row labels | RangeIndex(start=0, stop=4, step=1) |
| **df.head()** | Displays first 5 rows | Preview of top rows |
| **df.tail()** | Displays last 5 rows | Preview of bottom rows |
| **df.info()** | Summary of data, types, and memory | "non-null counts, dtypes" |
| **df.describe()** | Statistics summary for numeric columns | mean, std, min, max, etc. |

Try these:

```
print(df.shape)
print(df.dtypes)
print(df.columns)
print(df.index)
print(df.head())
print(df.info())
print(df.describe())
```

Each of these helps you quickly **understand the dataset** without manually checking.

## 📂 Step 3: Practice Loading Real Data (CSV)

Let's use a real file (CSV = Comma Separated Values).
Example **students.csv:**

```
1. Name,Age,Score
2. Alice,24,85
3. Bob,27,90
4. Charlie,22,88
5. David,32,95
6. Code:

7. df = pd.read_csv('students.csv')
8. print(df.head())
9. Output:

10.       Name  Age  Score
11. 0    Alice   24     85
12. 1      Bob   27     90
13. 2  Charlie   22     88
14. 3    David   32     95
```

✅ You just read a CSV file into a DataFrame!
Now you can analyze, sort, or clean it easily.

## 🔄 Step 4: Converting Between Python and Pandas

Pandas makes it easy to convert between **Python's native structures** (like lists or dictionaries) and **DataFrames**.

**Example 1**: Dict ⟶ DataFrame:

```
1. data = {'Name': ['A', 'B'], 'Age': [10, 20]}
2. df = pd.DataFrame(data)
3. print(df)
```

**Example 2:** DataFrame ⟶ Dict:

```
1. back_to_dict = df.to_dict()
2. print(back_to_dict)
```

**Example 3:** Save DataFrame ⟶ CSV:

```
1. df.to_csv('output.csv', index=False)
```

✅ This saves your data to a CSV file.

## Mini-Check: Key Concepts

Try answering these:

1️⃣ What is a **Series** in Pandas?
2️⃣ What is a **DataFrame**?
3️⃣ How are they related?
4️⃣ What does df.shape show?
5️⃣ How do you load a CSV file into Pandas?
6️⃣ How would you explain "index" in one sentence?

## Quick Review Summary

| Concept | Explanation |
|---|---|
| **Series** | 1D labeled array: one column of data |
| **DataFrame** | 2D labeled table: multiple Series combined |
| **Index** | Row labels that help identify data |
| **Common Methods** | **.shape(), .head(), .info(), .describe()** |
| **Conversion** | Easily move between Python dicts, lists, and CSVs |

## ✅ Mini Project Idea

Create a simple student score sheet:

1. Make a dictionary with student names, ages, and marks.
2. Convert it to a DataFrame.
3. Save it as students.csv.
4. Load it back and display the first 3 rows.

# Reading & Writing Data (Input / Output in Pandas)

## 🎯 Goal

By the end of this module, you'll:
✅ Understand **how data is stored and shared** in the real world.
✅ Learn **how to load** (read) and **save** (write) data in different formats.
✅ Know **common problems** (encoding, date parsing) and how to fix them.
✅ Work confidently with small real datasets.

## 🧠 Step 1: How Data Is Stored

Before we touch code, let's understand the *formats* we meet in real life:

| Format | What it is | Looks like | Used for |
|---|---|---|---|
| **CSV (Comma-Separated Values)** | Text file where each line = a row, separated by commas | Name,Age,Score | Simple datasets, spreadsheets |
| **TSV (Tab-Separated Values)** | Like CSV but uses a tab instead of comma | Alice\t24\t85 | Data with commas inside text |
| **Excel (.xlsx)** | Spreadsheet file (can have multiple sheets) | Like your normal Excel | Office / Business reports |
| **JSON (JavaScript Object Notation)** | Text format for structured data | {"name": "Alice", "age": 24 } | Web APIs, config files |
| **SQL Databases** | Structured data stored in tables | Rows and columns | Business databases, apps |
| **Parquet** | Compressed binary format | PAR1 Ó��^����🔲 | Big data / Machine learning pipelines |

**Parquet** is just a file format like CSV or Excel but smaller and faster. It is a compressed, column-based file format used for storing tables of data.

**Why People Use it**

- Much smaller than CSV (because it compresses data efficiently)
- Much faster to read (because it stores data by column, not by row)
- Keeps data types (while CSV loses them)
- Designed for large datasets

It will **not** look like rows and columns, and it will **not** look like CSV or JSON.

That's because Parquet is:

- *compressed*
- *column-oriented*
- *binary encoded*

It is designed to be read by **computers**, not people.

In simple terms:

> Pandas can "talk to" all these file types, open them, edit them, and save them back.

## Step 2: Reading Data (Loading Files)

Let's start small: **CSV** first.

### 1️⃣ Reading CSV Files

Imagine a file called **students.csv:**

```
Name,Age,Score
Alice,24,85
Bob,27,90
Charlie,22,88
David,32,95
Code:

import pandas as pd

df = pd.read_csv('students.csv')
print(df)
Output:

      Name  Age  Score
0    Alice   24     85
1      Bob   27     90
2  Charlie   22     88
```

```
16. 3    David   32     95
```

✅ **pd.read_csv()** turns a CSV file into a **DataFrame**.
Now you can easily inspect or modify it.

### ✏️ Optional arguments for read_csv

```
1. pd.read_csv('students.csv', delimiter=',')   # change separator
2. pd.read_csv('students.csv', nrows=2)         # only read 2 rows
3. pd.read_csv('students.csv', usecols=['Name','Score'])  # select specific columns
4. pd.read_csv('students.csv', encoding='utf-8') # fix character encoding
5. pd.read_csv('students.csv', parse_dates=['Date']) # if there's a date column
```

### 2️⃣ Writing CSV Files

Once you process or filter your data, you can save it again.

**Example**:

```
1. filtered = df[df['Score'] > 85]
2. filtered.to_csv('top_students.csv', index=False)
```

✅ This saves only students with Score > 85.
The argument **index=False** means "don't save row numbers."

## Step 3: Reading & Writing Excel Files

### Read Excel

Let's say you have **students.xlsx.**

```
1. df = pd.read_excel('students.xlsx', sheet_name='Sheet1')
2. print(df.head())
```

### 📗 Write Excel

```
df.to_excel('output.xlsx', index=False)
```

✅ Works exactly like CSV but supports multiple sheets.

## Step 4: Reading & Writing JSON Files

JSON is **text data stored as key-value pairs**, like this:

**students.json:**

```
[
  {"Name": "Alice", "Age": 24, "Score": 85},
  {"Name": "Bob", "Age": 27, "Score": 90}
]
```

Read it:

```
df = pd.read_json('students.json')
print(df)
```

Write it back:

```
df.to_json('output.json', orient='records', indent=2)
```

✅ **orient='records'** ⟶ Each row becomes a JSON object.
✅ **indent=2** ⟶ Makes JSON more readable.

### Nested JSONs: Flattening

Sometimes a JSON file is nested (like dictionaries inside dictionaries).

Example:

```
{
  "student": {
    "name": "Alice",
    "scores": {"math": 90, "english": 85}
  }
```

```
}
```

To flatten:

```
import pandas as pd
from pandas import json_normalize

data = {
  "student": {
    "name": "Alice",
    "scores": {"math": 90, "english": 85}
  }
}

flat = json_normalize(data)
print(flat)
Output:
  student.name  student.scores.math  student.scores.english
0        Alice                   90                      85
```

✅ That's how Pandas "flattens" nested data into columns.

## Step 5: SQL and Parquet ( Optional)

If you ever work with databases or ML pipelines, these are useful.

### SQL Example:

```
import sqlite3
```

```
2.
3. conn = sqlite3.connect('students.db')
4. df = pd.read_sql('SELECT * FROM scores', conn)
```

### Parquet Example (fast and compressed):

```
1. df.to_parquet('data.parquet')
2. df = pd.read_parquet('data.parquet')
```

✅ Parquet is mostly used in **big data and AI projects** (very fast and memory efficient).

## Step 6: Common Pitfalls (Beginner Mistakes)

| Problem | Why it happens | Fix |
|---|---|---|
| **Encoding error** | Some files use different character sets (e.g., "latin1") | **pd.read_csv('file.csv', encoding='latin1')** |
| **Dates appear as strings** | Pandas doesn't auto-parse them | **parse_dates=['Date']** |
| **Large file load too slow** | File is huge | **chunksize=1000 (load piece by piece)** |
| **Wrong separator** | Not comma-separated | **delimiter='\t' for TSV** |

## Step 7: Practice Exercise

### Task 1

1. Download a small CSV (e.g., from Kaggle or your own sample).
2. Load it using **pd.read_csv()**.
3. Display its shape and first 3 rows.
4. Filter rows (e.g., **df[df['Age'] > 25]**).
5. Save filtered result as **output.csv**.

