📘 Pandas Tutorial (Zero to Hero)

1. Introduction to Pandas

- What is Pandas? (built on NumPy, for data manipulation & analysis)
- Why Pandas vs Python lists/dictionaries/NumPy?
- Installing Pandas

---

2. Pandas Basics

- `Series` (1D labeled array)
- `DataFrame` (2D labeled data structure)
- Creating Series & DataFrames (from lists, dicts, NumPy arrays, CSV, Excel)

---

3. DataFrame Attributes

- `.shape`, `.ndim`, `.dtypes`, `.columns`, `.index`, `.values`
- Getting quick insights: `.head()`, `.tail()`, `.info()`, `.describe()`

---

4. Indexing & Selection

- Column selection (`df['col']`, `df.col`)
- Row selection (`.loc`, `.iloc`)
- Slicing DataFrames
- Boolean indexing & filtering
- Setting index & resetting index

---

5. Data Cleaning

- Handling missing values (`isna`, `fillna`, `dropna`)
- Handling duplicates (`duplicated`, `drop_duplicates`)
- Renaming columns
- Changing data types (`astype`)

---

6. Operations

- Arithmetic operations
- String operations (`.str`)
- Apply functions (`apply`, `map`, `applymap`)
- Sorting (`sort_values`, `sort_index`)

---

7. Grouping & Aggregation

- `groupby` basics
- Aggregate functions (`sum`, `mean`, `count`, `agg`)
- Pivot tables

---

8. Merging, Joining & Concatenation

- `concat`, `merge`, `join`
- Differences between them
- Examples (like SQL joins)

---

9. Working with Dates & Time

- `pd.to_datetime`
- Extracting year, month, day
- Resampling & time-series analysis

10. Input/Output with Files

- Reading: `read_csv`, `read_excel`, `read_json`, `read_sql`
- Writing: `to_csv`, `to_excel`, `to_json`

---

11. Advanced Topics

- MultiIndex (hierarchical indexing)
- Window functions (rolling, expanding)
- Categorical data
- Sparse data
- Performance tips (vectorization, `.eval`, `.query`)

---

12. Exercises & Projects

- Small exercises after each section
- Mini-projects at the end:
  - Analyzing a CSV dataset (e.g., Titanic dataset, sales dataset)
  - Cleaning messy data (missing values, duplicates, etc.)
  - Simple exploratory data analysis (EDA)

⚡ **Extra Tips & Tricks**

- Use `.copy()` to avoid SettingWithCopyWarning
- Prefer `loc` over chained indexing
- Use `df.query()` for cleaner filtering
- Use `.astype('category')` for memory optimization

```
Start coding or generate with AI.
```

📘 1. Introduction to Pandas

🔷 What is Pandas?

- **Pandas** is an open-source **Python library** used for **data manipulation and analysis**.
- It is **built on top of NumPy**, which means it uses NumPy arrays under the hood for fast computations.
- It provides two main data structures:
  - **Series** → 1D labeled data (like a column in Excel)
  - **DataFrame** → 2D labeled data (like a full Excel sheet or SQL table)

👉 In short: Pandas is the **go-to library** for working with structured (tabular) data in Python.

🔷 Why Pandas vs. Python lists/dictionaries/NumPy?

- **Python lists/dictionaries**
  - Good for small tasks, but become slow & messy with large datasets.
  - No built-in tools for filtering, grouping, or joining data.
- **NumPy arrays**
  - Great for numerical data & fast calculations.
  - But lacks **labels** (column names, row indices).
  - Harder to handle **heterogeneous data** (numbers + text).
- **Pandas** ✅ Combines the **speed of NumPy** with the **flexibility of labels**. ✅ Offers **built-in methods** for filtering, cleaning, grouping, merging, reshaping. ✅ Works seamlessly with **CSV, Excel, SQL, JSON, and more**.

🔷 Installing Pandas

Pandas doesn't come pre-installed with Python. Install it using:

```
pip install pandas
```

Or if you're using **Anaconda** (recommended for data science):

```
conda install pandas
```

Verify the installation inside Python:

```
import pandas as pd
print(pd.__version__)
```

```
Start coding or generate with AI.
```

📘 2. Pandas Basics

Pandas has two main data structures:

1. **Series** → One-dimensional (1D) labeled array.
2. **DataFrame** → Two-dimensional (2D) labeled data structure (rows + columns).

🔷 2.1 Series (1D Labeled Array)

- Think of a **Series** like a single column in Excel or a single column in a SQL table.
- It has two parts:
  - **Values** → actual data (numbers, strings, etc.)
  - **Index** → labels for each value

Example 1: Creating a Series from a list

```
import pandas as pd

# Create a Series from a list
data = [10, 20, 30, 40]
s = pd.Series(data)
print(s)
```

Output:

```
0    10
1    20
2    30
3    40
dtype: int64
```

👉 By default, Pandas assigns **integer indices (0,1,2,3)**.

Example 2: Custom Index

```
s = pd.Series([10, 20, 30, 40], index=["A", "B", "C", "D"])
print(s)
```

Output:

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

```
D    40
dtype: int64
```

👉 Now, instead of default numbers, we gave labels.

Example 3: From Dictionary

```
data = {"Apple": 3, "Banana": 5, "Orange": 2}
s = pd.Series(data)
print(s)
```

Output:

```
Apple     3
Banana    5
Orange    2
dtype: int64
```

🔷 2.2 DataFrame (2D Labeled Data Structure)

- A **DataFrame** is like an Excel sheet or SQL table → rows + columns.
- Each column is a **Series**, and the whole table is a **DataFrame**.

Example 1: From Dictionary of Lists

```
data = {
    "Name": ["Ali", "Sara", "John"],
    "Age": [25, 30, 28],
    "City": ["Cairo", "Paris", "London"]
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
   Name  Age    City
0   Ali   25   Cairo
1  Sara   30   Paris
2  John   28  London
```

Example 2: From List of Dictionaries

```
data = [
    {"Name": "Ali", "Age": 25, "City": "Cairo"},
    {"Name": "Sara", "Age": 30, "City": "Paris"},
    {"Name": "John", "Age": 28, "City": "London"}
]

df = pd.DataFrame(data)
print(df)
```

Example 3: From NumPy Array

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
df = pd.DataFrame(arr, columns=["A", "B", "C"])
print(df)
```

Output:

```
   A  B  C
0  1  2  3
1  4  5  6
```

Example 4: From CSV/Excel File

```
# From CSV
df_csv = pd.read_csv("data.csv")

# From Excel
df_excel = pd.read_excel("data.xlsx")

print(df_csv.head())   # First 5 rows
print(df_excel.head())
```

📝 Exercises

Exercise 1

Create a **Series** that represents the number of books each student has:

- Ali: 3, Sara: 7, John: 5, Noor: 9

Exercise 2

Create a **DataFrame** for the following student data:

Name	Age	Grade
Ali	20	A
Sara	22	B
John	21	A
Noor	23	C

Exercise 3

Use **NumPy** to create a 2D array of shape (3,3), then convert it into a Pandas **DataFrame** with column names `X, Y, Z`.

✅ Solutions

Solution 1

```
books = {"Ali": 3, "Sara": 7, "John": 5, "Noor": 9}
s = pd.Series(books)
print(s)
```

Solution 2

```
data = {
    "Name": ["Ali", "Sara", "John", "Noor"],
    "Age": [20, 22, 21, 23],
    "Grade": ["A", "B", "A", "C"]
}

df = pd.DataFrame(data)
print(df)
```

Solution 3

```
import numpy as np

arr = np.array([[1,2,3], [4,5,6], [7,8,9]])
df = pd.DataFrame(arr, columns=["X", "Y", "Z"])
print(df)
```

```
Start coding or generate with AI.
```

3) DataFrame Attributes — Deep Dive

We'll use this small dataset throughout:

```
import pandas as pd

df = pd.DataFrame({
    "Name": ["Ali", "Sara", "John", "Noor", "Mona", "Omar"],
    "Dept": ["IT", "HR", "Finance", "IT", "HR", "Finance"],
    "Age": [20, 22, 21, 23, 29, 35],
    "Salary": [5000, 4500, 6000, 5500, 5200, 7200],
    "Joined": pd.to_datetime(["2022-01-10","2021-05-03","2020-07-19","2022-02-01","2019-09-30","2018-11-11"]),
    "Remote": [True, False, None, True, False, True]
})
```

A) Shape & Dimensionality

`.shape` → `(rows, columns)`

```
df.shape      # e.g., (6, 6)
```

`.ndim` → number of dimensions

- `2` for DataFrame; `1` for Series.

```
df.ndim       # 2
```

Related helpers

```
len(df)       # rows
df.size       # rows * columns (total cells)
df.axes       # [row_index, column_index]
```

B) Dtypes (Data Types)

`.dtypes` — each column's dtype

```
df.dtypes
```

Typical dtypes: `int64`, `float64`, `bool`, `object` (strings/mixed), `datetime64[ns]`, `category`, and **nullable** types like `Int64`, `boolean`, `string`.

Pick columns by dtype

```
df.select_dtypes(include="number")
df.select_dtypes(exclude=["object","datetime"])
```

Convert dtypes (smartly)

```
df2 = df.convert_dtypes()  # uses pandas' nullable types where possible
```

Force a dtype

```
df["Dept"] = df["Dept"].astype("category")
```

> Tip: `object` often means "strings" but can hide mixed types; converting to `category` or `string` saves memory and clarifies intent.

C) Columns & Index

`.columns` — column labels (an `Index`)

```
df.columns
df.columns.tolist()
```

Rename or set:

```
df = df.rename(columns={"Dept": "Department"})
df.columns = [c.lower() for c in df.columns]   # overwrite all at once
```

`.index` — row labels

```
df.index           # RangeIndex(...) by default
df.set_index("name", inplace=False)   # use a column as index (returns a copy)
df.reset_index(drop=True)             # back to RangeIndex
```

Common index types: `RangeIndex`, `Index` (generic), `DatetimeIndex`, `MultiIndex`.

> Tip: Give your index a meaning (e.g., an ID or date). It improves merging, slicing, and time-series operations.

D) Values as Arrays

`.values` (NumPy ndarray)

Returns the underlying array. Mixed dtypes → `object` array.

