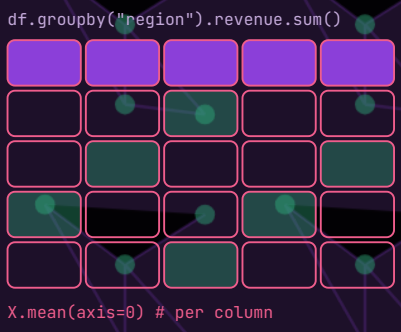
STUDY PAGE · AREA 01 OF 14

Python for ML: NumPy & pandas

01

Every ML project starts as data in a table. Learn the Python, NumPy and pandas you will use every day — and the three silent bugs that ruin datasets before any model sees them.



14 h of study · 4 modules · 8 diagrams · 3 interactives

Free tools · Python 3.10+, or Google Colab · 20-question test · 70% to pass

Lab: fix a data-prep script

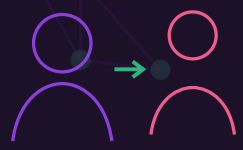
1 · Python → 2 · NumPy → 3 · pandas → 4 · Plots → Lab · data prep → Area test · 70%

lists · dicts · functions | axis · broadcasting | NaN · groupby · joins | notebook workflow | three silent bugs | 20 questions

BEFORE YOU START · A

Who this area is for

The first area of the AI/ML Engineer roadmap. Nothing is assumed beyond basic programming: by the end you can load, clean, reshape and summarise real data in Python.



🎓

CS / data / stats students

Students preparing for ML internships and new-grad roles who want the practical path, not only theory.

💻

Software engineers

Backend and full-stack developers moving into ML engineering, who already write good code and want the ML on top.

📊

Data analysts

Analysts who know SQL and spreadsheets and want to build and ship models, not just report on data.

🧪

QA & test engineers

Test automation engineers heading into ML: evaluation, data checks and model testing use the same mindset.

🔧

Data engineers

Pipeline builders who want to own the models their data feeds, and the MLOps around them.

🔄

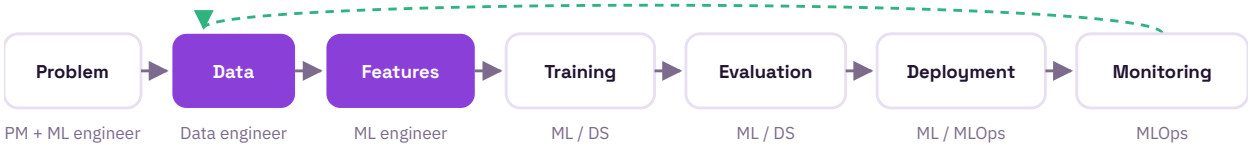
Career switchers

Engineers from other fields (EE, mechanical, finance) with some Python, aiming for an ML engineer job.

Prerequisites

You need	Level	Where to get it
Programming basics (variables, if, loops) in any language	Beginner	Any intro course; Module 1 recaps the Python
Spreadsheets (rows, columns, sum)	Helpful	Everyday experience
Maths	None yet	Area 02 covers the maths for ML

Where this area sits in the ML lifecycle — and the roles that use it



Highlighted: this area's tools power the data and feature stages

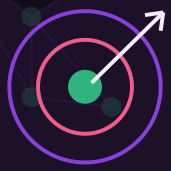
Most ML time is spent before training. NumPy and pandas are how data gets from raw files to model-ready features.

Role	How you use this area on the job	Typical first title
ML engineer	Write the data-loading and feature code that feeds every model	ML Engineer I / Machine Learning Engineer
Data scientist	Explore and summarise data in notebooks every day	Data Scientist / Applied Scientist
AI / LLM engineer	Prepare and clean datasets for evaluation and fine-tuning	AI Engineer / LLM Engineer
MLOps / platform	Build reproducible data steps that run in pipelines	MLOps Engineer / ML Platform Engineer
Data engineer	Hand clean tables to ML teams; check them with pandas	Data Engineer

BEFORE YOU START · B

What you will be able to do

Every outcome is practised in the lab and checked by the area test.