### Task 2

1. Create a small JSON file manually.
2. Read it into Pandas.
3. Write it back as CSV.

## Step 8: Mini Review

| Concept | You Should Know |
|---|---|
| **pd.read_csv()** | Load comma-separated text into DataFrame |
| **df.to_csv()** | Save DataFrame as CSV file |
| **pd.read_excel() / df.to_excel()** | Work with Excel files |
| **pd.read_json() / df.to_json()** | Handle JSON data |
| **pd.read_sql() / df.to_parquet()** | Advanced formats |
| **Common Pitfalls** | Encoding, date parsing, wrong separators |

## 💡 Real-World Context

Pandas I/O is used everywhere:

- 🏢 Companies import Excel/CSV reports daily
- 💻 AI engineers read JSON datasets
- 🧪 Scientists load large CSVs or SQL tables
- 🧠 Machine Learning models load training data via Pandas

So even if you're new, this module gives you **real practical skills**.

## ✅ Mini Project Idea

Create a **Data Import/Export Tool**:

1. Ask user to choose file type (CSV / Excel / JSON).
2. Read the file.
3. Filter rows (e.g., top scores).
4. Save results in another format.
   👉 You'll understand the full I/O cycle!

# Indexing, Selection & Boolean Masking

## 🎯 Goal

By the end of this module, you'll:
✅ Know how to **pick rows and columns** from a DataFrame.
✅ Understand **label-based vs positional indexing**.
✅ Learn **boolean masks** to filter data using conditions.
✅ Use handy functions like **.isin()**, **.between()**, **.query().**

This is **core power** of Pandas once you master it, you can extract exactly the data you want.

## Step 1: Label-based vs Positional Indexing

Imagine a small dataset:

```
1. import pandas as pd
2.
3. data = {
4.     'Name': ['Alice', 'Bob', 'Charlie', 'David'],
5.     'Age': [24, 27, 22, 32],
6.     'Score': [85, 90, 88, 95]
7. }
8.
9. df = pd.DataFrame(data, index=['a','b','c','d'])
10. print(df)
11. Output:
12.       Name  Age  Score
13. a    Alice   24     85
14. b      Bob   27     90
15. c  Charlie   22     88
16. d    David   32     95
```

- Rows are **labeled** a, b, c, d.
- Columns are labeled 'Name', 'Age', 'Score'.

## 1. Label-based Indexing: .loc

- **Pick a single row**:

```
print(df.loc['b'])
Output:
Name     Bob
Age       27
Score     90
Name: b, dtype: object
```

- **Pick multiple rows**:

```
print(df.loc[['a','c']])
Output:
      Name  Age  Score
a    Alice   24     85
c  Charlie   22     88
```

- **Pick specific row + column**:

```
print(df.loc['b', 'Score'])  # Row b, column Score
Output:
90
```

- **Pick multiple rows + columns**:

```
print(df.loc[['a','d'], ['Name','Score']])
Output:
     Name  Score
a  Alice     85
d  David     95
```

## 2. Positional Indexing: .iloc (index-location)

- **Pick by row number / column number** (0-based index):

```
print(df.iloc[1])       # 2nd row
print(df.iloc[0,2])     # Row 0, Column 2
print(df.iloc[[0,3],[0,2]])  # Rows 0,3; Columns 0,2
```

Rule of thumb:

- **.loc** → uses labels, human-friendly labels,
- **.iloc** → uses numbers, programmer-friendly indices.

`iloc` = **index-location**
It selects **rows and columns by number**, *not by name.*

Think of your DataFrame as a big table where everything is numbered starting at **0**.

```
1. Row numbers → 0, 1, 2, 3, ...
2. Column numbers → 0, 1, 2, 3, ...
```

**`iloc`** lets you pick things using these numbers.

### Basic Pattern

**df.iloc[rows, columns]**

### Where:

- **rows** = which row numbers you want
- **columns** = which column numbers you want

### Step 2: Column Selection

Single column:

```
1. print(df['Name'])
```

Multiple columns:

```
1. print(df[['Name','Score']])
```

## Step 3: Boolean Masking

Boolean masks allow **filtering rows based on conditions**.

**Example**: Pick all students older than 25:

```
1. mask = df['Age'] > 25
2. print(mask)
3. Output:
4. a    False
5. b     True
6. c    False
7. d     True
8. Name: Age, dtype: bool
```

Apply mask:

```
1. print(df[mask])
2. Output:
3.   Name  Age  Score
4. b  Bob   27     90
5. d David  32     95
```

Shorter version:

```
1. print(df[df['Age'] > 25])
```

### Boolean Operators

| Symbol | Meaning | Example |
|---|---|---|
| & | AND | (df['Age']>25) & (df['Score']>90) |
| \| | OR | df[(df)] |
| ~ | NOT | ~(df['Age']>25) |

```
1. print(df[(df['Age']>25) & (df['Score']>90)])
```

## Handy Functions

**.isin()** → check membership in a list:

```
1. print(df[df['Name'].isin(['Alice','David'])])
```

**.between()** → check if value falls in a range:

```
1. print(df[df['Age'].between(24, 30)])
```

**.query()** → filter using string expressions:

```
1. print(df.query('Age > 25 & Score > 90'))
```

## Step 4: Row / Column Slices

Rows 1–3:

```
1. print(df.iloc[1:4])
```

Columns 0–1:

```
1. print(df.iloc[:,0:2])
```

### 1. Select rows (only)

**df.iloc[0]**

Get first 3 rows:

**df.iloc[0:3]**

Get rows 1, 4, and 5:

**df.iloc[[1, 4, 5]]**

### 2. Select columns (only)

First column:

**df.iloc[:, 0]**

`:` means "all rows".

Columns 0–2:

**df.iloc[:, 0:3]**

### 3. Select rows AND columns

Row 0, Column 1**:**

**df.iloc[0, 1]**

Rows 0–2, Columns 0–1**:**

**df.iloc[0:3, 0:2]**

Rows 1 and 4, Columns 2 and 3:

**df.iloc[[1, 4], [2, 3]]**

### 4. Select entire blocks (rectangle of data)

Rows 0–4 and Columns 1–3:

**df.iloc[0:5, 1:4]**

This is like slicing a rectangular window out of your table.

### Important: iloc ignores names

Even if your DataFrame columns are named `"age"` or `"city"`, `iloc` doesn't care.

It only uses numbers.

### So if your columns are:

| Column Name | Column Number |
|---|---|
| "name" | 0 |
| "age" | 1 |
| "city" | 2 |

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

Means "give me column 1," which is `"age"`, even though we never wrote `"age"`.

### Mix slicing + masking:

```
1. print(df.loc[df['Score']>85, ['Name','Score']])
```

## Step 5: Mini-Tips

Always test your mask:

```
1. mask = df['Age'] > 25
2. print(mask.sum())          # how many True?
3. print(mask.value_counts()) # True/False counts
```

- This avoids accidentally selecting **empty DataFrames**.
- When using multiple conditions, **wrap each in parentheses**:

```
1. (df['Age']>25) & (df['Score']>90)
```

## Step 6: Practice Exercises

1. Load a CSV with Name, Age, Score.
2. Select **all students with Score > 85**.
3. Pick **only Name and Score** columns.
4. Use .isin() to select specific students.

5. Filter students **Age between 24 and 30**.
6. Try .query() to get students **Score > 90**.

## Step 7: Key Concepts Recap

| Concept | Key Point |
|---|---|
| **.loc** | Label-based selection |
| **.iloc** | Position-based selection |
| **df['col']** | Single column |
| **df[['c1','c2']]** | Multiple columns |
| **Boolean mask** | Filter rows with True/False conditions |
| **.isin()** | Membership check |
| **.between()** | Range check |
| **.query()** | String-based filtering |
| **Testing mask** | Avoid empty DataFrames |

Think of **Boolean masks** as "filters" like a sieve: only rows passing the condition stay.

✅ **Mini Project Idea:**

- Take a small dataset (CSV or Excel).
- Pick only rows where Age > 25 **and** Score > 90.
- Select only Name and Score columns.
- Save it to a new CSV called filtered_students.csv

# Cleaning Data: Missing Values & Duplicates

## 🎯 Goal

Real-world data is messy. You almost never get perfect datasets. The goal here is to learn how to **detect, handle, and clean missing values and duplicates** so your data is ready for analysis or AI.

Think of it like **preparing ingredients for cooking**: if you have rotten vegetables or missing spices, the recipe won't turn out well. Cleaning data is making sure your "ingredients" are good.

## Step 1: Missing Values

- In Pandas, missing values are **NaN** (`Not a Number`) or **None**.
- They appear when a dataset has gaps, like surveys where some people skip questions.

### Detect Missing Values

```
1. df.isna()      # True where value is missing
2. df.isnull()    # Equivalent to isna()
```

This produces a **Boolean DataFrame** showing where data is missing.

### Count Missing Values

```
1. df.isna().sum()
```

Gives you **how many missing values per column**.

# Handle Missing Values

### 1. Drop missing values

**`dropna`** = remove rows or columns that contain missing values (NaN).
This removes any **row** that contains **at least one** NaN.

### Drop rows with any NaN

```
1.   age  city
2. 0  25  Paris
3. 1  NaN  London
```

```
4. 2   30    NaN
```

```
1. df.dropna()
# Output
  age   city
  0   25   Paris
```

Rows 1 and 2 are removed because they had missing values.