```
arr = df.values
```

`.to_numpy(dtype=..., copy=...)` (preferred)

```
arr = df.to_numpy()           # safe, respects extension dtypes
arr64 = df.select_dtypes("number").to_numpy(dtype="float64")
```

> Prefer `to_numpy()` for clarity and control over dtype/copy. `.values` can surprise you with `object` dtype when columns are mixed.

E) Quick Peek / Profiling

`.head(n)` / `.tail(n)` — first/last rows

```
df.head(3)
df.tail(2)
```

`.sample(n=..., frac=...)` — quick random check

```
df.sample(2, random_state=0)
```

`.info()` — schema summary, nulls, memory

```
df.info()
df.info(show_counts=True)         # show non-null counts explicitly
df.info(memory_usage="deep")      # more accurate memory sizes
```

`.describe()` — stats summary

```
df.describe()                     # numeric columns only
df.describe(include="all")        # all columns (object/category counts, top, freq)
df.describe(percentiles=[.05,.5,.95])
```

Fast missingness snapshot

```
df.isna().sum()                   # NA count per column
df.isna().mean()                  # NA ratio per column
```

Other handy peeks

```
df.nunique()                      # distinct values per column
df["dept"].value_counts(dropna=False)  # frequency table for a column
```

F) Other Handy Attributes

```
df.T              # transpose (swap rows/cols)
df.empty          # True if df has 0 elements (rows*cols == 0)
df.memory_usage(deep=True)      # per-column memory usage
df.columns.name = "fields"      # name the columns index
df.index.name = "row_id"        # name the row index
df.attrs["source"] = "HR export"  # attach arbitrary metadata
```

📝 Exercises (at least 10 — here are 14)

Assume `df` is the DataFrame defined at the top unless stated otherwise.

1. **Shapes & Sizes**
   - Print: number of rows, number of columns, total cells (three separate lines).
2. **Dimensionality Check**
   - Verify that `df` is 2D and that `df["age"]` is 1D using `.ndim`.
3. **Index Naming**
   - Set the index to the `name` column (lowercase), name the index `"employee"`, then reset it back.
4. **Column Cleanup**
   - Lowercase all column names and replace spaces with underscores.
5. **Dtype Audit**
   - Show a table of: column name, dtype, number of nulls, number of unique values.
6. **Select by Dtype**
   - Get a DataFrame containing only numeric columns. Then compute `.describe()` on it with percentiles at 5%, 50%, 95%.
7. **Memory Matters**
   - Show memory usage per column with `deep=True`.
   - Convert the `dept` column to `category`. Show memory usage again.
   - Report how many bytes you saved.
8. **Smart Type Conversion**

- Run `convert_dtypes()` and print the dtypes before vs after. Which columns changed and how?

9. **Missingness Snapshot**
   - Compute a two-line summary:
     - (a) NA counts per column
     - (b) NA ratios (0–1) per column, sorted descending.
10. **Describe Everything**
    - Run `df.describe(include="all")` and interpret: which column has the most frequent value shown by `top` / `freq`?
11. **Quick Peek Trio**
    - Show the first 2 rows, last 2 rows, and a random sample of 2 rows (fixed `random_state` for reproducibility).
12. **Values vs to_numpy**
    - Extract all numeric data as a NumPy array with dtype `float64` using **one** line of code (no intermediate variables).
    - Explain why `.values` could be risky on mixed-type DataFrames.
13. **Index Types**
    - Create `df2 = df.set_index("joined")`. What is the index type now? Verify using `type(df2.index)` and `.dtypes`.
14. **Profile Helper**
    - Build a small function `quick_profile(df)` that returns a DataFrame with columns: `["col","dtype","non_null","nulls","nunique","memory_bytes"]` sorted by `memory_bytes` descending.

✅ Solutions

1) Shapes & Sizes

```
rows, cols = df.shape
print(rows)
print(cols)
print(df.size)
```

2) Dimensionality Check

```
print(df.ndim)          # 2
print(df["age"].ndim)   # 1
```

3) Index Naming

```
df_ix = df.set_index("name").copy()
df_ix.index.name = "employee"
print(df_ix.index)      # check name
df_back = df_ix.reset_index()
```

4) Column Cleanup

```
df.columns = [c.lower().replace(" ", "_") for c in df.columns]
print(df.columns)
```

5) Dtype Audit

```
audit = pd.DataFrame({
    "col": df.columns,
    "dtype": df.dtypes.values,
    "nulls": df.isna().sum().values,
    "nunique": df.nunique(dropna=False).values
})
print(audit)
```

6) Select by Dtype

```
num_df = df.select_dtypes(include="number")
print(num_df.describe(percentiles=[.05,.5,.95]))
```

7) Memory Matters

```
before = df.memory_usage(deep=True)
df_cat = df.copy()
df_cat["dept"] = df_cat["dept"].astype("category")
after = df_cat.memory_usage(deep=True)
saved = int(before.sum() - after.sum())
print(before)
print(after)
print("Bytes saved:", saved)
```

8) Smart Type Conversion

```
print("Before:\n", df.dtypes)
df_conv = df.convert_dtypes()
print("After:\n", df_conv.dtypes)
# You'll likely see 'remote' → boolean, strings → string dtype
```

9) Missingness Snapshot

```
na_counts = df.isna().sum()
na_ratio = df.isna().mean().sort_values(ascending=False)
print(na_counts)
print(na_ratio)
```

10) Describe Everything

```
desc_all = df.describe(include="all")
print(desc_all)
# Look at rows 'top' and 'freq' for non-numeric columns (e.g., dept)
```

11) Quick Peek Trio

```
print(df.head(2))
print(df.tail(2))
print(df.sample(2, random_state=42))
```

12) Values vs to_numpy

```
arr = df.select_dtypes(include="number").to_numpy(dtype="float64")
# Why .values can be risky:
# On mixed dtypes, .values may become object dtype, losing numeric efficiency and semantics.
```

13) Index Types

```
df2 = df.set_index("joined")
print(type(df2.index))   # <class 'pandas.core.indexes.datetimes.DatetimeIndex'>
print(df2.dtypes)
```

14) Profile Helper

```
def quick_profile(x: pd.DataFrame) -> pd.DataFrame:
    return pd.DataFrame({
        "col": x.columns,
        "dtype": x.dtypes.astype(str).values,
        "non_null": x.notna().sum().values,
        "nulls": x.isna().sum().values,
        "nunique": x.nunique(dropna=False).values,
```

```
        "memory_bytes": x.memory_usage(deep=True, index=False).values
    }).sort_values("memory_bytes", ascending=False).reset_index(drop=True)

print(quick_profile(df))
```

```
Start coding or generate with AI.
```

4. Indexing & Selection in Pandas

Indexing & selection is how you **zoom in** on the rows and columns you need in a DataFrame. Think of it like controlling a giant Excel sheet with precise commands.

4.1 Column Selection

Two main ways:

```
import pandas as pd

data = {
    "Name": ["Ali", "Sara", "Omar", "Nora"],
    "Age": [25, 30, 22, 28],
    "City": ["Cairo", "Alexandria", "Giza", "Mansoura"]
}
df = pd.DataFrame(data)

# Bracket notation
print(df["Name"])   # Always works

# Dot notation
print(df.Age)       # Works if column has no spaces/special characters
```

👉 **Tips & Tricks:**

- Prefer `df["col"]` for reliability (dot notation may fail if col = "count" or "sum").
- You can select multiple columns at once: `df[["Name", "City"]]`.

4.2 Row Selection

Two main approaches:

```
# .loc -> label-based
print(df.loc[0])        # Row with label 0
print(df.loc[1:2])      # From label 1 to label 2 (inclusive!)

# .iloc -> position-based
print(df.iloc[0])       # Row at position 0
print(df.iloc[1:3])     # Position 1 and 2 (end-exclusive)
```

👉 **Tips & Tricks:**

- `.loc[a:b]` includes the last index, `.iloc[a:b]` excludes it.
- Combine row & column selection: `df.loc[0, "Name"]`.

4.3 Slicing DataFrames

You can slice rows like lists, and columns with `.loc`.

```
# Slice rows
print(df[1:3])   # Rows 1 and 2

# Select subset of columns
print(df.loc[:, ["Name", "City"]])
```

```
# Range of rows + specific columns
print(df.loc[1:3, ["Name", "Age"]])
```

👉 **Tip:** Use `:` to mean "all" (e.g., `df.loc[:, "Age"]` means all rows, only Age column).

4.4 Boolean Indexing & Filtering

This is where Pandas shines ✨.

```
# Rows where Age > 25
print(df[df["Age"] > 25])

# Multiple conditions
print(df[(df["Age"] > 25) & (df["City"] == "Cairo")])
```

👉 **Tips & Tricks:**

- Use `&` for AND, `|` for OR, and wrap conditions in parentheses.
- Use `.isin()` for matching multiple values: `df[df["City"].isin(["Cairo","Giza"])]`.
- Use `.str.contains()` for text filters: `df[df["City"].str.contains("Cai")]`.

4.5 Setting Index & Resetting Index

Sometimes a column is more meaningful as the index.