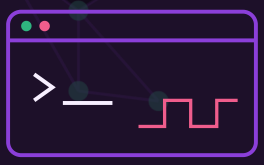


- Set up Python in a virtual environment and install packages with pip.
- Use lists, dicts, comprehensions and functions to transform data.
- Create, index and slice NumPy arrays and use boolean masks.
- Reduce along the right axis and explain broadcasting.
- Load a CSV into pandas and inspect shape, dtypes and missing values.
- Filter, add columns, group and aggregate with pandas.
- Choose the right join and verify row counts.
- Find and fix three silent data bugs in a real script.

BEFORE YOU START · C

Tools & software

Everything in this area is free and runs on any laptop, or in the browser with Google Colab.



Tool	What it does	Used in this area for	Cost
Python 3.10 – 3.13	The language of ML: every library on this page is a Python package	Every example and the lab	Free
venv + **pip**	An isolated environment per project and the package installer	Installing the area's packages	Free
NumPy	Fast n-dimensional arrays and maths	Arrays, axes, broadcasting (Module 2)	Free
pandas	Tables (DataFrames): load, clean, group and join data	Tables, NaN, groupby and joins (Module 3)	Free
Matplotlib	Charts: histograms, scatter plots, learning curves	Histograms, scatter and line plots (Module 4)	Free
JupyterLab	Notebooks: run code cell by cell and see tables and plots inline	Exploring data cell by cell	Free
VS Code + Python & Jupyter extensions	Editor, debugger and notebooks in one place	Writing and debugging code	Free
Google Colab	Hosted notebooks in the browser, with a free GPU on request	No-install option for every example	Free

Platforms you will meet at work

Platform	Vendor	What teams use it for
SageMaker	AWS	Managed notebooks, training jobs, model registry and endpoints
Vertex AI	Google Cloud	Training, pipelines, feature store, model serving and Gemini APIs
Azure Machine Learning	Microsoft	Workspaces, MLflow tracking, managed endpoints
Databricks	Databricks	Spark + notebooks + MLflow on a lakehouse
Weights & Biases	W&B	Experiment tracking and model evaluation dashboards

Same code, bigger machines

The pandas and NumPy you learn here are exactly what runs on SageMaker, Vertex AI or Databricks notebooks; only the machine gets bigger.

BEFORE YOU START · D

Install step by step

Pick your computer. About 15 minutes. If anything fails, Google Colab works with nothing installed.



Windows

Recommended: WSL 2 (Ubuntu inside Windows) — the same commands as Linux, cloud servers and most ML teams.

1

Turn on WSL and install Ubuntu

Open **PowerShell as Administrator**, run the command, then restart.

```
wsl --install
```

After the restart an Ubuntu window opens: choose a user name and password.

2

Install Python's venv support

```
sudo apt update
sudo apt install -y python3 python3-venv python3-pip git make
```

3

Create an environment and install the packages

```
python3 -m venv ~/ml
source ~/ml/bin/activate
pip install --upgrade pip
pip install numpy pandas matplotlib jupyterlab
```

Run `source ~/ml/bin/activate` in every new terminal.

4

Editor

Install VS Code, the **WSL**, **Python** and **Jupyter** extensions, then type `code .` in Ubuntu.

5

Check it works

```
python -c "import numpy, pandas, matplotlib; print(numpy.__version__, pa
```

i

No WSL on your laptop?

Install Python 3.12 from python.org (tick *Add python.exe to PATH*), then in PowerShell: `py -m venv %USERPROFILE%\ml` , `%USERPROFILE%\ml\Scripts\activate` and the same `pip install` line. Every lab runs the same.

macOS

1

Install Apple's command-line tools

```
xcode-select --install
```

2

Install Homebrew and Python

Follow the one-line command on brew.sh, then:

```
brew install python@3.12 git make
```

3

Create an environment and install the packages

```
python3 -m venv ~/ml
source ~/ml/bin/activate
pip install --upgrade pip
pip install numpy pandas matplotlib jupyterlab
```

4

Editor

Install VS Code with the **Python** and **Jupyter** extensions. Apple-silicon Macs run every lab natively.

5

Check it works

```
python -c "import numpy, pandas, matplotlib; print(numpy.__version__, pa
```

Linux

1

Ubuntu / Debian

```
sudo apt update
sudo apt install -y python3 python3-venv python3-pip git make
```

2

Fedora

```
sudo dnf install -y python3 python3-pip git make
```

3 **Arch**

```
sudo pacman -S python python-pip git make
```

4 **Create an environment and install the packages**

```
python3 -m venv ~/ml
source ~/ml/bin/activate
pip install --upgrade pip
pip install numpy pandas matplotlib jupyterlab
```

5 **Check it works**

```
python -c "import numpy, pandas, matplotlib; print(numpy.__version__, pa
```

Browser only

1 **Open Google Colab**

Go to colab.research.google.com and sign in with a Google account.