## Drop rows only if *all* columns are NaN

```
1. df.dropna(how="all")
```

Example:

```
1. age   city
2. 25    Paris
3. NaN   NaN
4. 30    London
```

Row with both NaN values is removed.

## Drop columns instead of rows

Use **`axis=1`**:

```
1. df.dropna(axis=1)
```

This removes any column that contains at least one NaN.

Example:

```
1.    age   city   score
2. 0   25   Paris  10
3. 1  NaN   London NaN
4. 2   30    NaN   18
```

If you run:

```
1. df.dropna(axis=1)
```

Only columns with no NaN remain (maybe none).

### Drop rows that have NaN in specific columns

**Example:** Drop rows where `"age"` is NaN:

```
df.dropna(subset=["age"])
```

This removes only rows with missing age, not missing city.

### Require a minimum number of non-NaN values

**Example:** keep rows with at least 2 valid values:

```
df.dropna(thresh=2)
```

This is good when you have wide tables.

### In-place vs returning a copy

Like most Pandas operations:

- Without **`inplace=True`** = returns a new DataFrame
- With **`inplace=True`** = modifies the DataFrame directly

Example:

```
df.dropna(inplace=True)
```

**Analogy**: Throw away any rotten ingredient. But **don't throw away too much** you might lose valuable data.

## 2. Fill missing values (`fillna`)

It's used  when your DataFrame has blanks, NaN, or missing data.

Example of missing values:

```
   age   city
0   25   Paris
1  NaN   London
2   30    NaN
```

**`fillna`** lets you replace those **`NaN`** values with something else.

### Basic idea

```
1. df.fillna(value)
```

Replace all missing values with the given **value**.

## Replace all NaN with one value

Example:

```
1. df.fillna(0)
```

All missing values become `0`.

If the DataFrame was:

```
1. age   city
2. 25    Paris
3. NaN   London
4. 30    NaN
```

### After:

```
1. age   city
2. 25    Paris
3. 0     London
4. 30    0
```

## Replace NaN in a specific column

```
1. df["age"].fillna(0, inplace=True)
```

### Or without modifying in-place:

```
1. df["age"] = df["age"].fillna(0)
```

### Fill different values for different columns

Use a dictionary:

```
df.fillna({
    "age": 0,
    "city":"Unknown"
})
```

### Forward fill (copy previous value downward)

This fills NaN with the value *above* it.

```
df.fillna(method="ffill")
```

Example:

```
age
10
NaN
NaN
20
```

Becomes:

```
10
10
10
20
```

### Backward fill (copy next value upward)

```
df.fillna(method="bfill")
```

```
age
10
NaN
20
```

Becomes:

```
1. 10
2. 20
3. 20
```

### Fill only a limited number of NaN values

```
1. df.fillna(0, limit=1)
```

This only replaces the first NaN per column.

### Key notes

- **`fillna`** returns a new DataFrame unless you use **`inplace=True`**
- It works with:
    - DataFrames
    - Series
    - numeric columns
    - text columns

## Step 2: Duplicates

Sometimes datasets have **exactly repeated rows**.

### Find duplicates

```
1. df.duplicated()
```

This returns `True` for each row that is a duplicate of a previous row.

Example:

| name | age |
|---|---|
| Alice | 25 |
| Bob | 30 |
| Alice | 25 |

```
1. df.duplicated() → [False, False, True]
```

### Remove duplicate rows

```
1. df.drop_duplicates()
```

Removes any repeated rows.

**Result:**

| name | age |
|---|---|
| Alice | 25 |
| Bob | 30 |

### 📌 Optional settings

**Keep the *first* occurrence (default)**

```
1. df.drop_duplicates(keep="first")
```

**Keep the *last* occurrence**

```
1. df.drop_duplicates(keep="last")
```

**Drop all duplicates (remove every repeated row)**

```
1. df.drop_duplicates(keep=False)
```

**Drop duplicates only based on specific column(s)**
Example: Remove people with the same name:

```
1. df.drop_duplicates(subset=["name"])
```

**If you want to check multiple columns:**

```
1. df.drop_duplicates(subset=["name", "age"])
```

**Analogy**: Imagine a survey filled twice by the same person, you only need **one copy**.

## Step 3: Correct Data Types

Pandas sometimes guesses wrong types.

Real-world datasets often load incorrectly:

- numbers become strings
- dates become strings

- booleans become text
- integers become floats with `.0`

You fix them using **`.astype()`** or specialized converters.

### Convert a column to integer

```
df["age"] = df["age"].astype(int)
```

## Convert to float

```
df["price"] = df["price"].astype(float)
```

## Convert to string

```
df["id"] = df["id"].astype(str)
```

## Convert to boolean

```
df["is_active"] = df["is_active"].astype(bool)
```

## Convert to datetime (VERY common)

Use **`to_datetime`**:

```
df["date"] = pd.to_datetime(df["date"])
```

Works for many formats:

- `"2023-01-01"`
- `"1/1/2023"`
- `"Jan 1, 2023"`

## 🔍 Finding wrong data types

```
df.dtypes
```

Example output:

```
name        object
age         object
salary      float64
hire_date   object
```

This tells you what needs fixing.

### Convert multiple columns at once

```
df = df.astype({
    "age": int,
    "salary": float,
})
```

## 🌟 Summary Table

| Task | Command |
|---|---|
| Find duplicates | `df.duplicated()` |
| Remove duplicates | `df.drop_duplicates()` |
| Remove duplicates by column | `df.drop_duplicates(subset=["col"])` |
| Convert type | `df["col"] = df["col"].astype(type)` |
| Convert to datetime | `pd.to_datetime(df["col"])` |
| See types | `df.dtypes` |

This ensures your operations work correctly, like making sure an oven is the right temperature before baking.

✅ **Mini-Practice:**

1. Find missing values in a CSV.
2. Fill numerical columns with mean, categorical with mode.
3. Drop duplicate rows.
4. Convert a “Date” column to datetime.

# Transforming Columns & Strings

## 🎯 Goal

Make your data **usable, consistent, and enriched**. Transform columns, combine data, manipulate strings. Think of it like **prepping vegetables, chopping and mixing**.

## Step 1: Column Arithmetic

You can **create new columns** from existing ones:

```
df['Total'] = df['Math'] + df['English']
df['BMI'] = df['Weight'] / (df['Height']/100)**2
```

**Analogy**: You're **mixing ingredients** to create a new dish.

**`df.assign()`** is method-chaining friendly:

```
df = df.assign(Total = df['Math'] + df['English'])
```

## Step 2: String Operations

String data often needs cleaning (e.g., emails, names):

```
df['Email'].str.lower()          # Convert to lowercase
df['Email'].str.strip()          # Remove leading/trailing spaces
df['Email'].str.contains('gmail')  # Check if 'gmail' in email
df['Name'].str.split(' ')        # Split first and last names
df['Name'].str.replace(' ', '_') # Replace spaces with underscore
```

**Analogy**: Like washing and slicing vegetables so they're ready to cook.

# Step 3: Safe Type Conversion

Safe type conversion means: Convert a column to another data type without crashing, and without losing important data.

Real-world data is messy, so unsafe conversions will fail.

Example of unsafe conversion:

```
1. df['age'] = df['age'].astype(int)
```

This will error if values like " " or "unknown" exist.

Safe conversion avoids that.

## Safe numeric conversion

Use **`pd.to_numeric(..., errors="coerce")`**

```
1. df["age"] = pd.to_numeric(df["age"], errors="coerce")
```

## What this does:

- Valid numbers → become numbers
- Invalid values → become `NaN` (instead of an error)

Example:

| age (raw) | after safe conversion |
|---|---|
| "25" | 25 |
| " 40 " | 40 |
| "unknown" | NaN |
| "" | NaN |

This is **safe** because it never crashes

## When to use it

- CSVs with mixed types
- Numeric columns with text inside
- Data scraped from the web
- Cleaning user-entered data

## Safe datetime conversion

Use **`pd.to_datetime(..., errors="coerce")`**

```
1. df["date"] = pd.to_datetime(df["date"], errors="coerce")
```

## What happens?

- Valid dates ⟶ parsed
- Invalid dates ⟶ NaT (Not a Time)

Example:

| date (raw) | after conversion |
|---|---|
| "2023-01-01" | 2023-01-01 |
| "Jan 5" | 2023-01-05 |
| "not a date" | NaT |

## Why this matters

If you use the unsafe version:

```
1. df["date"] = df["date"].astype("datetime64[ns]")
```

Pandas will throw error instantly.

## Safe conversion to boolean

| col |
|---|
| "True" |
| "false" |
| "yes" |
| "0" |
| "" |

Safe method:

```
1. df["flag"] = df["flag"].map({"yes": True, "True": True, "1": True, "no": False, "False": False, "0": False})
```

Or convert numeric → boolean:

```
1. df["flag"] = pd.to_numeric(df["flag"], errors="coerce").fillna(0).astype(bool)
```

### Safe conversion from float → int:

If you do this:

```
df["age"] = df["age"].astype(int)
```

It will break if you have NaN.