```
# Set index
df2 = df.set_index("Name")
print(df2)

# Select row by index
print(df2.loc["Omar"])

# Reset to default integer index
print(df2.reset_index())
```

👉 **Tips & Tricks:**

- Use `inplace=True` if you don't want a copy: `df.set_index("Name", inplace=True)`.
- After setting index, you can quickly access rows with `df.loc[index_value]`.

📝 Exercises

Using the same DataFrame `df`:

```
import pandas as pd

data = {
    "Name": ["Ali", "Sara", "Omar", "Nora", "Mona"],
    "Age": [25, 30, 22, 28, 35],
    "City": ["Cairo", "Alexandria", "Giza", "Mansoura", "Cairo"]
}
df = pd.DataFrame(data)
```

Tasks

1. Select the "**City**" column using both methods.
2. Select the **first two rows** using `.iloc`.
3. Select rows with labels **1 to 3** using `.loc`.
4. Get the **Name and Age** columns for the first three rows.
5. Select all rows where **Age > 25**.
6. Select all rows where **City is Cairo OR Giza**.

7. Select only the **Name** of people older than 28.
8. Set "Name" as the index and access Omar's row.
9. Reset the index back to default integers.
10. Slice the DataFrame to get rows **2 to 4** and only the "City" column.

✅ Solutions

```
# 1
print(df["City"])
print(df.City)

# 2
print(df.iloc[0:2])

# 3
print(df.loc[1:3])

# 4
print(df.loc[0:2, ["Name", "Age"]])

# 5
print(df[df["Age"] > 25])

# 6
print(df[df["City"].isin(["Cairo", "Giza"])])

# 7
print(df.loc[df["Age"] > 28, "Name"])

# 8
df2 = df.set_index("Name")
print(df2.loc["Omar"])

# 9
print(df2.reset_index())

# 10
print(df.loc[2:4, ["City"]])
```

```
Start coding or generate with AI.
```

5. Data Cleaning in Pandas

Real-world datasets are rarely perfect — they have **missing values, duplicates, wrong column names, and incorrect data types**. Pandas gives us tools to fix all that.

5.1 Handling Missing Values

Missing values often appear as `NaN`.

```
import pandas as pd
import numpy as np

data = {
    "Name": ["Ali", "Sara", "Omar", "Nora", "Mona"],
    "Age": [25, np.nan, 22, 28, np.nan],
    "City": ["Cairo", "Alexandria", None, "Mansoura", "Cairo"]
}
df = pd.DataFrame(data)

print(df.isna())       # Check missing values (True/False)
print(df.isna().sum()) # Count missing values per column
```

```
# Fill missing values
print(df.fillna("Unknown"))      # Fill NaN with "Unknown"
print(df["Age"].fillna(df["Age"].mean()))  # Replace NaN with mean Age

# Drop missing values
print(df.dropna())     # Drop rows with any NaN
print(df.dropna(subset=["Age"])) # Drop only if Age is missing
```

👉 **Tips & Tricks:**

- Use `.fillna(method="ffill")` (forward fill) or `.fillna(method="bfill")` (backward fill) for sequential data.
- `.dropna(axis=1)` removes **columns** with missing values.

5.2 Handling Duplicates

```
data2 = {
    "Name": ["Ali", "Sara", "Omar", "Sara"],
    "Age": [25, 30, 22, 30],
    "City": ["Cairo", "Alexandria", "Giza", "Alexandria"]
}
df2 = pd.DataFrame(data2)

print(df2.duplicated())        # True if row is duplicate
print(df2.drop_duplicates())   # Remove duplicates
```

👉 **Tips & Tricks:**

- Keep first/last occurrence: `drop_duplicates(keep="first")` or `keep="last"`.
- Drop based on certain columns:

  ```
  df2.drop_duplicates(subset=["Name"])
  ```

5.3 Renaming Columns

```
df3 = df.rename(columns={"Name": "FullName", "Age": "Years"})
print(df3)
```

👉 **Tips & Tricks:**

- Rename in place:

  ```
  df.rename(columns={"Name": "FullName"}, inplace=True)
  ```

- Use `df.columns = [list]` to rename all at once:

  ```
  df.columns = ["Name", "Age", "City"]
  ```

5.4 Changing Data Types

```
df["Age"] = df["Age"].astype("float")   # Convert to float
df["Age"] = df["Age"].astype("Int64")   # Nullable integer
```

👉 **Tips & Tricks:**

- Use `.astype(str)` to convert numbers to strings.
- Use `pd.to_datetime(df["col"])` for dates.
- Use `pd.to_numeric(df["col"], errors="coerce")` to force invalid numbers into NaN.

📝 Exercises

Using this DataFrame:

```
data = {
    "Name": ["Ali", "Sara", "Omar", "Nora", "Sara"],
    "Age": [25, None, 22, 28, 30],
    "City": ["Cairo", "Alexandria", None, "Mansoura", "Alexandria"]
}
df = pd.DataFrame(data)
```

Tasks

1. Check how many missing values are in each column.
2. Fill missing Age values with the average Age.
3. Fill missing City values with `"Unknown"`.
4. Drop rows where `"City"` is missing.
5. Identify which rows are duplicates.
6. Remove duplicate rows, keeping the first occurrence.
7. Remove duplicates based only on `"Name"`.
8. Rename `"Age"` to `"Years"`.
9. Change `"Years"` column to float type.
10. Convert `"City"` to string type.

✅ Solutions

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

# 2
df["Age"].fillna(df["Age"].mean(), inplace=True)

# 3
df["City"].fillna("Unknown", inplace=True)

# 4
print(df.dropna(subset=["City"]))

# 5
print(df.duplicated())

# 6
print(df.drop_duplicates())

# 7
print(df.drop_duplicates(subset=["Name"]))

# 8
df.rename(columns={"Age": "Years"}, inplace=True)

# 9
df["Years"] = df["Years"].astype(float)

# 10
df["City"] = df["City"].astype(str)
```

```
Start coding or generate with AI.
```

Perfect 😃 — let's turn **Data Cleaning** into something closer to how it actually happens in real-world datasets. I'll give you a couple of **mini case studies** that combine the techniques we just covered. That way you'll see how missing values, duplicates, renaming, and type conversions play together.

🏥 Case Study 1: Hospital Patient Records

Suppose you receive this dataset:

```
import pandas as pd
import numpy as np

data = {
    "PatientID": [101, 102, 103, 104, 104, 105],
    "Name": ["Ali", "Sara", "Omar", "Nora", "Nora", "Mona"],
    "Age": [25, np.nan, 60, None, None, 45],
    "AdmissionDate": ["2023-01-10", "2023-02-15", "not_recorded", "2023-04-01", "2023-04-01", "2023-05-20"],
    "City": ["Cairo", "Alex", None, "Cairo", "Cairo", "Mansoura"]
}
df = pd.DataFrame(data)
print(df)
```

Issues we see immediately:

- Missing Ages
- Invalid AdmissionDate (`"not_recorded"`)
- Duplicate rows (PatientID 104 repeated)
- City values inconsistent (`Alex` vs `Alexandria`)

Cleaning Steps

```
# 1. Handle missing values
df["Age"].fillna(df["Age"].mean(), inplace=True)   # Fill age with mean

# 2. Fix invalid dates
df["AdmissionDate"] = pd.to_datetime(df["AdmissionDate"], errors="coerce")

# 3. Handle missing cities
df["City"].fillna("Unknown", inplace=True)

# 4. Standardize city names
df["City"].replace({"Alex": "Alexandria"}, inplace=True)

# 5. Remove duplicates
df = df.drop_duplicates()

# 6. Set PatientID as index
df.set_index("PatientID", inplace=True)

print(df)
```

✅ **Result:** A clean dataset where

- All patients have an Age
- Dates are in proper `datetime` format
- City names are consistent
- No duplicate patients

🏢 Case Study 2: Employee Salary Records

Dataset:

```
data = {
    "EmpID": [1, 2, 3, 4, 4, 5],
    "Full Name": ["Khaled", "Sara", "Omar", "Nora", "Nora", "Mona"],
    "Salary": ["5000", "NaN", "7000", "8000", "8000", "ten thousand"],
```

```
    "JoinDate": ["2021/01/10", "2021/02/15", None, "2021/04/01", "2021/04/01", "2021/05/20"]
}
df = pd.DataFrame(data)
print(df)
```

Problems here:

- `"Salary"` column stored as strings
- `"NaN"` and `"ten thousand"` are not valid numbers
- Missing JoinDate
- Duplicate employee ID (4)
- Column name `"Full Name"` has a space

Cleaning Steps

```
# 1. Fix column names
df.rename(columns={"Full Name": "Name"}, inplace=True)

# 2. Convert Salary to numeric (invalid values → NaN)
df["Salary"] = pd.to_numeric(df["Salary"], errors="coerce")

# 3. Fill missing salaries with median
df["Salary"].fillna(df["Salary"].median(), inplace=True)

# 4. Convert JoinDate to datetime
df["JoinDate"] = pd.to_datetime(df["JoinDate"], errors="coerce")

# 5. Drop duplicates
df = df.drop_duplicates(subset=["EmpID"])

# 6. Set EmpID as index
df.set_index("EmpID", inplace=True)

print(df)
```

✅ **Result:**

- Clean `Salary` as numbers, with bad values handled
- Proper `datetime` for JoinDate
- No duplicate employees
- Column names standardized

🌟 What you should notice

- Real cleaning **always combines multiple steps**.
- You often have to decide: **fill vs drop** missing values (depends on use case).
- Text normalization (like `"Alex"` → `"Alexandria"`) is very common.
- Changing types (`str → float`, `str → datetime`) is essential before analysis.