2 **New notebook**

File → New notebook. NumPy, pandas, scikit-learn, Matplotlib and PyTorch are already installed.

3 **Upload the lab**

Click the folder icon, upload the lab zip, then run `!unzip lab.zip && cd lab && python check.py` in a cell (prefix shell commands with `!`).

4 **Check the versions**

```
python -c "import numpy, pandas, matplotlib; print(numpy.__version__, pa
```

BEFORE YOU START · E

First run in 5 minutes

Make a small table, add a column and total it.



1 Activate your environment and create `first.py`

```
import numpy as np
import pandas as pd

rng = np.random.default_rng(42)
df = pd.DataFrame({
    "price": rng.normal(40, 12, 5).round(2),
    "qty": rng.integers(1, 6, 5),
})
df["revenue"] = df["price"] * df["qty"]
print(df)
print("total revenue:", round(df["revenue"].sum(), 2))
print("numpy", np.__version__, "· pandas", pd.__version__)
```

2 Run it

```
python first.py
```

Output (captured from a real run):

```
   price  qty  revenue
0  43.66    3   130.98
1  27.52    5   137.60
2  49.01    4   196.04
3  51.29    4   205.16
4  16.59    4    66.36
total revenue: 736.14
numpy 2.5.3 · pandas 3.0.5
```

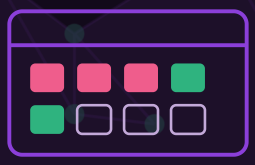
3 What to notice

The same seed (42) gives everyone the same numbers. `df["revenue"] = df["price"] * df["qty"]` multiplied whole columns at once — no loop. That is the idea behind everything in this area.

BEFORE YOU START · F

Study plan

14 hours at about 7 hours a week.



Session	Do this	Time
Day 1	Install · first run · Module 1 (Python essentials)	3 h
Day 2	Module 2 (NumPy: axis and broadcasting) + both widgets	3.5 h
Day 3	Module 3 (pandas: NaN, groupby, joins)	3.5 h
Day 4	Module 4 (plots and workflow) · Lab: fix prep.py	3 h
Day 5	Lab review · area test	1 h

MODULE 1 OF 4

Python essentials for data work

The ordinary Python every ML codebase is built from.

- Lists, dicts and comprehensions
- Functions and f-strings
- Why NumPy beats lists for numbers
- One environment per project

```
def mean(xs):
    return sum(xs)
        / len(xs)

[x*2 for x in xs]
```

⏱ 3 hours | Area 01 · Python for ML

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

1 A variable is a labelled box

≈ In plain words

A variable gives a name to a value so you can use it later. The value can be a number, some text, or a whole collection.

☀ Think of it like

A sticky label on a storage box: the label is the name, the contents are the value. Re-labelling doesn't change what's inside.

✎ Worked example

```
price = 19.99
qty = 2
revenue = price * qty
print(revenue)
```

```
# output
39.98
```

Aa The word for it — **Variable**, **assignment** (=), **expression**.

! Why it matters — Every data step is a chain of named values. Good names (`revenue` , not `x2`) make ML code readable months later.

2 STEP 2 — Lists and dicts hold many values

≈ In plain words — A list keeps values in order; a dictionary looks values up by a name (key).

☀ Think of it like — A list is a queue at a shop — first, second, third. A dict is a phone book — find the number by the name.

✎ Worked example

```
prices = [19.99, 5.50, 42.00]
print(prices[0], len(prices))

region = {"east": 119.0, "west": 40.0}
print(region["west"])
```

```
# output
19.99 3
40.0
```

Aa The word for it — **list** (indexed from 0) and **dict** (key → value).

! Why it matters — Rows of data, feature names, model settings (hyperparameters) and results are all lists and dicts.

PAUSE & TRY **What does `[19.99, 5.50, 42.00][1]` give?** Answer

5.5 — Python counts from 0, so index 1 is the second item.

3

STEP 3

Comprehensions transform a whole list

≈ **In plain words**

A comprehension builds a new list by doing the same thing to every item, in one line.