Safe method:

```
df["age"] = pd.to_numeric(df["age"], errors="coerce")  # convert safely
df["age"] = df["age"].fillna(0).astype(int)            # now convert
```

Or keep NaNs using Pandas' special nullable integer type:

```
df["age"] = pd.to_numeric(df["age"], errors="coerce").astype("Int64")
```

This allows integers **with NaN**, which normal int cannot do.

## 🌟 Summary: Safe Conversion

| Goal | Safe Code |
|---|---|
| Convert to number | `pd.to_numeric(df[col], errors="coerce")` |
| Convert to datetime | `pd.to_datetime(df[col], errors="coerce")` |
| Convert float → int with NaN allowed | `.astype("Int64")` |
| Convert booleans safely | Map text values → True/False |
| Convert multiple columns safely | Use dict with coercion |

### ✅ Mini-Practice:

1. Extract **domain** from email column.
2. Create BMI column.
3. Standardize names to lowercase.
4. Convert numeric strings to numbers safely.

# 📊 Grouping & Aggregation (SQL "GROUP BY")

## 🎯 Goal

Summarize data by groups. Example: Average sales by region, total users per category. Think **like a chef grouping ingredients** to make different dishes.

## Step 1: Grouping

Think of **groupby** as:

> **"Split the data into groups based on a column, then do something to each group."**

### Simple everyday example

Imagine this table:

| city | sales |
|---|---|
| Paris | 10 |
| Paris | 20 |
| London | 15 |
| London | 25 |

If you want total sales per city, you group by `city` and sum `sales`.

### Basic structure

```
1. df.groupby("column_to_group").agg(function)
```

Or even simpler:

```
1. df.groupby("column").function()
```

### Most common grouping: sum by group

```
1. df.groupby("city")["sales"].sum()
```

Result:

| city | sales |
|---|---|
| London | 40 |
| Paris | 30 |

### Count how many rows per group

```
1. df.groupby("city")["sales"].count()
```

If you don't specify a column:

```
1. df.groupby("city").size()
```

```
1. grouped = df.groupby('Region')
```

`grouped` is a **GroupBy object,** like a folder for each region.

### Mean (average) per group

```
1. df.groupby("city")["sales"].mean()
```

## Step 2: Aggregations

Aggregation = **calculating something** for each group (sum, mean, count, etc.).

Compute **mean, sum, count, etc.**:

### More useful aggregates

| Function | Meaning |
|---|---|
| `sum()` | total |
| `mean()` | average |
| `count()` | number of rows |
| `max()` | biggest value |
| `min()` | smallest value |
| `median()` | middle value |

Example:

```
1. df.groupby("city")["sales"].max()
```

### Multiple aggregations at once

```
1. df.groupby("city")["sales"].agg(["sum", "mean", "count"])
```

### Break it down step by step

```
1. df.groupby("city")
```

This splits the DataFrame into groups based on the values in the `city` column.

**If your data looks like this:**

| city | sales |
|---|---|
| Paris | 10 |
| Paris | 20 |
| London | 15 |
| London | 25 |

Then groupby creates 2 groups:

- A *Paris* group → rows 1 and 2
- A *London* group → rows 3 and 4

`["sales"]`

This selects only the **`sales column`** from each group.

So now you have:

- Paris → `[10, 20]`
- London → `[15, 25]`

**`agg`** means `apply multiple aggregations`.

It asks Pandas to compute:

- `"sum"` → total sales
- `"mean"` → average sales
- `"count"` → number of sales entries

**for each city.**

📊 **Final result:**

| city | sum | mean | count |
|---|---|---|---|
| London | 40 | 20 | 2 |
| Paris | 30 | 15 | 2 |

### Aggregating multiple columns

```
df.groupby("city").agg({
    "sales":"sum",
    "customers":"mean"
})
```

Each column can have its own aggregation.

### Grouping by multiple columns

Example table:

| city | product | sales |
|---|---|---|
| Paris | A | 10 |
| Paris | A | 20 |
| Paris | B | 5 |
| London | A | 15 |

**Group by city AND product:**

```
df.groupby(["city", "product"])["sales"].sum()
```

Result:

| city | product | sales |
|---|---|---|
| London | A | 15 |
| Paris | A | 30 |
| Paris | B | 5 |

### Get rid of hierarchical index (optional but common)

```
df.groupby("city")["sales"].sum().reset_index()
```

This turns the result back into a normal DataFrame.

### When to use grouping & aggregation

- Total revenue per customer
- Average score per student
- Count of orders per day
- Maximum temperature per city
- Sum of sales per product category

## Step 3: Transform vs Filter

**`transform`** returns a result the *same size* as the original DataFrame.
It calculates something for each group, then broadcasts it back to all rows.

This is different from `agg`, which *shrinks* the data.

**Think of it like this:**

> "Give every row a value that belongs to its group."

Example DataFrame:

| city | sales |
|---|---|
| Paris | 10 |
| Paris | 20 |
| London | 15 |
| London | 25 |

**Example:** Create a column of average sales per city

```
1. df["city_avg"] = df.groupby("city")["sales"].transform("mean")
```

Result:

| city | sales | city_avg |
|---|---|---|
| Paris | 10 | 15 |
| Paris | 20 | 15 |
| London | 15 | 20 |
| London | 25 | 20 |

The DataFrame is the same length, but each group now has its average repeated.

**Another example:** get sales as a percentage of city total

```
df["pct_of_city_total"] = (
    df["sales"] / df.groupby("city")["sales"].transform("sum")
)
```

### Filter

**`filter`** removes entire groups based on a condition.
If the condition is **False,** the WHOLE group is removed.

**Example:** keep only cities where total sales > 30

```
df.groupby("city").filter(lambda g: g["sales"].sum() > 30)
```

**Paris total** = 30 ⟶ removed
**London total** = 40 ⟶ kept

Result:

| city | sales |
|---|---|
| London | 15 |
| London | 25 |

**Another example:** only keep cities with at least 2 rows

```
df.groupby("city").filter(lambda g: len(g) >= 2)
```

This removes groups with only 1 entry.

### Summary Table

| Feature | `agg` | `transform` | `filter` |
|---|---|---|---|
| Output size | smaller | same size | smaller (groups removed) |
| Works per group | yes | yes | yes |
| Returns | one value per group | one value per row | whole groups |
| Typical use | summarizing | adding new columns | keeping/removing groups |

**Way to remember**

- **`agg`** → summaries
- **`transform`** → add info back to each row
- **`filter`** → remove whole groups

## Step 4: Flatten grouped DataFrame

When you group a DataFrame and aggregate it, Pandas often produces a **MultiIndex** in either:

- the **row index**,
- or the **column index**,
- or both.

"Flattening" simply means: **remove the MultiIndex and turn it back into normal columns.**

Here are the clearest and most common ways to flatten a grouped DataFrame.

1. **Use `.reset_index()`**

This flattens **row index** created by groupby.

Example:

```
1. df2 = df.groupby("city")["sales"].sum()
```

This gives:

```
1. city
2. London    40
3. Paris     30
4. Name: sales, dtype: int64
```

Flatten it:

```
1. df2 = df2.reset_index()
```

Result:

| city | sales |
|---|---|
| London | 40 |
| Paris | 30 |

2. **Flatten MultiIndex columns after `.agg()`**

This happens when you use multiple aggregations:

```
df2 = df.groupby("city")["sales"].agg(["sum","mean","count"])
```

You get MultiIndex columns:

```
        sum  mean  count
city
London   40    20      2
Paris    30    15      2
```

To flatten columns:

**Method A - join column levels**

```
df2.columns = ["_".join(col) if isinstance(col, tuple) else col for col in df2.columns]
df2.reset_index(inplace=True)
```

Result:

| city | sales_sum | sales_mean | sales_count |
|---|---|---|---|
| London | 40 | 20 | 2 |
| Paris | 30 | 15 | 2 |

**Method B - use `map` (cleaner)**

```
df2.columns = df2.columns.map("_".join)
df2 = df2.reset_index()
```

**Method C - name aggregations manually (best practice)**

```
df2 = df.groupby("city").agg(
    sales_sum=("sales","sum"),
    sales_mean=("sales","mean"),
    sales_count=("sales","count"),
)
df2 = df2.reset_index()
```

This avoids MultiIndex entirely.

Result is already flattened:

| city | sales_sum | sales_mean | sales_count |
|---|---|---|---|
| London | 40 | 20 | 2 |
| Paris | 30 | 15 | 2 |

3. **Flatten after grouping multiple columns**

If you have:

```
1. df2 = df.groupby(["city","product"])["sales"].sum()
```

Flatten:

```
1. df2 = df2.reset_index()
```

Turns this:

```
1. city   product
2. London A         15
3. Paris  A         30
4. Paris  B          5
```

Into a normal DataFrame:

| city | product | sales |
|---|---|---|
| London | A | 15 |
| Paris | A | 30 |
| Paris | B | 5 |

**Summary**

| Goal | Command |
|---|---|
| Flatten grouped **rows** | `.reset_index()` |
| Flatten MultiIndex **columns** | `df.columns = df.columns.map("_".join)` |
| Avoid MultiIndex entirely | Use named aggregations in `.agg()` |

✅ **Mini-Practice:**

1. Compute **average sales by region**.
2. Count users per category.
3. Find **top scoring user per department** using groupby + transform.
4. Flatten grouped results for exporting.