👉 Now here's the fun part: do you want me to create a **bigger "challenge dataset"** (like a messy customer database with missing, wrong, and duplicate entries) and then **walk you through step-by-step cleaning it** as if you were preparing it for analysis? That would feel like a real project.

```
Start coding or generate with AI.
```

🛒 Challenge Dataset: Messy Customer Records

```
import pandas as pd

data = {
```

```
    "CustomerID": [1, 2, 2, 3, 4, 5, 6, 7, 8, 9],
    "Name": ["Ali", "Sara", "Sara", "Omar", "Nora", "Mona", None, "Khaled", "Sara", "Omar"],
    "Age": [25, "Thirty", None, 40, 29, 35, 22, None, 30, 40],
    "SignupDate": ["2021-01-10", "2021-02-30", "2021-02-15", "not_recorded", None, "2021-06-01", "2021/07/20", "2021-08-15", "202
    "Email": ["ali@example.com", "sara@", "sara@", "omar@example.com", "nora@example.com", "mona@example.com", "bad_email", None,
    "City": ["Cairo", "Alex", "Alex", None, "Cairo", "Mansoura", "Mansoura", "Cairo", "Alexandria", "Cairo"]
}
df = pd.DataFrame(data)
print(df)
```

🚨 Problems spotted:

- **Duplicates:** CustomerID 2 appears twice, Omar appears twice.
- **Name column:** Missing values.
- **Age column:** Mixed types (string `"Thirty"`, None, int).
- **SignupDate column:** Invalid dates (`"2021-02-30"`, `"not_recorded"`, None).
- **Email column:** Invalid emails (`"sara@"`, `"bad_email"`, None).
- **City column:** Inconsistent naming (`"Alex"` vs `"Alexandria"`), missing values.

🛠️ Step-by-Step Cleaning

1. Handle duplicates

```
df = df.drop_duplicates()
```

---

2. Fix missing names

```
df["Name"].fillna("Unknown", inplace=True)
```

---

3. Fix ages

```
# Convert to numeric, invalid → NaN
df["Age"] = pd.to_numeric(df["Age"], errors="coerce")

# Fill missing ages with median
df["Age"].fillna(df["Age"].median(), inplace=True)
```

---

4. Fix dates

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

# Fill missing/invalid dates with the earliest valid signup date
min_date = df["SignupDate"].min()
df["SignupDate"].fillna(min_date, inplace=True)
```

---

5. Clean emails

```
# Simple rule: keep only if contains '@' and '.'
df["Email"] = df["Email"].where(df["Email"].str.contains("@") & df["Email"].str.contains("\."), "invalid@example.com")
```

---

6. Normalize cities

```
df["City"].replace({"Alex": "Alexandria"}, inplace=True)
df["City"].fillna("Unknown", inplace=True)
```

---

7. Set a meaningful index

```
df.set_index("CustomerID", inplace=True)
```

✅ Final Cleaned DataFrame

After cleaning, `df` is now consistent, with:

- No duplicates
- Valid numbers for Age
- Valid dates in SignupDate
- Standardized City names
- Emails either valid or flagged as `"invalid@example.com"`

👉 That's a **realistic data cleaning workflow**. You've touched:

- Missing values
- Duplicates
- Type conversion
- String cleanup
- Normalization

```
Start coding or generate with AI.
```

6. Operations in Pandas

6.1 Arithmetic Operations

Pandas lets you do math directly on columns (like Excel formulas).

```
import pandas as pd

data = {
    "Name": ["Ali", "Sara", "Omar", "Nora"],
    "Math": [80, 90, 75, 60],
    "Science": [85, 95, 70, 65]
}
df = pd.DataFrame(data)

# Column-wise arithmetic
df["Total"] = df["Math"] + df["Science"]
df["Average"] = df["Total"] / 2

print(df)

# Broadcasting with scalars
df["Bonus"] = df["Math"] * 1.1
```

👉 **Tips & Tricks:**

- Operations are vectorized → much faster than loops.
- Use `.add(), .sub(), .mul(), .div()` for element-wise with `fill_value` in case of NaN.

6.2 String Operations (`.str`)

When dealing with text columns, use `.str` for transformations.

```
df2 = pd.DataFrame({"City": ["cairo", "Alexandria", "giza", "Mansoura"]})

print(df2["City"].str.upper())     # Convert to uppercase
```

```
print(df2["City"].str.lower())     # Convert to lowercase
print(df2["City"].str.len())       # Length of each string
print(df2["City"].str.contains("a"))  # Boolean mask
```

👉 **Tips & Tricks:**

- Use `.str.replace("old", "new")` for corrections.
- `.str.strip()` removes spaces; `.str.split(",")` splits strings.

6.3 Apply Functions (`apply`, `map`, `applymap`)

You can apply custom logic to rows, columns, or individual elements.

```
# Column-wise with apply
df["Grade"] = df["Average"].apply(lambda x: "Pass" if x >= 75 else "Fail")

# Element-wise with map (for Series)
df["NameLength"] = df["Name"].map(len)

# Element-wise with applymap (for entire DataFrame)
df_numeric = df[["Math", "Science"]]
print(df_numeric.applymap(lambda x: x + 5))  # Add 5 to all numbers
```

👉 **Tips & Tricks:**

- `applymap` → works on entire DataFrame, element by element.
- `apply` → works on Series (or DataFrame row/col).
- `map` → only for Series.

6.4 Sorting (`sort_values`, `sort_index`)

Sorting is essential for organizing data.

```
# Sort by column values
print(df.sort_values("Average", ascending=False))

# Sort by multiple columns
print(df.sort_values(["Average", "Math"], ascending=[False, True]))

# Sort by index
print(df.sort_index())
```

👉 **Tips & Tricks:**

- Use `ascending=False` for descending order.
- `na_position="first"` moves NaN to the top.

📝 Exercises

Using this dataset:

```
data = {
    "Name": ["Ali", "Sara", "Omar", "Nora", "Mona"],
    "Math": [80, 90, 75, 60, 85],
    "Science": [85, 95, 70, 65, 88],
    "City": ["cairo", "Alexandria", "giza", "Mansoura", "Cairo"]
}
df = pd.DataFrame(data)
```

Tasks

1. Create a new column `"Total"` = Math + Science.
2. Create a new column `"Average"` = Total / 2.

3. Multiply all `"Math"` scores by 1.1 (bonus).
4. Convert all `"City"` names to uppercase.
5. Count the number of characters in each `"City"`.
6. Create a new column `"Result"` → `"Pass"` if Average ≥ 80, else `"Fail"`.
7. Add a column `"NameLength"` containing the length of each name.
8. Add 10 to all numbers in both `"Math"` and `"Science"` columns.
9. Sort the DataFrame by `"Average"` in descending order.
10. Sort the DataFrame by `"City"` alphabetically.

✅ Solutions

```
# 1
df["Total"] = df["Math"] + df["Science"]

# 2
df["Average"] = df["Total"] / 2

# 3
df["MathBonus"] = df["Math"] * 1.1

# 4
df["CityUpper"] = df["City"].str.upper()

# 5
df["CityLength"] = df["City"].str.len()

# 6
df["Result"] = df["Average"].apply(lambda x: "Pass" if x >= 80 else "Fail")

# 7
df["NameLength"] = df["Name"].map(len)

# 8
df[["Math", "Science"]] = df[["Math", "Science"]].applymap(lambda x: x + 10)

# 9
print(df.sort_values("Average", ascending=False))

# 10
print(df.sort_values("City"))
```

```
Start coding or generate with AI.
```

💼 Case Study: Employee Performance Dataset

You receive this raw dataset from HR:

```
import pandas as pd

data = {
    "EmpID": [101, 102, 103, 104, 105],
    "Name": ["ali ", "SARA", "Omar", "nora", "Mona"],
    "Department": ["sales", "sales", "tech", "tech", "sales"],
    "BaseSalary": [5000, 6000, 7000, 6500, 6200],
    "Bonus%": [10, 15, 5, 12, 20],
    "Projects": [3, 5, 2, 4, 6]
}
df = pd.DataFrame(data)
print(df)
```

🚨 Problems to solve:

- `Name` column is inconsistent (extra spaces, mixed case).
- We need **total salary after bonus**.
- We want to **categorize employees** as `"High Performer"` or `"Regular"` based on projects.
- Need to **sort employees** by performance and salary.

🛠️ Cleaning & Operations

1. Fix Names (string ops)

```
df["Name"] = df["Name"].str.strip().str.title()
```

✅ Removes spaces and makes names consistent (`"ali "` → `"Ali"`, `"SARA"` → `"Sara"`).

---

2. Calculate Total Salary (arithmetic ops)

```
df["TotalSalary"] = df["BaseSalary"] + (df["BaseSalary"] * df["Bonus%"] / 100)
```

---

3. Categorize Employees (apply function)

```
df["Performance"] = df["Projects"].apply(lambda x: "High Performer" if x >= 4 else "Regular")
```

---

4. Sort by Performance then Salary

```
df_sorted = df.sort_values(["Performance", "TotalSalary"], ascending=[False, False])
print(df_sorted)
```

✅ Final Result (Clean + Analyzed)

EmpID	Name	Department	BaseSalary	Bonus%	Projects	TotalSalary	Performance
105	Mona	sales	6200	20	6	7440	High Performer
102	Sara	sales	6000	15	5	6900	High Performer
104	Nora	tech	6500	12	4	7280	High Performer
103	Omar	tech	7000	5	2	7350	Regular
101	Ali	sales	5000	10	3	5500	Regular

🌟 What you did here

- **String ops**: Cleaned inconsistent names.
- **Arithmetic ops**: Calculated salaries.
- **Apply functions**: Created performance categories.
- **Sorting**: Ranked employees by performance + pay.

This is exactly how operations come together in practice.

```
Start coding or generate with AI.
```

6. Operations in Pandas

Pandas allows you to perform a wide range of operations on DataFrames and Series. These include **arithmetic operations, string operations, applying functions, and sorting.**

---

1. Arithmetic Operations

Pandas supports **element-wise arithmetic operations** between DataFrames and Series.

```
import pandas as pd

df = pd.DataFrame({
    'A': [10, 20, 30],
    'B': [1, 2, 3]
})

print(df['A'] + df['B'])   # Addition
print(df['A'] - df['B'])   # Subtraction
print(df['A'] * df['B'])   # Multiplication
print(df['A'] / df['B'])   # Division
```

👉 **Tip:** Pandas automatically **aligns by index**. If the indices don't match, it fills missing values with NaN.

---

2. String Operations (`.str`)

String operations are applied to **Series of strings** using `.str`.