☀ **Think of it like**

A stamping machine: every item that goes in comes out stamped the same way.

✎ **Worked example**

```
prices = [19.99, 5.50, 42.00]
with_tax = [round(p * 1.08, 2) for p in prices]
cheap = [p for p in prices if p < 20]
print(with_tax, cheap)
```

```
# output
[21.59, 5.94, 45.36] [19.99, 5.5]
```

Aa **The word for it**

List comprehension: `[expression for item in items if condition]`.

! **Why it matters**

It's the bridge to vectorized thinking: “do this to every value” is exactly what NumPy and pandas do — only much faster.

4

STEP 4

Functions name a step

≈ **In plain words**

A function packages steps under a name. You give it inputs, it gives back an output.

☀ **Think of it like**

A recipe: ingredients in (parameters), dish out (return value). Cook it again with different ingredients any time.

Worked example

```python
def mean(xs):
    return sum(xs) / len(xs)

print(mean([2, 4, 9]))
```

```
# output
5.0
```

The word for it — **Function**, **parameter**, **return value**.

Why it matters — Data cleaning in real projects is a set of small tested functions — exactly like `clean_orders` in the lab.

PAUSE & TRY **What does a function without `return` give back?** Answer

`None`. A very common bug: the function computes the answer but never returns it.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Interpreter
The program (`python`) that reads your code and runs it line by line.

Virtual environment
A private folder of packages for one project, made with `python -m venv`, so projects never fight over versions.

List
An ordered, changeable collection: `[3, 1, 2]`. Can hold anything, including mixed types.

Dictionary
Key → value lookup: `{"east": 119}`. Keys are unique.

Comprehension
A one-line loop that builds a list or dict: `[x * 2 for x in xs]`.

Function
Named, reusable code with inputs (parameters) and an output (`return`).

Package
Installable code others wrote — NumPy, pandas, scikit-learn — added with `pip install`.

f-string
Formatted text: `f"{mean:.2f}"` puts a value into a string with 2 decimals.

The Python you actually need for data work

ML code is mostly ordinary Python gluing libraries together. Four things carry most of the weight: lists and dicts to hold things, loops and comprehensions to transform them, functions to name a step, and f-strings to report results.

```
prices = [19.99, 5.50, 42.00, 7.25]
qty = [2, 10, 1, 4]

revenue = [p * q for p, q in zip(prices, qty)]        # comprehension + zip
by_month = {"2026-01": 39.98, "2026-02": 84.00}        # dict
by_month["2026-03"] = 24.00                            # add a key

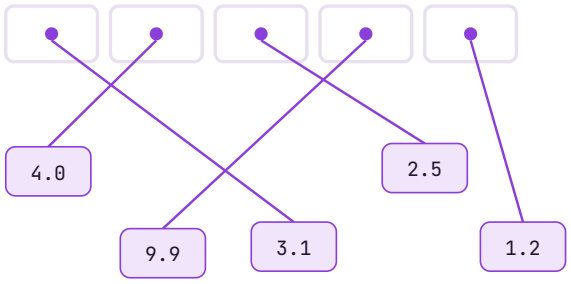
def mean(xs):
    """Average of a list of numbers."""
    return sum(xs) / len(xs)

print(revenue)
print(f"mean price = {mean(prices):.2f}")
print(sorted(by_month.items(), key=lambda kv: kv[1], reverse=True)[0])
```

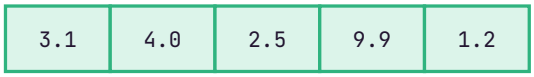
```
# output
[39.98, 55.0, 42.0, 29.0]
mean price = 18.68
('2026-02', 84.0)
```

Need	Python	Example
Keep items in order	list	`[3, 1, 2]`
Look up by name	dict	`d["east"]`
Unique values	set	`set(regions)`
Fixed record	tuple	`("east", 119)`
Transform every item	comprehension	`[f(x) for x in xs]`
Pair two lists	zip	`zip(prices, qty)`
Index and value	enumerate	`for i, x in enumerate(xs)`

Python list: pointers to objects



NumPy array: one block of numbers



dtype float64 · 8 bytes each · contiguous

a * 2 runs one C loop over the block

≈ 50–100× faster than a Python for-loop

Why ML code uses NumPy, not lists, for numbers. A list holds pointers to separate objects; an array is one typed block that C loops over at full speed.