## Key Takeaways

| Concept | Why it Matters | Analogy |
|---|---|---|
| Missing values | Real data gaps | Rotten or missing ingredients |
| Drop / Fillna | Clean data for calculations | Remove or replace bad veggies |
| Duplicates | Avoid double-counting | Two identical ingredients in dish |
| Column arithmetic | Create new insights | Mix ingredients to make a new dish |
| String ops | Standardize text | Chop and clean vegetables |
| GroupBy | Summarize by category | Organize ingredients by type |
| Aggregations | Compute metrics per group | Count/measure ingredients for recipes |
| Transform vs Filter | Advanced grouped calculations | Apply operations across each ingredient folder |

✅ **Suggested Mini-Project:**

- Take a small sales CSV:
    - Fill missing sales data with **mean per region**.
    - Remove duplicate entries.
    - Create new column **Profit** = **Revenue - Cost**.
    - Group by **Region** → compute **average Profit**.
    - Filter regions with **average Profit > 1000**.

# Merging, Joining & Concatenation (Combining Tables)

## 🎯 Goal

You'll often have data split across multiple files or tables, maybe one file for customers, another for their purchases.
You'll need to **combine** them into a single table.

Think of this like **assembling puzzle pieces**: each piece has part of the picture, but together, they form a complete view.

Think of Pandas like working with tables in Excel or SQL.

- **merge** → combine tables *side-by-side* based on matching columns
- **join** → same as merge, but easier when using indexes
- **concat** → stack tables *on top of each other* (or side-by-side without matching)

This table helps:

| ask | Use |
|---|---|
| Add columns from another table using a key | **merge** |
| Combine tables using index as key | **join** |
| Stack tables vertically (more rows) | **concat** |
| Stick tables horizontally (more columns) without matching keys | **concat** |

## Step 1: Concatenation (Stacking DataFrames)

Concatenation means **putting DataFrames together,** either one **on top of another (rows)** or **side by side (columns)**.

Two main modes:

**A. Vertical stacking (add rows)**

Example:

```
1. pd.concat([df1, df2], axis=0)
```

**df1**

| name | age |
|---|---|
| Alice | 25 |

**df2**

| name | age |
|---|---|
| Bob | 30 |

Result:

| name | age |
|---|---|
| Alice | 25 |
| Bob | 30 |

No matching needed, it just stacks.

**B. Horizontal stacking (add columns)**

```
1. pd.concat([df1, df2], axis=1)
```

This sticks them side-by-side, aligning by index.

**Analogy:** Imagine each month's sales sheet is one page, concatenating them stacks the pages into one big book.

# Step 2: Merging (Like SQL Joins)

**Merge** = SQL JOIN

You match rows between DataFrames using one or more key columns.

Example:

**Table A (df1)**

| id | name |
|---|---|
| 1 | Alice |
| 2 | Bob |

### Table B (df2)

| id | city |
|---|---|
| 1 | Paris |
| 2 | London |

Merge on "id":

```
1. df = pd.merge(df1, df2, on="id")
```

Result:

| id | name | city |
|---|---|---|
| 1 | Alice | Paris |
| 2 | Bob | London |

### Types of merges

**inner** (keep matching rows only)

```
1. pd.merge(df1, df2, on="id", how="inner")
```

**left** (keep all rows from left table)

```
1. pd.merge(df1, df2, on="id", how="left")
```

**right**

```
1. pd.merge(df1, df2, on="id", how="right")
```

### outer (keep everything)

```
1. pd.merge(df1, df2, on="id", how="outer")
```

## JOINING (merge using indexes)

Same idea as `merge`, but it defaults to using index instead of columns.

### Example

```
1. df1.join(df2)
```

This matches **`df1.index`** to **`df2.index`**.

### When to use `join`

- You already set indexes
- You want to combine many DataFrames easily:

```
1. df1.join([df2, df3])
```

### Join with a key (like merge but simpler)

```
1. df1.join(df2.set_index("id"), on="id")
```

If your index matches the join key:

Good when you already have **aligned indexes**, like time series.

## ⚠️ Pitfalls

- Different datatypes on join keys: `"1"` (string) vs `1` (int) won't match.
- Duplicates cause multiple matches.
- Missing join columns cause errors.

## ✅ Mini Practice

1. Merge **sales** and **customers** on `customer_id`.
2. Concatenate monthly reports into one big file.
3. Try different join types (`inner`, `outer`) to see differences.

# Reshaping Data (Pivot, Melt, Stack, Unstack)

## 🎯 Goal

Sometimes your data is not in the right **shape** for analysis or plotting.
You'll learn to **pivot (widen)** or **melt (lengthen)** your data, like reshaping clay.

Reshaping = changing the layout of your DataFrame.
Think of your table like clay, sometimes you want:

- wide format (more columns)
- long format (more rows)
- stacked (turn columns into an index)
- unstacked (turn index levels back into columns)

### Step 1: Pivot Tables, Like Excel Summaries

### A. PIVOT - long → wide

Makes columns from row values.

**Example starting table (long format)**

| city | year | sales |
|---|---|---|
| Paris | 2022 | 10 |
| Paris | 2023 | 20 |
| London | 2022 | 15 |
| London | 2023 | 25 |

**Goal**: one row per city, one column per year.

**Pivot:**

```
1. df.pivot(index="city", columns="year", values="sales")
```

**Result (wide format):**

| city | 2022 | 2023 |
|---|---|---|
| Paris | 10 | 20 |
| London | 15 | 25 |

❗ **Important:**

- **Each city-year pair must be unique**
  If duplicates exist → use **pivot_table** instead.

**B. PIVOT_TABLE** - safer pivot with aggregation

Same as pivot BUT:

- handles duplicates
- lets you choose how to combine them (`sum`, `mean`, etc.)

```
df.pivot_table(
    index="city",
    columns="year",
    values="sales",
    aggfunc="sum"
)
```

Good for:

- sales per product per year
- students per grade per class
- totals where duplicates exist

**Analogy**: Pivoting is like turning rows into columns so you can **compare side by side**.

# Step 2: Melt - wide → long (opposite of pivot)

Think of melting columns into rows.

**Example (wide):**

| city | 2022 | 2023 |
|---|---|---|
| Paris | 10 | 20 |
| London | 15 | 25 |

**Melt:**

```
1. df.melt(id_vars="city", var_name="year", value_name="sales")
```

**Result:**

| city | year | sales |
|---|---|---|
| Paris | 2022 | 10 |
| Paris | 2023 | 20 |
| London | 2022 | 15 |
| London | 2023 | 25 |

Melt is the **inverse** of pivot.

# Step 3: STACK - columns → index (makes DataFrame long)

Stack "pushes" columns down into the row index.

**Start:**

| city | 2022 | 2023 |
|---|---|---|
| Paris | 10 | 20 |

**Stack:**

```
1. df.stack()
```

**Result:**

```
1. city
2. Paris   2022    10
3.         2023    20
4. dtype: int64
```

**Now you have a MultiIndex Series:**

- **Level 1:** city
- **Level 2:** year

**B. UNSTACK — index → columns (reverse of stack)**

Take the MultiIndex and turn one level into columns.

If you start with:

```
city    year
Paris  2022    10
          2023    20
London 2022    15
          2023    25
```

**Unstack by "year":**

```
df.unstack(level="year")
```

You get wide form again:

| city | 2022 | 2023 |
|---|---|---|
| Paris | 10 | 20 |
| London | 15 | 25 |

**SUMMARY TABLE**

| Function | Direction | Description |
|---|---|---|
| `pivot` | long → wide | reshape using unique index-column pairs |
| `pivot_table` | long → wide | pivot with aggregation + handles duplicates |
| `melt` | wide → long | turn columns into rows |
| `stack` | wide → long | columns → MultiIndex rows |
| `unstack` | long → wide | MultiIndex rows → columns |

**FASTEST WAY TO REMEMBER**

- **pivot** = make new columns
- **melt** = remove columns ⟶ create rows
- **stack** = move columns ⟶ index
- **unstack** = move index ⟶ columns
- **pivot_table** = pivot + aggregation

**Analogy**: Stack = compressing levels down; Unstack = spreading them out again.

## ✅ Mini Practice

1. Pivot a transaction log to see **total sales per day per product**.
2. Melt a wide-format survey (columns = questions) into a long one (rows = answers).
3. Try stacking and unstacking to see how data structure changes.

# Time Series Basics with Pandas

## 🎯 Goal

Time-based data (like stock prices or sales per day) needs special handling. Pandas is **excellent** for this.

A **time series** is simply data where the index represents **time**:

- days
- hours
- months
- timestamps
- etc.

### Example:

| date | sales |
|---|---|
| 2023-01-01 | 10 |
| 2023-01-02 | 15 |
| 2023-01-03 | 12 |

Time series analysis means:

- converting strings to datetimes
- setting time as the index
- resampling (group by time periods)
- rolling windows (moving averages)

### Convert column to datetime

This is the MOST important first step:

```
1. df["date"] = pd.to_datetime(df["date"])
```

Why?

- Pandas will now recognize it as real date/time.
- You can resample, extract year/month/day easily.