```
s = pd.Series(["Pandas", "is", "awesome", "python"])
print(s.str.upper())     # Convert to uppercase
print(s.str.contains("a"))  # Check if string contains 'a'
print(s.str.len())       # Length of each string
print(s.str.replace("python", "great"))
```

👉 **Tip:** You can chain multiple string operations together:

```
s.str.lower().str.replace("python", "pandas")
```

---

3. Apply Functions (`apply`, `map`, `applymap`)

- **apply()** → Works on **rows or columns** of a DataFrame.
- **map()** → Works on a **Series** (1D).
- **applymap()** → Works on **element-wise DataFrame**.

```
# Using apply on DataFrame
print(df.apply(sum))   # Sum of each column

# Using map on Series
print(df['A'].map(lambda x: x * 2))

# Using applymap on DataFrame
print(df.applymap(lambda x: x ** 2))
```

👉 **Tip:** Use **vectorized operations** whenever possible—they are faster than apply/map.

---

4. Sorting

Sort by column values

```
print(df.sort_values(by='A', ascending=False))
```

Sort by index

```
print(df.sort_index())
```

👉 **Tip:** Use `df.sort_values(by=['A', 'B'])` for **multi-column sorting**.

✅ Exercises (with Solutions)

Exercise 1

Create a DataFrame with columns `Math`, `Science`, `English`. Add 10 to all values in `Math`.

✅ **Solution:**

```
df = pd.DataFrame({'Math': [50, 60, 70], 'Science': [80, 90, 100], 'English': [65, 75, 85]})
df['Math'] = df['Math'] + 10
print(df)
```

Exercise 2

Check which strings contain the word `"data"` in a Series.

✅ **Solution:**

```
s = pd.Series(["data science", "machine learning", "deep learning"])
print(s.str.contains("data"))
```

Exercise 3

Convert all strings in a Series to title case.

✅ **Solution:**

```
s = pd.Series(["python is fun", "pandas tutorial"])
print(s.str.title())
```

Exercise 4

Square all numbers in a DataFrame using `applymap`.

✅ **Solution:**

```
df = pd.DataFrame({'X': [1, 2, 3], 'Y': [4, 5, 6]})
print(df.applymap(lambda x: x ** 2))
```

Exercise 5

Get the length of each string in a Series.

✅ **Solution:**

```
s = pd.Series(["apple", "banana", "cherry"])
print(s.str.len())
```

Exercise 6

Sort a DataFrame by column `Age` in descending order.

✅ **Solution:**

```
df = pd.DataFrame({'Name': ['Ali', 'Sara', 'John'], 'Age': [25, 30, 22]})
print(df.sort_values(by='Age', ascending=False))
```

Exercise 7

Apply a function that adds `100` to all values in column `Salary`.

✅ **Solution:**

```
df = pd.DataFrame({'Salary': [1000, 2000, 3000]})
df['Salary'] = df['Salary'].apply(lambda x: x + 100)
```

```
print(df)
```

Exercise 8

Convert all values in a Series to lowercase.

✅ **Solution:**

```
s = pd.Series(["HELLO", "WORLD", "PYTHON"])
print(s.str.lower())
```

Exercise 9

Check if each string in a Series starts with `"p"`.

✅ **Solution:**

```
s = pd.Series(["python", "Pandas", "Numpy"])
print(s.str.startswith("p"))
```

Exercise 10

Sort a DataFrame by multiple columns (`Score` ascending, `Age` descending).

✅ **Solution:**

```
df = pd.DataFrame({
    'Name': ['A', 'B', 'C'],
    'Score': [90, 80, 90],
    'Age': [20, 25, 22]
})
print(df.sort_values(by=['Score', 'Age'], ascending=[True, False]))
```

```
import pandas as pd
data = {
    "Department": ["IT", "IT", "HR", "HR", "Finance", "Finance"],
    "Employee": ["A", "B", "C", "D", "E", "F"],
    "Salary": [60000, 65000, 55000, 52000, 70000, 72000],
    "Bonus": [5000, 6000, 4000, 4500, 7000, 7500]
}

df = pd.DataFrame(data)
print(df)
```

```
  Department Employee  Salary  Bonus
0         IT        A   60000   5000
1         IT        B   65000   6000
2         HR        C   55000   4000
3         HR        D   52000   4500
4    Finance        E   70000   7000
5    Finance        F   72000   7500
```

```
grouped = df.groupby("Department")
print(grouped["Salary"].mean())  # average salary per department
```

```
Department
Finance    71000.0
HR         53500.0
IT         62500.0
Name: Salary, dtype: float64
```

```
pivot = df.pivot_table(
    values="Salary",
    index="Department",
    aggfunc="mean"
)
print(pivot)
```

```
             Salary
Department
Finance     71000.0
HR          53500.0
IT          62500.0
```

Start coding or generate with AI.

7. Grouping & Aggregation in Pandas

Grouping and aggregation are powerful ways to **summarize** and **analyze** data. They let you split data into groups, apply computations, and combine results efficiently.

7.1 GroupBy Basics

The main idea: **Split → Apply → Combine**

```
import pandas as pd

data = {
    "Department": ["IT", "IT", "HR", "HR", "Finance", "Finance"],
    "Employee": ["A", "B", "C", "D", "E", "F"],
    "Salary": [60000, 65000, 55000, 52000, 70000, 72000],
    "Bonus": [5000, 6000, 4000, 4500, 7000, 7500]
}

df = pd.DataFrame(data)
print(df)
```

👉 **Group by Department**:

```
grouped = df.groupby("Department")
print(grouped["Salary"].mean())  # average salary per department
```

7.2 Aggregate Functions

Aggregation is applying a **summary function** on groups. Common aggregate functions:

- `sum()` – total
- `mean()` – average
- `count()` – number of items
- `min(), max()` – smallest/largest
- `std(), var()` – standard deviation, variance
- `median()` – middle value

```
# Total salary per department
print(df.groupby("Department")["Salary"].sum())

# Multiple aggregations
print(df.groupby("Department")["Salary"].agg(["mean", "sum", "max"]))
```

👉 Using **different functions on different columns**:

```
print(df.groupby("Department").agg({
    "Salary": "mean",
    "Bonus": "sum"
}))
```

7.3 Pivot Tables

Pivot tables are like Excel pivot tables—summarize data by categories.

```
pivot = df.pivot_table(
    values="Salary",
    index="Department",
    aggfunc="mean"
)
print(pivot)
```

👉 With multiple values and functions:

```
pivot2 = df.pivot_table(
    values=["Salary", "Bonus"],
    index="Department",
    aggfunc={"Salary": "mean", "Bonus": "sum"}
)
print(pivot2)
```

💡 Tips & Tricks

1. **`as_index=False`** keeps grouped results as a DataFrame instead of setting group keys as index.

   ```
   df.groupby("Department", as_index=False).mean()
   ```

2. Use **`reset_index()`** after grouping to flatten the result.
3. **Pivot tables** are more flexible than groupby for multi-level summaries.
4. Use **`margins=True`** in `pivot_table` to get totals.

   ```
   df.pivot_table(values="Salary", index="Department", aggfunc="mean", margins=True)
   ```

5. For **faster computation on large datasets**, use `.agg()` with dictionaries instead of chaining multiple groupby operations.

📝 Exercises

Exercise 1

Group the employees by `Department` and calculate the **average bonus**.

Exercise 2

Find the **total salary** paid by each department.

Exercise 3

Count how many employees work in each department.

Exercise 4

Find the **maximum bonus** in each department.

Exercise 5

Group by `Department` and calculate both **mean Salary** and **sum Bonus**.

Exercise 6

Create a pivot table showing the **average Salary** per Department.

Exercise 7

Modify the pivot table to show both **average Salary** and **sum of Bonus** per Department.

Exercise 8

Add `margins=True` to the pivot table and observe the total row.

Exercise 9

Use `.agg()` to calculate **min Salary** and **max Bonus** per Department.

Exercise 10

Group by Department and return results as a normal DataFrame (not index-based).

✅ Solutions

```
# 1. Average bonus
print(df.groupby("Department")["Bonus"].mean())

# 2. Total salary
print(df.groupby("Department")["Salary"].sum())

# 3. Employee count
print(df.groupby("Department")["Employee"].count())

# 4. Max bonus
print(df.groupby("Department")["Bonus"].max())

# 5. Mean Salary + Sum Bonus
print(df.groupby("Department").agg({"Salary": "mean", "Bonus": "sum"}))

# 6. Pivot avg salary
print(df.pivot_table(values="Salary", index="Department", aggfunc="mean"))

# 7. Pivot salary+bonus
print(df.pivot_table(values=["Salary", "Bonus"], index="Department",
                     aggfunc={"Salary": "mean", "Bonus": "sum"}))

# 8. Pivot with margins
print(df.pivot_table(values="Salary", index="Department",
                     aggfunc="mean", margins=True))

# 9. Min salary & Max bonus
print(df.groupby("Department").agg({"Salary": "min", "Bonus": "max"}))

# 10. Keep DataFrame format
print(df.groupby("Department", as_index=False).mean())
```

```
Start coding or generate with AI.
```

8. Merging, Joining & Concatenation

1. Concatenation (`pd.concat`)

- Used to **stack** DataFrames vertically (row-wise) or horizontally (column-wise).
- Think of it as “gluing” datasets together.

```
import pandas as pd

df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']})
df2 = pd.DataFrame({'A': ['A3', 'A4', 'A5'], 'B': ['B3', 'B4', 'B5']})

# Vertical concatenation (default: axis=0 → rows)
result = pd.concat([df1, df2])

# Horizontal concatenation (axis=1 → columns)
result_h = pd.concat([df1, df2], axis=1)
```

---

2. Merge (`pd.merge`)

- Similar to **SQL joins**.
- Combines DataFrames on **common columns** or **indices**.