Loops vs vectorized code: measured

```
import time
import numpy as np

xs = list(range(1_000_000))
t0 = time.perf_counter(); ys = [x * 2.0 for x in xs]; t_list = time.perf_counter
a = np.arange(1_000_000, dtype=float)
t0 = time.perf_counter(); b = a * 2.0; t_np = time.perf_counter() - t0
print(f"list comprehension: {t_list * 1000:.1f} ms")
print(f"numpy vectorized:   {t_np * 1000:.1f} ms")
print(f"speed-up: about {t_list / t_np:.0f}x")
```

```
# output
list comprehension: 38.8 ms
numpy vectorized:   3.3 ms
speed-up: about 12x
```

★ **One environment per project**

Create a venv for each project and keep a `requirements.txt` (`pip freeze > requirements.txt`). The commonest “works on my machine” bug in ML is two projects sharing one Python with different library versions.

× **Common mistakes**

Mutating a list while looping over it · using a list of numbers where an array belongs · forgetting `return` (the function gives `None`) · installing packages into the system Python instead of a venv · comparing floats with `==` (use `math.isclose` / `np.isclose`).

INTERVIEW CORNER

Q. List vs NumPy array? **A.** A list holds arbitrary Python objects (pointers); an array holds one dtype in contiguous memory, so maths on it runs in compiled loops and is far faster.

Q. Why use a virtual environment? **A.** To pin each project's package versions independently and make the setup reproducible with requirements.txt.

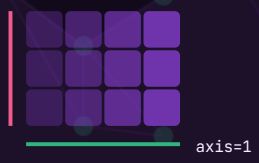
Key takeaways

1. Lists, dicts, comprehensions and functions are 90% of the Python you need.
2. Numbers in bulk belong in NumPy arrays.
3. One venv and one requirements.txt per project.

MODULE 2 OF 4

NumPy arrays, broadcasting & vectorization

Think in whole arrays: shapes, axes and the broadcasting rule.



- shape, dtype and indexing
- Boolean masks
- Reductions and axis
- Broadcasting

⏱ 3.5 hours | Area 01 · Python for ML

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

An array is a grid of numbers of one type

In plain words

NumPy's array stores many numbers of the same type, packed together. It can be a row, a table, or more dimensions.

Think of it like

An egg carton: fixed slots, all holding the same kind of thing, in neat rows and columns.

Worked example

```
import numpy as np
X = np.array([[1, 2, 3], [4, 5, 6]])
print(X.shape, X.dtype)
print(X * 10)
```

```
# output
(2, 3) int64
[[10 20 30]
 [40 50 60]]
```

The word for it

ndarray with a **shape** (rows, columns) and a **dtype**.

Why it matters

Every ML model takes its input as an array (often called X): one row per example, one column per feature.

STEP 2

2 Vectorization: whole-array maths, no loop

In plain words

Instead of writing a loop, you write the maths once and NumPy applies it to every element.

Think of it like

Painting a fence with a roller instead of a tiny brush, one plank at a time.

Worked example

```
import numpy as np
price = np.array([19.99, 5.50, 42.00])
qty = np.array([2, 10, 1])
print(price * qty)        # element by element
print((price * qty).sum())
```

```
# output
[39.98 55.   42.  ]
136.98
```

Aa The word for it

Vectorized operation; **element-wise** maths.

! Why it matters

It's typically 50–100× faster than a Python loop and shorter to read.

3 STEP 3 axis: which way to collapse

≈ In plain words

When you take a mean or sum of a table, `axis` says whether to squash the rows (one answer per column) or the columns (one answer per row).

✳ Think of it like

A class grade sheet: averaging down each column gives the average per subject; averaging across each row gives each student's average.

✎ Worked example

	Maths	Physics
Ana	80	70
Ben	60	90
axis=0 mean	**70**	**80**

Across each row (`axis=1`) gives Ana 75 and Ben 75.

Aa The word for it

axis=0: along rows → per column. **axis=1**: along columns → per row.

! Why it matters

Features are columns. Scaling features needs per-column statistics — `axis=0` . Forgetting it is lab bug 2.

PAUSE & TRY **A (100, 5) array of 100 examples and 5 features. Shape of `X.mean(axis=0)` ?** Answer

(5,) — one mean per feature.