## Set the datetime as index

Many time-series operations require this:

```
df = df.set_index("date")
```

Your DataFrame becomes:

```
                sales
date
2023-01-01    10
2023-01-02    15
2023-01-03    12
```

This is the classic time-series format.
**Analogy**: Imagine taking a daily temperature log and summarizing it **per week**.

## Sorting by time

Always sort your index:

```
df = df.sort_index()
```

## Extract datetime components

Once converted, it's easy:

```
df["year"] = df.index.year
df["month"] = df.index.month
df["day"] = df.index.day
df["weekday"] = df.index.day_name()
```

**Example:**

| date | sales | year | month | weekday |
|---|---|---|---|---|
| 2023-01-01 | 10 | 2023 | 1 | Sunday |

### Resampling (VERY important)

Resampling = grouping by time periods.

**Examples:**

**Daily → Monthly**

```
1. df.resample("M")["sales"].sum()
```

**Weekly average**

```
1. df.resample("W")["sales"].mean()
```

**Hourly → 15-min bins**

```
1. df.resample("15T").sum()
```

Common time codes:

| Code | Meaning |
|---|---|
| `D` | day |
| `W` | week |
| `M` | month end |
| `Q` | quarter |
| `Y` | year |
| `H` | hour |
| `T` or `min` | minute |

## Rolling windows (moving window calculations)

Moving average (7-day window):

```
1. df["7day_avg"] = df["sales"].rolling(7).mean()
```

Rolling sum:

```
1. df["7day_sum"] = df["sales"].rolling(7).sum()
```

This is extremely common in forecasting and smoothing noisy data.

## Shifting

Shift data forward/backward in time.

Shift down 1 day:

```
1. df["yesterday"] = df["sales"].shift(1)
```

### Percentage change:

```
1. df["pct_change"] = df["sales"].pct_change()
```

### Filtering by dates

Select date range:

```
1. df.loc["2023-01-01":"2023-01-10"]
```

Select a specific month:

```
1. df.loc["2023-02"]
```

Select a specific year:

```
1. df.loc["2023"]
```

## Date offsets (adding/subtracting time)

Add 1 month:

```
1. df.index + pd.DateOffset(months=1)
```

Subtract 7 days:

```
1. df.index - pd.Timedelta(days=7)
```

## Summary

| Task | Code |
|---|---|
| Convert to datetime | `pd.to_datetime()` |
| Set index to date | `.set_index("date")` |
| Resample | `.resample("M").sum()` |
| Rolling window | `.rolling(7).mean()` |
| Shift | `.shift()` |
| Select date ranges | `df.loc["2023-01"]` |

✅ **Mini Practice**

1. Load daily sales data.
2. Convert date column → datetime.
3. Resample to weekly totals.
4. Compute 7-day rolling average.
5. Compare to previous day with `.shift()`.

## 💡 Tip:

Start simple, ignore time zones until you need them (for global data).
Always use **`pd.to_datetime()`** before working with date columns.

## Summary Table

| Concept | Key Function | Analogy |
|---|---|---|
| Concatenate | `pd.concat()` | Stack pages of a book |
| Merge | `pd.merge()` | Match puzzle pieces |
| Pivot | `df.pivot_table()` | Rotate table like Excel |
| Melt | `df.melt()` | Unfold wide data into long |
| Resample | `df.resample()` | Group by time periods |
| Rolling | `df.rolling()` | Sliding window for trends |

✅ **Mini-Project Idea (for Week 9):**

- Load 3 months of sales CSVs → concatenate into a yearly dataset.
- Merge with customer info.
- Resample daily sales to weekly totals.
- Add a rolling average and visualize trend (using matplotlib or seaborn later).

# Performance & Scaling (Speed Tricks)

When you work with small datasets (hundreds or thousands of rows), pandas feels lightning fast. But in the real world (millions of rows or large CSVs), **performance optimization** becomes critical.

Think of pandas like a **sports car**:

- It can go incredibly fast **if you stay on the smooth road (vectorized operations)**.
- But if you try to take it off-road (manual Python loops, `.apply()` abuse), it slows down badly.

## Key Ideas

### 1. Vectorization

- Instead of looping through each row, pandas uses **NumPy under the hood**, applying operations to entire columns at once, like doing 1 million calculations in one shot.
- Example:

```
1. df['price_with_tax'] = df['price'] * 1.07
```

is way faster than looping with `for` or **`.apply()`**.

Think of a factory assembly line, vectorized operations are like machines working on all products at once. Python loops are like a single worker doing them by hand.

### 2. Categorical Data Types

- Columns with **repeated text values** (like cities, product names, categories) can use less memory when stored as `category`.
- Example:

```
1. df['city'] = df['city'].astype('category')
```

This saves memory **and** speeds up comparisons and groupby operations.

Instead of storing "New York" 10,000 times, pandas stores it once and uses integer codes internally.

### 3. Avoid `.apply()` When Possible

- **`.apply()`** feels powerful, but it runs Python functions row by row (slow).
- Prefer native methods or vectorized expressions.

Example:

```
df['total'] = df['price'] + df['tax']    # Fast
df['total'] = df.apply(lambda x: x['price'] + x['tax'], axis=1)   # Slow
```

### 4. Efficient Memory Management

- Use **`.memory_usage(deep=True)`** to inspect how much memory each column consumes.
- Convert heavy columns using:
  - **`astype('category')`**
  - **`astype('float32')` or `int32` (when safe)**

**Example**:

```
df.memory_usage(deep=True)
```

### 5. Chunked Processing

For massive files, load in **chunks**:

```
chunks = pd.read_csv('bigfile.csv', chunksize=10000)
for chunk in chunks:
    process(chunk)
```

Each chunk is a small DataFrame, letting you process big data without crashing RAM.
Like reading a giant novel **one chapter at a time** instead of trying to memorize it all at once.

## Performance Tools

| Method | Description | Use Case |
|---|---|---|
| `.eval()` | Evaluates expressions using optimized backend | Fast column math |
| `.query()` | Filters data with fast expressions | Filtering with conditions |
| `.astype('category')` | Converts text to categorical | Reduces memory |
| `.memory_usage(deep=True)` | Shows memory use per column | Debugging performance |
| `pd.read_csv(..., chunksize=n)` | Reads large data in parts | Big data handling |

## Practice

1. Load a 1M-row CSV.
2. Convert text columns to `category`.
3. Compare memory usage before and after.
4. Replace `.apply()` with vectorized math.
5. Use `.query()` for fast filtering.

# Visualization Quick Hits (pandas + matplotlib)

Before deep data visualization (like with Seaborn or Plotly), pandas gives you **quick, built-in plots** for **Exploratory Data Analysis (EDA),** to understand patterns fast.

Think of it like **the diagnostic dashboard of your car** not fancy, but gives you essential signals.

## Plot Basics

Any DataFrame column can be visualized using:

```
1. df['sales'].plot(kind='line')
```

or multiple columns:

```
1. df[['profit', 'sales']].plot(kind='line')
```

## Common Plot Types

| Type | Method | Purpose |
|---|---|---|
| **Line plot** | `df.plot(kind='line')` | Trend over time |
| **Histogram** | `df.plot(kind='hist')` | Distribution of numeric values |
| **Scatter** | `df.plot(kind='scatter', x='age', y='income')` | Relationship between two variables |
| **Bar plot** | `df.plot(kind='bar')` | Compare categorical data |
| **Box plot** | `df.plot(kind='box')` | Detect outliers |
| **Area plot** | `df.plot(kind='area')` | Show cumulative totals |

## Practice

1. Plot the distribution of `salary`.
2. Plot sales trend over months.
3. Scatter plot `marketing_spend` vs `profit`.
4. Add `title`, `xlabel`, and `ylabel` for readability.

EDA plots are like **x-rays** of your dataset, they don't tell the whole story but help spot what's broken or interesting fast.

# Advanced Topics (Optional but Powerful)

Now we dive into **professional-grade pandas** features used in analytics pipelines and advanced modeling prep.

## 1. Window Functions

Window functions compute values over a sliding window of rows.

They are extremely useful for:

- moving averages
- rolling sums
- cumulative totals
- ranking values
- smoothing time series data

### A. Rolling Window

A rolling window looks at the previous N rows, then computes something.

**Example:** 3-day moving average

```
1. df["ma_3"] = df["sales"].rolling(3).mean()
```

If your sales are:

| day | sales |
|---|---|
| 1 | 10 |
| 2 | 20 |
| 3 | 30 |
| 4 | 40 |

**The moving average is:**

| day | sales | ma_3 |
|---|---|---|
| 1 | 10 | NaN |
| 2 | 20 | NaN |
| 3 | 30 | 20 |
| 4 | 40 | 30 |

**Why are the first values NaN?**

A 3-day moving averages means:
"Average the current day + previous 2 days."

So for the first rows:

**Row 1**

There are no previous days ⟶ needs 3 values, but ony one exists ⟶ Nan

**Row 2**

Still not enough values ⟶ only 2 exist ⟶ Nan

**Row 3**

Now you have 3 values ⟶ first valid average

## Example

Assume sales = `[10, 20, 30, 40]`

| Day | Sales | 3-Day Moving Average |
|---|---|---|
| 1 | 10 | NaN (needs 10, ?, ?) |
| 2 | 20 | NaN (needs 10, 20, ?) |
| 3 | 30 | (10 + 20 + 30) / 3 = **20** |
| 4 | 40 | (20 + 30 + 40) / 3 = **30** |

So you need 3 rows before you can compute a 3-row average.