```
left = pd.DataFrame({'id': [1, 2, 3], 'name': ['Alice', 'Bob', 'Charlie']})
right = pd.DataFrame({'id': [1, 2, 4], 'score': [85, 90, 75]})

# Inner Join (default)
pd.merge(left, right, on='id', how='inner')

# Left Join
pd.merge(left, right, on='id', how='left')

# Right Join
pd.merge(left, right, on='id', how='right')

# Outer Join
pd.merge(left, right, on='id', how='outer')
```

---

3. Join (`DataFrame.join`)

- A shortcut for joining on **index**.
- Works best when combining a main DataFrame with a **lookup table**.

```
left = pd.DataFrame({'name': ['Alice', 'Bob']}, index=[1, 2])
right = pd.DataFrame({'score': [85, 90]}, index=[1, 2])

# Joins on index
left.join(right)
```

---

4. Differences Between concat, merge, join

Function	Use Case
concat	Just stack/append along rows or columns
merge	SQL-like joins using keys (columns)
join	Convenient method to join on **index**

🔑 Tips & Tricks

- Use `ignore_index=True` with `concat` to reindex automatically.
- Use `indicator=True` in `merge` to see join origin (`left_only`, `right_only`, `both`).
- Rename overlapping columns with `suffixes=('_left', '_right')`.
- Always check for duplicate keys before merging to avoid data explosion.

📝 Exercises

Exercise 1

Concatenate the following two DataFrames row-wise and column-wise:

```
df1 = pd.DataFrame({'X': [1, 2], 'Y': [3, 4]})
df2 = pd.DataFrame({'X': [5, 6], 'Y': [7, 8]})
```

Exercise 2

Perform an **inner join** between:

```
df1 = pd.DataFrame({'id': [1, 2, 3], 'value': ['A', 'B', 'C']})
df2 = pd.DataFrame({'id': [2, 3, 4], 'score': [10, 20, 30]})
```

Exercise 3

Do a **left join** on the same DataFrames.

Exercise 4

Perform an **outer join** on the same DataFrames.

Exercise 5

Join two DataFrames on their **index**:

```
df1 = pd.DataFrame({'A': ['foo', 'bar']}, index=[1, 2])
df2 = pd.DataFrame({'B': ['baz', 'qux']}, index=[1, 2])
```

Exercise 6

Concatenate three small DataFrames (`df1`, `df2`, `df3`) along **rows**.

Exercise 7

Merge with different column names:

```
df1 = pd.DataFrame({'emp_id': [1, 2], 'name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'id': [1, 2], 'salary': [1000, 1500]})
```

(Hint: use `left_on` and `right_on`.)

Exercise 8

Perform a merge with `indicator=True` and explain what the output column means.

Exercise 9

Use `concat` with `keys=['df1', 'df2']` to create a hierarchical index.

Exercise 10

Create two DataFrames with duplicate keys and show how merging causes **data duplication**.

8. Merging, Joining & Concatenation in Pandas

Working with multiple datasets is one of the most common tasks in data analysis. Pandas provides **powerful tools** to combine DataFrames in different ways, similar to SQL.

📌 1. Concatenation (`pd.concat`)

- **Definition**: Combines multiple DataFrames **vertically** (stacking rows) or **horizontally** (stacking columns).
- **Syntax**:

```
pd.concat([df1, df2], axis=0)   # Vertical (default)
pd.concat([df1, df2], axis=1)   # Horizontal
```

- **Options**:
  - `ignore_index=True` → resets index.
  - `keys` → adds hierarchical index.

✅ **Tip**: Use `axis=0` for more rows, `axis=1` for more columns.

📌 2. Merge (`pd.merge`)

- **Definition**: Combines DataFrames based on one or more **common columns or indices**.
- **Syntax**:

```
pd.merge(df1, df2, on="key")
```

- **Join types (like SQL)**:
  - `how="inner"` (default): Keep only matching rows.
  - `how="left"`: Keep all rows from left, add matching from right.
  - `how="right"`: Keep all rows from right, add matching from left.
  - `how="outer"`: Keep all rows from both.

✅ **Tip**: Always check for **duplicate column names** — Pandas will add suffixes like `_x`, `_y`.

📌 3. Join (`df.join`)

- **Definition**: A shortcut for merging, often using **index** instead of columns.
- **Syntax**:

```
df1.join(df2, how="left")
```

- Works well when you want to combine DataFrames **by index**.

✅ **Tip**: Use `.set_index()` first if you want to join on a specific column.

📊 Example: SQL-style joins

```
import pandas as pd

employees = pd.DataFrame({
    "id": [1, 2, 3, 4],
    "name": ["Alice", "Bob", "Charlie", "David"]
})

salaries = pd.DataFrame({
    "id": [1, 2, 4, 5],
    "salary": [50000, 60000, 70000, 80000]
})

# INNER JOIN
print(pd.merge(employees, salaries, on="id", how="inner"))

# LEFT JOIN
print(pd.merge(employees, salaries, on="id", how="left"))

# OUTER JOIN
print(pd.merge(employees, salaries, on="id", how="outer"))
```

💡 Tips & Tricks

- Use `indicator=True` in `merge` → shows which rows came from which DataFrame.
- Use `concat` with `keys` to distinguish data sources.
- Use `suffixes=('_left', '_right')` to avoid confusion in column names after merge.
- For performance, **set indexes** before merging on large DataFrames.

📝 Exercises

Exercise 1: Concatenation of rows

Combine two DataFrames vertically.

```
df1 = pd.DataFrame({"A": [1, 2], "B": [3, 4]})
df2 = pd.DataFrame({"A": [5, 6], "B": [7, 8]})
```

✅ **Solution:**

```
pd.concat([df1, df2], ignore_index=True)
```

Exercise 2: Concatenation of columns

Stack `df1` and `df2` side by side.

✅ **Solution:**

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

Exercise 3: Inner merge

Merge employees and salaries only where IDs match.

✅ **Solution:**

```
pd.merge(employees, salaries, on="id", how="inner")
```

Exercise 4: Left merge

Keep all employees, even if they don't have salaries.

✅ **Solution:**

```
pd.merge(employees, salaries, on="id", how="left")
```

Exercise 5: Outer merge

Show all employees and all salaries (IDs may not match).

✅ **Solution:**

```
pd.merge(employees, salaries, on="id", how="outer")
```

Exercise 6: Merge with different column names

```
df1 = pd.DataFrame({"emp_id": [1,2,3], "name": ["A","B","C"]})
df2 = pd.DataFrame({"id": [1,2,4], "dept": ["HR","IT","Sales"]})
```

✅ **Solution:**

```
pd.merge(df1, df2, left_on="emp_id", right_on="id", how="inner")
```

Exercise 7: Join using index

```
df1 = pd.DataFrame({"A": [10,20,30]}, index=["x","y","z"])
df2 = pd.DataFrame({"B": [40,50,60]}, index=["x","y","w"])
```

✅ **Solution:**

```
df1.join(df2, how="outer")
```

Exercise 8: Concatenation with keys

Combine `df1` and `df2` with a hierarchical index.

✅ **Solution:**

```
pd.concat([df1, df2], keys=["First", "Second"])
```

Exercise 9: Merge with indicator

See which rows matched or not.

✅ **Solution:**

```
pd.merge(employees, salaries, on="id", how="outer", indicator=True)
```

Exercise 10: Sorting after merge

Merge employees and salaries, then sort by salary descending.

✅ **Solution:**

```
merged = pd.merge(employees, salaries, on="id", how="inner")
merged.sort_values("salary", ascending=False)
```

```
Start coding or generate with AI.
```

9. Working with Dates & Time in Pandas

Pandas makes it very easy to work with dates, times, and time-series data. It provides flexible tools for parsing, manipulating, and analyzing temporal data.

---

1. Converting to DateTime

Most datasets have dates stored as **strings** (e.g., `"2024-08-25"`) or numbers. We can convert them into Pandas `datetime` objects for powerful operations.

```
import pandas as pd

# Example
df = pd.DataFrame({
    "date_str": ["2023-01-01", "2023-03-15", "2023-07-20"],
    "sales": [200, 340, 560]
})

# Convert string to datetime
df["date"] = pd.to_datetime(df["date_str"])
print(df)
```

👉 `pd.to_datetime` automatically detects formats, but you can also specify `format="%Y-%m-%d"` for efficiency.

---

2. Extracting Date/Time Components

Once you have a `datetime` column, you can extract components:

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

👉 This is super useful for grouping (e.g., sales by month, transactions by weekday).

---

3. Creating Date Ranges

Generate sequences of dates easily:

```
dates = pd.date_range(start="2023-01-01", end="2023-01-10", freq="D")
print(dates)
```

Common frequencies:

- `"D"` → daily
- `"W"` → weekly
- `"M"` → month end
- `"Q"` → quarter end
- `"Y"` → year end
- `"H"` → hourly

---

4. Setting Date as Index

Time series analysis often requires setting a datetime column as the index.

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

This enables **time-based indexing**:

```
df["2023-01"]     # All rows from January 2023
df["2023-03-15"]  # Specific date
df["2023":"2023-06"]  # Slice by range
```

---

5. Resampling

Resampling means changing the frequency of time series data:

- **Downsampling**: daily → monthly
- **Upsampling**: monthly → daily

```
# Example: Resample monthly and take sum of sales
monthly_sales = df["sales"].resample("M").sum()
print(monthly_sales)
```

Common resampling methods:

- `.sum()` → total
- `.mean()` → average
- `.count()` → number of entries

---

6. Shifting & Lagging

Shift values forward/backward for lag analysis:

```
df["prev_day_sales"] = df["sales"].shift(1)
df["sales_change"] = df["sales"] - df["prev_day_sales"]
```

👉 Useful in finance, forecasting, or trend detection.

---

7. Rolling Windows

Rolling aggregates (moving average, rolling sum, etc.):

```
df["7d_avg"] = df["sales"].rolling(window=7).mean()
```

Tips & Tricks

1. Always use `pd.to_datetime()` before working with dates.
2. If your dataset is huge, **specify the format** in `to_datetime` to speed things up.
3. Use `.dt` accessor to extract date/time components.
4. For time-based grouping, resampling is often easier than `groupby`.
5. Use `.asfreq("D")` to reindex with specific frequency and fill missing dates.
6. Combine `.shift()` + `.rolling()` for advanced time-series features.

---

10 Exercises (with Solutions)

```
import pandas as pd
import numpy as np

# Create sample dataset
dates = pd.date_range(start="2023-01-01", periods=10, freq="D")
df = pd.DataFrame({
    "date": dates,
    "sales": [200, 220, 250, 180, 300, 270, 290, 310, 400, 420]
})
```

---

1. Convert "date" column to datetime

✅ Solution:

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

---

2. Extract year, month, and weekday

✅ Solution:

```
df["year"] = df["date"].dt.year
df["month"] = df["date"].dt.month
df["weekday"] = df["date"].dt.day_name()
```

---

3. Set "date" as index

✅ Solution:

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

---

4. Select sales for `"2023-01-05"`

✅ Solution:

```
df.loc["2023-01-05", "sales"]
```

---

5. Select sales between `"2023-01-03"` and `"2023-01-07"`

✅ Solution:

```
df.loc["2023-01-03":"2023-01-07", "sales"]
```

---

6. Resample to weekly sales (sum)

✅ Solution:

```
df["sales"].resample("W").sum()
```

---

7. Calculate daily difference in sales

✅ Solution:

```
df["daily_change"] = df["sales"].diff()
```

---

8. Create a 3-day rolling average

✅ Solution:

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

---

9. Shift sales by 2 days (lag feature)

✅ Solution:

```
df["lag_2"] = df["sales"].shift(2)
```

---

10. Fill missing values in resampled daily data

✅ Solution:

```
df_daily = df["sales"].resample("D").asfreq()
df_daily_filled = df_daily.fillna(method="ffill")  # forward fill
```

```
Start coding or generate with AI.
```

10. Input/Output with Files

Pandas makes it super easy to **read and write** data in different formats.

📥 Reading Data

1. `pd.read_csv()` → Read CSV file

```
import pandas as pd

# Reading CSV file
df = pd.read_csv("data.csv")
print(df.head())
```

Illustration: If `data.csv` looks like this:

```
name,age,city
Ali,25,Cairo
Sara,30,Alexandria
```

Output:

```
    name  age       city
0    Ali   25      Cairo
1   Sara   30  Alexandria
```

---

2. `pd.read_excel()` → Read Excel file

```
df = pd.read_excel("data.xlsx", sheet_name="Sheet1")
print(df.head())
```

Reads directly from Excel sheets 📊.

---

3. `pd.read_json()` → Read JSON file

```
df = pd.read_json("data.json")
print(df)
```

If `data.json` looks like:

```
[
  {"name": "Ali", "age": 25},
  {"name": "Sara", "age": 30}
]
```

Output:

```
    name  age
0    Ali   25
1   Sara   30
```

---

4. `pd.read_sql()` → Read from SQL Database

```
import sqlite3

# Connect to database
conn = sqlite3.connect("mydb.db")

# Read SQL query into DataFrame
df = pd.read_sql("SELECT * FROM employees", conn)
print(df.head())
```

Works like pulling data from a database straight into Pandas. 🗄️

📤 Writing Data

1. `to_csv()` → Save to CSV

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

💡 `index=False` avoids saving the row numbers.

---

2. `to_excel()` → Save to Excel

```
df.to_excel("output.xlsx", index=False, sheet_name="Sheet1")
```

Great for exporting clean Excel reports. ✅

---

3. `to_json()` → Save to JSON

```
df.to_json("output.json", orient="records", lines=True)
```

💡 `orient="records"` makes JSON more human-readable.

💡 Tips & Tricks

- Use `nrows` in `pd.read_csv()` to read only part of a large file:

```
df = pd.read_csv("data.csv", nrows=100)
```

- Use `chunksize` to read huge files in pieces:

```
for chunk in pd.read_csv("bigfile.csv", chunksize=5000):
    print(chunk.shape)
```

- For Excel, install **openpyxl** for `.xlsx` files.

📝 Exercises

1. Read a CSV file named `students.csv` into a DataFrame.
2. Read only the first 50 rows of a CSV file.
3. Read the `Sheet2` from an Excel file.
4. Read a JSON file named `products.json`.
5. Read data from a SQL table named `orders`.
6. Save a DataFrame `df` to `result.csv` without the index column.
7. Save a DataFrame to Excel with sheet name `Report`.
8. Save a DataFrame to JSON in `records` orientation.
9. Load a huge CSV file in chunks of 10,000 rows.
10. Practice writing the same DataFrame to all 3 formats: CSV, Excel, JSON.

```
Start coding or generate with AI.
```

10. Input/Output with Files in Pandas

Pandas makes it super easy to **read from** and **write to** different file formats like CSV, Excel, JSON, SQL databases, and more.

📥 Reading Files

1. `pd.read_csv()` – Read CSV files

CSV (Comma-Separated Values) is the most common format.

```
import pandas as pd

# Reading CSV
df = pd.read_csv("data.csv")
print(df.head())
```

🔹 **Tips:**

- Use `sep=";"` if your file uses semicolons.
- Use `nrows=10` to read only the first 10 rows.
- Use `usecols=['col1','col2']` to read selected columns.

---

2. `pd.read_excel()` – Read Excel files

```
# Reading Excel (requires openpyxl installed)
df = pd.read_excel("data.xlsx", sheet_name="Sheet1")
print(df.head())
```

🔹 **Tips:**

- `sheet_name=None` → loads all sheets into a dictionary of DataFrames.
- `usecols="A:C"` → selects columns A to C.

---

3. `pd.read_json()` – Read JSON files

```
# Reading JSON
df = pd.read_json("data.json")
print(df.head())
```

🔹 **Tips:**

- JSON is often used in APIs.
- If the JSON is nested, use `json_normalize` to flatten.

---

4. `pd.read_sql()` – Read from SQL Databases

```
import sqlite3

# Connect to database
conn = sqlite3.connect("mydata.db")

# Read SQL query into DataFrame
df = pd.read_sql("SELECT * FROM employees", conn)
print(df.head())
```

📤 Writing Files

1. `to_csv()` – Save DataFrame to CSV

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

🔹 **Tips:**

- `index=False` removes the row index column.
- `sep=";"` → saves with semicolon instead of comma.

---

2. `to_excel()` – Save DataFrame to Excel

```
df.to_excel("output.xlsx", sheet_name="Results", index=False)
```

---

3. `to_json()` – Save DataFrame to JSON

```
df.to_json("output.json", orient="records", lines=True)
```

🔹 **Tips:**

- `orient="records"` → saves each row as a dictionary.
- `lines=True` → saves in newline-delimited JSON (good for big files).

💡 Extra Tricks

- Use `chunksize=1000` in `read_csv` to read large files in small pieces.
- Use compression:

  ```
  df.to_csv("output.csv.gz", index=False, compression="gzip")
  ```

- For very large data, consider **Parquet** format (`read_parquet` / `to_parquet`) for faster performance.

📝 Exercises

Exercise 1:

Read a CSV file `students.csv` and display only the first 5 rows.

Exercise 2:

Read an Excel file `grades.xlsx` from sheet `Math` and display only the columns `Name` and `Score`.

Exercise 3:

Save a DataFrame `df` into a CSV file without the index column.

Exercise 4:

Convert a DataFrame `df` into JSON format with `orient="records"`.

Exercise 5:

Connect to a SQLite database `school.db`, read the table `teachers`, and print the first 3 rows.

✅ Solutions

```
# Exercise 1
df = pd.read_csv("students.csv")
print(df.head())

# Exercise 2
df = pd.read_excel("grades.xlsx", sheet_name="Math", usecols=["Name", "Score"])
print(df.head())

# Exercise 3
df.to_csv("students_output.csv", index=False)

# Exercise 4
df.to_json("students.json", orient="records")

# Exercise 5
import sqlite3
conn = sqlite3.connect("school.db")
df = pd.read_sql("SELECT * FROM teachers", conn)
print(df.head(3))
```

```
Start coding or generate with AI.
```

11. Advanced Topics in Pandas

---

1. MultiIndex (Hierarchical Indexing)

👉 A **MultiIndex** allows you to have multiple levels of indexing (rows or columns). It's useful for handling higher-dimensional data in a 2D table.

```
import pandas as pd

# Create a MultiIndex DataFrame
arrays = [
    ['A', 'A', 'B', 'B'],
    ['one', 'two', 'one', 'two']
]
index = pd.MultiIndex.from_arrays(arrays, names=('letter', 'number'))

df = pd.DataFrame({'value': [10, 20, 30, 40]}, index=index)
print(df)
```

Output:

```
             value
letter number
```

```
A      one       10
       two       20
B      one       30
       two       40
```

🔷 Accessing data with MultiIndex:

```
df.loc['A']          # All rows under 'A'
df.loc[('B', 'two')]  # Specific cell
```

💡 **Tip**: Use `.xs()` (cross section) to slice one level:

```
df.xs('one', level='number')
```

---

2. Window Functions (Rolling, Expanding)

👉 Useful for **time-series analysis** (like moving averages).