4 STEP 4

Broadcasting: different shapes, one operation

≈ **In plain words**

NumPy can combine a big array with a smaller one by reusing the smaller one along a dimension.

☀ **Think of it like**

Adding the same discount row to every customer's bill — you write the discount once, it applies to every row.

✎ **Worked example**

1. X has shape (3, 4); the column means have shape (4,).
2. Line shapes up from the right: 4 matches 4.
3. The (4,) row is reused for all 3 rows.
4. `X - X.mean(axis=0)` subtracts each column's mean from that column.

Aa **The word for it**

Broadcasting: sizes compared right to left must be equal or 1.

! **Why it matters**

It's how feature scaling, normalization and many model computations are written in one line.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

ndarray
NumPy's n-dimensional array: one dtype, a fixed shape, contiguous memory.

shape
Size along each axis: `(3, 4)` is 3 rows and 4 columns.

dtype
Element type: `float64`, `int64`, `bool`...

axis
Which dimension an operation runs along. `axis=0` = down the rows (per column).

Vectorization
Expressing a computation on whole arrays instead of element-by-element Python loops.

Broadcasting
NumPy's rule for combining arrays of different shapes by stretching size-1 axes.

Boolean mask
An array of True/False used to select elements: `a[a > 0]`.

View vs copy
Slices are views of the same memory; changing a view changes the original.

```
import numpy as np

X = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]], dtype=float)
print(X.shape, X.dtype)
print(X[1])            # second row
print(X[:, 2])         # third column
print(X[X > 6])        # boolean mask
print(X.mean(axis=0))  # one value per column
print(X.mean(axis=1))  # one value per row
```

```
# output
(3, 4) float64
[5. 6. 7. 8.]
[ 3.  7. 11.]
[ 7.  8.  9. 10. 11. 12.]
[5. 6. 7. 8.]
[ 2.5  6.5 10.5]
```

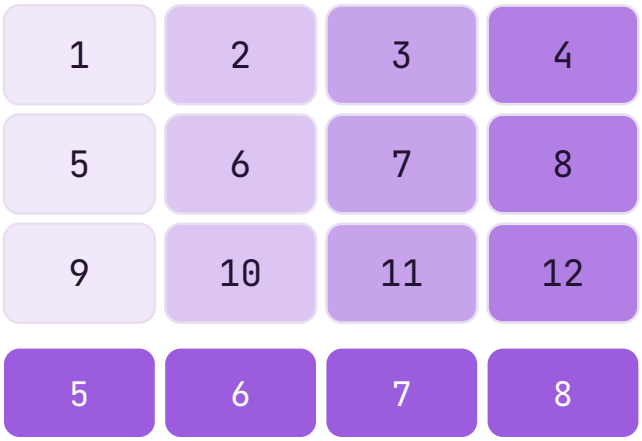


axis=0 · down the rows
`X.mean(axis=0) → [5. 6. 7. 8.]`
one value per column (per feature)

axis=1 · across the columns
`X.mean(axis=1) → [2.5 6.5 10.5]`
one value per row (per sample)

axis names the dimension that disappears. `axis=0` collapses rows → one result per column (per feature) — what you almost always want for scaling features.

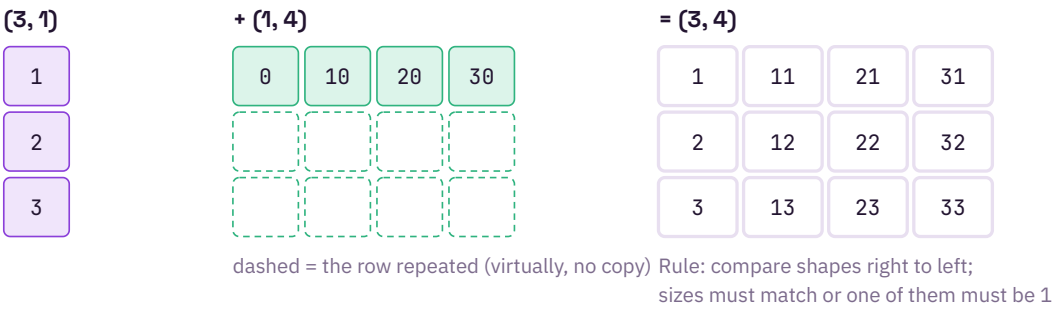
Try it: axis explorer




```
X.mean(axis=0)  →  [5 6 7 8]   shape (4,) · rows collapse → one value per column
```

Broadcasting: combining different shapes

When two arrays have different shapes, NumPy compares their shapes from the right. Each pair of sizes must be equal, or one of them must be 1 — a size-1 axis is stretched (without copying) to match. That is how you subtract a per-column mean from every row in one line.