**How does a moving average work?**

**A moving average is a sliding window calculation.**

For a window of size 3:

**At row 3**

Window = rows [1, 2, 3] ⟶ average ⟶ 20

**At row 4**

Window = rows [2, 3, 4] ⟶ average ⟶ 30

**At row 5**

Window = rows [3, 4, 5] ⟶ average ⟶ etc.

Each time you move down one row, the window "slides" down.

### Visual picture

For sales:

```
1. 20   30   40   50
```

Window size = 3:

```
1. [10 20 30] 40 50   ⟶ first window
2.  10 [20 30 40] 50  ⟶ second window
3.  10 20 [30 40 50]  ⟶ third window
```

Each window is averaged to produce one output value.

### Why does Pandas give NaN?

Because Pandas defaults to:

```
1. min_periods = window_size
```

**Meaning:**

> "Don't compute the average until you have the full window."

### But what if I *want* values for the first rows?

You can override the default:

```
1. df["ma3"] = df["sales"].rolling(3, min_periods=1).mean()
```

Now Pandas will use whatever data exists:

| Day | Sales | MA3 with min_periods=1 |
|---|---|---|
| 1 | 10 | 10 |
| 2 | 20 | (10 + 20) / 2 = 15 |
| 3 | 30 | (10 + 20 + 30) / 3 = 20 |
| 4 | 40 | 30 |

**In short:**

- Moving average of size N needs **N rows**
- Until you reach the Nth row, result is **NaN**
- This is expected and correct
- You *can* force it to compute earlier by using **`min_periods=1`**

## 2. Categorical Ordering

Use categoricals when:

- a column has repeated values
- the order matters (e.g. **small** < **medium** < **large**)
- the column should be memory-efficient

You can order categories manually for logical sorting or plotting.

### A. Convert a column to categorical

```
1. df["size"] = pd.Categorical(df["size"])
```

### B. Ordered categories

**Example:** S, M, L, XL

```
1. df["size"] = pd.Categorical(
2.     df["size"],
3.     categories=["S", "M", "L", "XL"],
4.     ordered=True
5. )
```

Now comparisons work correctly:

```
1. df[df["size"] > "M"]
```

Returns rows where size is L or XL.

### B. Sorting uses the defined order

```
1. df.sort_values("size")
```

Will follow S < M < L < XL (not alphabetical order).

Instead of sorting alphabetically, you define priority levels, like "Low < Medium < High."

## 3. MultiIndex (Hierarchical Index)

A MultiIndex allows **multiple levels of indexing** for rows or columns.

Very useful for:

- grouped data
- panel data
- pivot tables
- complex time series (e.g., city → year → month)

### A. Creating a MultiIndex from columns

```
1. df = df.set_index(["city","year"])
```

Now your index looks like:

```
1. city    year
2. Paris   2022
3. Paris   2023
4. London  2022
5. London  2023
```

### B. Selecting data

```
1. df.loc["Paris"]
```

Or select Paris 2023:

```
1. df.loc[("Paris", 2023)]
```

### C. Resetting the index

```
1. df.reset_index()
```

Back to normal columns.

### D. Columns can also have MultiIndex

This happens after **`pivot_table`**, **`agg`**, **etc.**

Flatten them:

```
1. df.columns = df.columns.map("_".join)
```

## 4. DISPLAY OPTIONS (viewing big DataFrames better)

Pandas has several display settings to make large tables easier to inspect.

**Show more columns**

```
pd.set_option("display.max_columns", 100)
```

**Increase column width**

```
pd.set_option("display.max_colwidth", None)
```

**Reset all display settings**

```
pd.reset_option("all")
```

**View entire DataFrame without truncation**

```
pd.set_option("display.width", None)
pd.set_option("display.max_colwidth", None)
```

## 5. Interoperability

Pandas integrates with almost everything:

**CSV**

```
df = pd.read_csv("file.csv")
df.to_csv("out.csv", index=False)
```

**Excel**

```
df = pd.read_excel("file.xlsx")
df.to_excel("out.xlsx", index=False)
```

**Parquet**

Efficient for big data:

```
df = pd.read_parquet("file.parquet")
df.to_parquet("out.parquet")
```

**JSON**

```
df.to_json("file.json", orient="records")
```

**SQL Databases**

```
1. df.to_sql("table", engine)
2. pd.read_sql("SELECT * FROM table", engine)
```

**Numpy**

```
1. arr = df.to_numpy()
2. df = pd.DataFrame(arr)
```

**Interacting with Matplotlib**

```
1. df["sales"].plot()
```

**Interacting with Polars, PySpark, Arrow**

Convert to Arrow:

```
1. table = df.to_arrow()
```

**Convert to Pandas:**

```
1. df = table.to_pandas()
```

# SUMMARY

| Feature | Purpose |
|---|---|
| **Window functions** | Rolling averages, cumulative values |
| **Categorical ordering** | Memory-efficient categories + custom order |
| **MultiIndex** | Multiple levels of index (hierarchical data) |
| **Display options** | Better viewing of large tables |
| **Interoperability** | Read/write CSV, Excel, Parquet, SQL, JSON |

## Practice

1. Create a rolling 7-day average sales.
2. Sort stages as categorical ordered levels.
3. Create MultiIndex pivot and use `.xs()` to select subset.
4. Adjust display settings to see formatted outputs.

## Summary: How These Final Modules Fit Together

| Module | Focus | Real-World Analogy |
|---|---|---|
| **Performance & Scaling** | Make pandas fast & memory-efficient | Sports car tuning |
| **Visualization** | Quick data x-rays | Doctor’s diagnostics |
| **Advanced Topics** | High-level control for complex analysis | Master dashboard with all knobs |

# Testing, Reproducibility & Pipeline Basics

### Why This Matters

When you start doing serious work, whether in data analysis, AI, or web projects, you'll quickly realize this:

> "It's not enough to make your code work once. It needs to work every time and for anyone who runs it."

That's what this module teaches:
How to make your **pandas scripts predictable, repeatable, and reliable.**

Imagine you clean and analyze a dataset today, and your manager asks you to redo it next week, you must be able to **get the exact same results**, not slightly different ones.
That's what **reproducibility** ensures.

## Step 1: Understanding Reproducibility

### What Does "Reproducible" Mean?

Reproducibility means:

> You, or anyone else, can run your code later and get *the same results.*

It's like writing a **recipe** not just cooking once, but being able to cook the *exact same meal* again, even months later.

### Common Problems Without Reproducibility

| Problem | Example | Why It's Dangerous |
|---|---|---|
| Random results | A machine learning model gives slightly different scores each time. | You can't compare models or trust your metrics. |
| Hidden data changes | You cleaned the file but didn't save it, now the next run breaks. | Your project can't be rerun by others. |
| Mixed data types | A column becomes "object" instead of "**int**." | Aggregations or numeric functions fail. |

## How to Make Code Reproducible

### 1. Set Random Seeds

When you use randomness (e.g., splitting data or sampling), always fix the "seed", a number that ensures the random process behaves the same every time.

```
1. import numpy as np
2. np.random.seed(42)
```

Now every random shuffle or sample you take from numpy will be identical across runs.

Imagine shuffling a deck of cards, but every time you use the same pattern, you'll always get the same order.

### 2. Be Consistent with Data Types

If a column switches between **`int`**, **`float`**, or **`object`** types, you'll get strange errors later.

Always check and convert explicitly:

```
1. df['age'] = df['age'].astype(float)
2. df['joined_date'] = pd.to_datetime(df['joined_date'])
```

If you're baking, you wouldn't mix tablespoons and grams randomly, same with dtypes.

### 3. Save Clean or Processed Data

Never rely on your memory or temporary notebook variables.
After you clean your dataset, save it:

```
1. df_clean.to_csv('cleaned_data.csv', index=False)
```

That way, next time you or your team runs analysis, you can start from the cleaned version — not redo everything.

# Pipelines - Automating Your Cleaning Steps

## What is a“Pipeline”?

A **pipeline** is just a **step-by-step process that transforms your raw data into ready-to-use data,** but written as *code* instead of manual steps.

Instead of cleaning your dataset cell by cell in Jupyter, you combine all the steps into a single reusable function.

## Example: Before (Manual)

```
df = pd.read_csv('sales.csv')
df.drop_duplicates(inplace=True)
df['price'] = df['price'].fillna(df['price'].median())
df['region'] = df['region'].fillna('Unknown')
df.to_csv('sales_clean.csv', index=False)
```

This works, but it’s messy if you need to repeat it 10 times.

## After (Reusable Pipeline)

Let’s wrap that into a function:

```
def clean_data(df):
    # Remove duplicate rows
    df = df.drop_duplicates()

    # Fill missing numeric values with median
    df['price'] = df['price'].fillna(df['price'].median())

    # Fill missing categorical values with a default
    df['region'] = df['region'].fillna('Unknown')

    # Standardize text
    df['region'] = df['region'].str.strip().str.lower()

    return df
```

Now, whenever you have a new sales dataset:

```
df = pd.read_csv('sales_july.csv')
df_clean = clean_data(df)
```

You've made your own **data cleaning machine**!
Every dataset goes in messy, and comes out ready to use.