```
import numpy as np

s = pd.Series([1, 2, 3, 4, 5])

# Rolling window (moving average)
print(s.rolling(window=3).mean())

# Expanding window (cumulative mean)
print(s.expanding().mean())
```

Output (rolling mean):

```
0    NaN
1    NaN
2    2.0
3    3.0
4    4.0
dtype: float64
```

💡 **Tip**: Combine with `.plot()` to visualize trends smoothly.

---

3. Categorical Data

👉 Converting columns into **categories** saves memory and speeds up operations.

```
df = pd.DataFrame({'fruit': ['apple', 'banana', 'apple', 'orange', 'banana']})

# Convert to categorical
df['fruit'] = df['fruit'].astype('category')

print(df['fruit'].cat.categories)
print(df['fruit'].cat.codes)
```

Output:

```
Index(['apple', 'banana', 'orange'], dtype='object')
0    0
1    1
2    0
3    2
4    1
dtype: int8
```

💡 **Tip**: Use categorical for columns with **few unique values** (like gender, country codes).

---

4. Sparse Data

👉 Sparse data structures help save memory when you have **lots of zeros** or missing values.

```
arr = pd.Series([0, 0, 0, 1, 0, 0, 2])
sparse_arr = pd.arrays.SparseArray(arr)
print(sparse_arr)
```

Output:

```
[0, 0, 0, 1, 0, 0, 2]
Fill: 0
IntIndex
Indices: array([3, 6])
```

💡 **Tip**: Store large matrices with many zeros using sparse arrays.

---

5. Performance Tips

✅ **Vectorization** (avoid loops!):

```
# Slow
df['new'] = [x * 2 for x in df['value']]

# Fast (vectorized)
df['new'] = df['value'] * 2
```

✅ **.eval() & .query()** for faster calculations:

```
df = pd.DataFrame({'a': range(100000), 'b': range(100000)})

# Faster than df['a'] + df['b']
df.eval('c = a + b', inplace=True)

# Filtering
result = df.query('a < 5 & b > 2')
print(result)
```

✨ Exercises (with Solutions)

Q1. Create a MultiIndex DataFrame for regions (Asia, Europe) and countries, then assign random population values.

Q2. Select all data for Europe only using `.loc`.

Q3. Using `.xs()`, select all countries named "India".

Q4. Create a Series of numbers and compute a **7-day rolling mean**.

Q5. Compute the expanding mean for the same series.

Q6. Convert a column `["yes", "no", "yes", "maybe"]` to categorical and print codes.

Q7. Create a Series with `[0, 0, 0, 5, 0, 0, 10]` as a sparse array.

Q8. Show the memory difference between categorical and string columns.

Q9. Use `.eval()` to compute `total = col1 + col2` on a DataFrame with 2 numeric columns.

Q10. Use `.query()` to filter a DataFrame where `age > 30 and salary < 5000`.

✅ Solutions

```
import pandas as pd
import numpy as np

# Q1
arrays = [['Asia','Asia','Europe','Europe'],['India','China','Germany','France']]
index = pd.MultiIndex.from_arrays(arrays, names=('Continent','Country'))
df = pd.DataFrame({'Population': [1400, 1300, 80, 65]}, index=index)

# Q2
print(df.loc['Europe'])

# Q3
print(df.xs('India', level='Country'))

# Q4
s = pd.Series(np.arange(1,15))
print(s.rolling(window=7).mean())

# Q5
print(s.expanding().mean())

# Q6
c = pd.Series(["yes","no","yes","maybe"], dtype="category")
print(c.cat.codes)

# Q7
sparse_series = pd.arrays.SparseArray([0,0,0,5,0,0,10])
print(sparse_series)

# Q8
df2 = pd.DataFrame({"Answer":["yes","no","yes","maybe"]*1000})
print(df2.memory_usage(deep=True))
df2['Answer'] = df2['Answer'].astype('category')
print(df2.memory_usage(deep=True))

# Q9
df3 = pd.DataFrame({"col1": [1,2,3], "col2": [4,5,6]})
df3.eval("total = col1 + col2", inplace=True)
print(df3)

# Q10
df4 = pd.DataFrame({"age":[25,35,40], "salary":[4000,6000,3000]})
print(df4.query("age > 30 and salary < 5000"))
```

```
Start coding or generate with AI.
```

Project 1: Titanic Dataset Analysis

Problem Statement

Analyze the Titanic dataset to uncover insights about survival patterns.

Tasks:

1. Download and load the dataset.
2. Clean missing values in `"Age"` and `"Embarked"`.
3. Compute survival rates by `"Sex"` and `"Pclass"`.
4. Create a pivot table showing survival rate by `"Pclass"` and `"Sex"`.
5. Visualize survival rates using a bar chart.

Download the Data

You can download the CSV directly:

```
https://calmcode.io/static/data/titanic.csv
```

([CalmCode][1])

Solution Code

```
import pandas as pd
import matplotlib.pyplot as plt

# 1. Load dataset
url = "https://calmcode.io/static/data/titanic.csv"
df = pd.read_csv(url)

# 2. Clean missing values
df["Age"].fillna(df["Age"].median(), inplace=True)
df["Embarked"].fillna(df["Embarked"].mode()[0], inplace=True)

# 3. Survival rates by Sex and Pclass
survival_by_sex = df.groupby("Sex")["Survived"].mean()
survival_by_pclass = df.groupby("Pclass")["Survived"].mean()

# 4. Pivot table
pivot = df.pivot_table(
    values="Survived",
    index="Pclass",
    columns="Sex",
    aggfunc="mean"
)

# 5. Visualization
pivot.plot(kind="bar", figsize=(8,6))
plt.title("Survival Rate by Class and Sex")
plt.ylabel("Survival Rate")
plt.xticks(rotation=0)
plt.legend(title="Sex")
plt.show()

# Display results
print("Survival Rate by Sex:\n", survival_by_sex)
print("\nSurvival Rate by Pclass:\n", survival_by_pclass)
print("\nPivot Table:\n", pivot)
```

Project 2: Retail Sales Dataset Analysis

Problem Statement

Examine a retail sales dataset to understand trends and product performance.

Tasks:

1. Download and load the dataset.
2. Convert `"Date"` to datetime and clean duplicates.
3. Compute a new `"Total"` column (`Quantity * Price`).
4. Group sales by `"Region"` and `"Product Category"` to find total and average sales.
5. Identify the overall best-selling product.
6. Plot total sales by region.

Download the Data

Use this sample dataset (1,000 rows, retail sales details):

```
Sample Retail Sales Dataset (1000 rows, 10 columns including date, product category, quantity, price, region)
```

([Gigasheet][2]) *(You'll need to copy the data manually or use your preferred CSV source.)*

Solution Code

```
import pandas as pd
import matplotlib.pyplot as plt

# 1. Load dataset
# Example path:
# df = pd.read_csv("retail-sales.csv")
# For demonstration, we'll create a simulated dataset:

import numpy as np
date_range = pd.date_range(start="2025-01-01", periods=100, freq="D")
np.random.seed(0)
df = pd.DataFrame({
    "Date": np.random.choice(date_range, 1000),
    "Product Category": np.random.choice(["Electronics", "Clothing", "Groceries"], 1000),
    "Quantity": np.random.randint(1, 5, 1000),
    "Price": np.random.uniform(5.0, 100.0, 1000),
    "Region": np.random.choice(["North", "South", "East", "West"], 1000)
})

# 2. Clean data
df["Date"] = pd.to_datetime(df["Date"])
df.drop_duplicates(inplace=True)

# 3. Compute Total
df["Total"] = df["Quantity"] * df["Price"]

# 4. Group by Region & Product Category
grouped = df.groupby(["Region", "Product Category"]).agg(
    Total_Sales=("Total", "sum"),
    Avg_Sales=("Total", "mean"),
    Count=("Total", "count")
).reset_index()

# 5. Best-selling product
best_product = grouped.loc[grouped["Total_Sales"].idxmax()]

# 6. Plot total sales by region
region_sales = df.groupby("Region")["Total"].sum()
region_sales.plot(kind="bar", figsize=(8,6))
plt.title("Total Sales by Region")
plt.ylabel("Total Sales")
plt.xticks(rotation=0)
plt.show()

# Display results
print("Sales by Region & Product:\n", grouped.head())
print("\nBest-selling product entry:\n", best_product)
```

```
Start coding or generate with AI.
```

Project 3: Movies Dataset – Ratings Analysis

Problem Statement: The Movies dataset contains movie titles, genres, ratings, and release years. Your task is to clean the dataset, find the most popular genres, and analyze average ratings per year.

Dataset: IMDB Movies Dataset

Steps & Solution:

```
# Load dataset
df = pd.read_csv("imdb_top_1000.csv")

# Most common genres
genres = df['Genre'].str.split(',').explode().value_counts().head(5)
print("Top 5 Genres:\n", genres)

# Average rating per year
avg_rating_year = df.groupby('Year')['IMDB Rating'].mean()
print("Average Rating by Year:\n", avg_rating_year.head())

# Highest rated movie per genre
best_per_genre = df.groupby('Genre')['IMDB Rating'].max()
print("Best Movies per Genre:\n", best_per_genre)
```

Project 4: Weather Data Analysis

Problem Statement: You have a weather dataset containing temperature, humidity, and wind speed recorded daily. Your task is to analyze temperature trends, detect missing values, and compute monthly averages.

Dataset: Daily Weather Dataset

Steps & Solution:

```
# Load dataset
df = pd.read_csv("DailyDelhiClimateTrain.csv")

# Convert date column
df['date'] = pd.to_datetime(df['date'])

# Missing values
print("Missing Values:\n", df.isna().sum())

# Monthly average temperature
monthly_temp = df.resample('M', on='date')['meantemp'].mean()
print("Monthly Avg Temperature:\n", monthly_temp)

# Highest temp day
hottest_day = df.loc[df['meantemp'].idxmax()]
print("Hottest Day:\n", hottest_day)
```

```
Start coding or generate with AI.
```