### Why Functions (Pipelines) Are Important

| Without Function | With Function |
|---|---|
| Hard to repeat | Reusable anytime |
| Many lines in notebook | One clean call |
| Error-prone | Consistent results |
| Hard for others to understand | Easy to share and test |

It's like making a washing machine instead of washing clothes by hand every time.

## Step 3: Testing -"Did My Cleaning Work?"

Testing means **checking that your pipeline did what you expected.**

You don't need to know formal unit testing yet, just add small **assertions** or checks.

### Example

```
1. df_clean = clean_data(df)
2.
3. # Test: No missing values in important columns
4. assert df_clean['price'].notna().all(), "Price column still has NaNs!"
5.
6. # Test: No duplicates left
7. assert not df_clean.duplicated().any(), "Duplicate rows remain!"
```

If a condition fails, your code will stop and show an error, that's a good thing!
It means your cleaning pipeline **caught a data issue early** before it spreads.

It's Like a smoke detector, you want it to go off if something's burning.

## Step 4: Organizing for Reuse

As you progress, it's good practice to **save your pipeline function in a script** (like **`utils.py`**).

Example project structure:

```
project/
├── data/
│   ├── raw/
│   └── clean/
├── scripts/
│   └── clean_pipeline.py
└── analysis.ipynb
```

Then import and use it:

```
from scripts.clean_pipeline import clean_data
df = pd.read_csv('data/raw/sales.csv')
df_clean = clean_data(df)
```

## Why All This Matters

| Concept | Why It Matters |
|---|---|
| **Reproducibility** | Ensures same results across runs and systems |
| **Pipelines** | Automate repetitive work and reduce errors |
| **Testing** | Prevents silent bugs and broken data |
| **Consistency** | Keeps data types and results predictable |
| **Reusability** | Lets you apply the same logic to new datasets quickly |

## Recap

Imagine you're running a laundry service:

- **Raw clothes = messy data**
- **Washing process = your pipeline**
- **Detergent, water temp = cleaning steps**
- **Quality check = testing**
- **Same wash settings = reproducibility**

Every time, you get clean clothes (data) that pass inspection **fast, consistent, repeatable.**

## Mini Practice Challenge

1. Take a small messy CSV (like one with missing values and duplicates).
2. Write a `clean_data(df)` function that:
    - Removes duplicates
    - Fills NaNs
    - Converts dates correctly
3. Save your cleaned version with `to_csv('cleaned.csv')`.
4. Add `assert` checks to verify it worked.

You've just built your **first data cleaning pipeline**, the foundation for real-world projects and machine learning workflows.

# Mini Projects (Applied Pandas)

Now, this is where all previous modules come together.
Each mini project focuses on a **real-world scenario**, combining multiple pandas skills.

Let's unpack what each teaches:

## Project A - Exploratory Data Analysis (EDA) Report

**Goal:** Learn to **understand data before modeling.**

- Load data (like Titanic)
- Clean missing values
- Create visual summaries (histograms, correlations)
- Group by key variables (e.g., survival rate by gender)

*Skills combined:* IO, cleaning, selection, grouping, plotting.
Like being a detective, observing patterns before solving the case.

## Project B - Data Cleaning Pipeline

**Goal:** Automate messy data cleanup.

- Use `.isna()`, `.fillna()`, `.astype()`, `.drop_duplicates()`
- Save the process as a reusable function or script
- Export clean data to `cleaned.csv`

Like building your own **"data washing machine."**

## Project C - Small ETL (Extract, Transform, Load)

**Goal:** Simulate real-world workflow.

- Extract: load multiple daily CSVs
- Transform: clean, combine, and aggregate
- Load: save the processed monthly report

You're now an **engineer building the pipes** that data flows through.

## Project D - Kaggle-Style Starter

**Goal:** Prepare data for machine learning.

- Load dataset (e.g., Housing Prices)
- Clean and encode features
- Handle missing values and outliers
- Export a ready-to-model CSV

You're now the **chef prepping ingredients** before cooking the ML dish.

# Essential Pandas Functions (with Explanations & Why They Matter)

## 🗂️ I/O (Input/Output)

**Functions:**
**`read_csv`, `to_csv`, `read_excel`, `to_excel`**

**Purpose:** Bring data *in* and *out* of Pandas.

- **`read_csv("data.csv")`**: loads your dataset from a CSV file.
- **`to_csv("cleaned.csv")`**: saves your DataFrame after cleaning.
- Excel equivalents handle **`.xlsx`** files.

**Why it matters:** Everything starts with reading data, and ends with saving results. You'll use I/O functions in every project.

## 👀 Inspect

**Functions:**
`head`, `tail`, `info`, `describe`, `dtypes`, `shape`

**Purpose:** Quickly *understand* your data.

- `head()` / `tail()`: peek at first or last few rows.
- `info()`: data types and null counts.
- `describe()`: summary stats (mean, std, min, etc.).
- `dtypes` & `shape`: know variable types and dimensions.

**Why it matters:** Inspection helps you "see" the structure and health of your dataset before diving into cleaning or analysis.

## 🎯 Selection

**Functions:**
**`df[col]`, `df[['a','b']]`, `loc`, `iloc`, `at`, `iat`**

**Purpose:** Extract specific rows, columns, or cells.

- **`df['col']`**: one column.
- **`df[['a','b']]`**: multiple columns.
- **`loc[row_label, col_label]`**: label-based.
- **`iloc[row_index, col_index]`**: index-based.
- **`at`**, **`iat`**: for very fast access to single cells.

**Why it matters:** Selection is your daily bread, every analysis depends on filtering or focusing on certain parts of your data.

## ⚠️ Missing Data

**Functions:**
**`isna`**, **`fillna`**, **`dropna`**

**Purpose:** Handle incomplete or NaN values.

- `isna()`: detect missing data.
- `fillna(value)`: replace missing values.
- `dropna()`: remove missing rows or columns.

**Why it matters:** Dirty or missing data breaks analysis and models. You *must* clean or impute before moving forward.

## 👥 Grouping & Aggregation

**Functions:**
**`groupby`**, **`agg`**, **`transform`**

**Purpose:** Summarize or compute stats by groups.

- **`df.groupby('category').sum()`**

- `agg({'price':'mean', 'quantity':'sum'})`
- `transform()` applies functions but keeps original size.

**Why it matters:** This is how you move from raw data to insights, total sales per region, average score per class, etc.

## 🔗 Merging, Joining & Concatenation

**Functions:**
`merge`, `concat`, `join`

**Purpose:** Combine datasets like SQL joins.

- `merge(df1, df2, on='id', how='left')`: join by key.
- `concat([df1, df2], axis=0|1)`: stack vertically or horizontally.
- `join`: merge by index.

**Why it matters:** Real-world data often lives in multiple files/tables, you must know how to stitch them together properly.

## 🔄 Reshaping

**Functions:**
`pivot_table`, `melt`, `stack`, `unstack`

**Purpose:** Change the "shape" of your data.

- `pivot_table()`: wide-format summary tables.
- `melt()`: convert wide → long format for modeling.
- `stack/unstack`: move between multi-index levels.

**Why it matters:** Data for visualization or ML often needs reshaping. This gives flexibility in analysis workflows.

## ⏰ Time Series

**Functions:**
`to_datetime`, `resample`, `rolling`, `shift`

**Purpose:** Work with time-indexed data.

- `to_datetime()`: convert date strings to timestamps.

- **`resample('W')`**: aggregate daily → weekly or monthly.
- **`rolling(window=7).mean()`**: moving averages.
- **`shift(1)`**: lag values for comparisons.

**Why it matters:** Time-based trends and rolling windows are essential for forecasting, trading, and monitoring patterns.

### ⚡ Performance

**Functions:**
**`astype('category')`**, **`eval`**, **`query`**

**Purpose:** Speed up code and reduce memory use.

- **`astype('category')`**: compress repeating text.
- **`eval("col1 + col2")`**: vectorized math without loops.
- **`query("age > 30")`**: fast filtering using strings.

**Why it matters:** Efficient data operations matter for large datasets. These tricks make your code cleaner, faster, and scalable.

## Final Summary - The Panda's Journey 🐼

| Stage | Goal | Key Skills |
|---|---|---|
| **1. Load Data** | Bring data into Pandas | `read_csv`, `read_excel` |
| **2. Inspect** | Understand structure | `head`, `info`, `describe` |
| **3. Clean** | Handle missing or wrong types | `fillna`, `astype` |
| **4. Transform** | Select, group, and reshape | `loc`, `groupby`, `pivot_table` |
| **5. Combine** | Merge multiple sources | `merge`, `concat`, `join` |
| **6. Analyze** | Compute summaries & patterns | `agg`, `rolling`, `resample` |
| **7. Visualize** | Quick insights | `plot()` |
| **8. Optimize** | Make it fast | `eval`, `query`, `astype('category')` |
| **9. Reuse** | Write reproducible code | wrap cleaning & analysis in functions |

---