Shapes are compared right to left. (3,1) and (1,4) both stretch to (3,4). (3,4) and (3,) also work; (3,4) and (3,) *do not* if you meant per-row — the 3 lines up with the 4 and fails.

```
import numpy as np

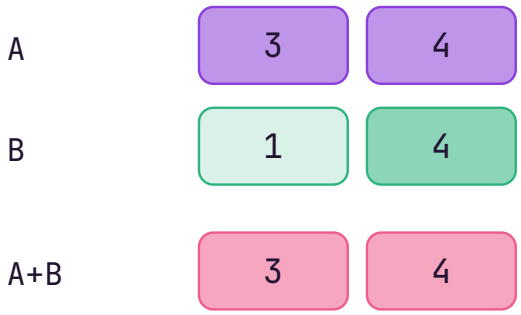
X = np.array([[1., 2., 3., 4.], [5., 6., 7., 8.], [9., 10., 11., 12.]])
col_mean = X.mean(axis=0)               # shape (4,)
Z = (X - col_mean) / X.std(axis=0)      # (3,4) - (4,) broadcasts per column
print(Z.round(2))
print("column means:", Z.mean(axis=0).round(6))

row_mean = X.mean(axis=1)               # shape (3,)
try:
    X - row_mean                        # (3,4) - (3,) does NOT broadcast
except ValueError as e:
    print("ValueError:", e)
print((X - row_mean[:, None]).round(1))  # make it (3,1) and it works
```

```
# output
[[-1.22 -1.22 -1.22 -1.22]
 [ 0.    0.    0.    0.  ]
 [ 1.22  1.22  1.22  1.22]]
column means: [0. 0. 0. 0.]
ValueError: operands could not be broadcast together with shapes (3,4) (3,)
[[-1.5 -0.5  0.5  1.5]
 [-1.5 -0.5  0.5  1.5]
 [-1.5 -0.5  0.5  1.5]]
```

Try it: will these shapes broadcast?

A shape 3,4 B shape 4




```
OK → result shape (3, 4)
3 vs 1 → stretch the 1 → 3
4 vs 4 → equal → 4
```

Operation	NumPy	Note
Create	`np.array(...)`, `np.zeros((3,4))`, `np.arange(10)`, `np.linspace(0,1,5)`	set `dtype=` if needed
Random (reproducible)	`rng = np.random.default_rng(42); rng.normal(size=5)`	new Generator API
Reshape	`a.reshape(3, 4)`, `a[:, None]`	-1 lets NumPy infer one size
Select	`a[1:3]`, `a[a > 0]`, `a[[0, 2]]`	slices are views
Reduce	`a.sum(axis=0)`, `a.mean()`, `a.argmax()`	axis = which dim disappears
Combine	`np.concatenate`, `np.stack`, `a @ b`	`@` is matrix multiply

X.mean() vs X.mean(axis=0)

Without an axis, NumPy averages *every* number in the array into one value. Standardizing features with it mixes price and quantity together — the result has the wrong mean and spread per column. That is planted bug 2 in the lab.

Common mistakes

Forgetting the axis in reductions · assuming a slice is a copy (it is a view — use `.copy()`) · integer arrays silently truncating (`np.array([1, 2]) / 2` is fine, but assigning 0.5 into an int array stores 0) · shape (n,) vs (n,1) confusion · using `np.random.seed` globals instead of a Generator.

INTERVIEW CORNER

Q. What is broadcasting? **A.** Combining arrays of different shapes by comparing sizes right to left; sizes must match or be 1, and size-1 axes are stretched virtually.

Q. What does axis=0 mean in X.mean(axis=0)? **A.** Reduce along the rows: the row dimension disappears and you get one mean per column.

Key takeaways

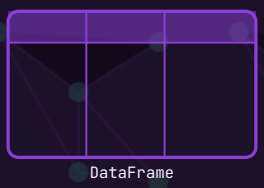
1. Arrays: one dtype, a shape, fast vectorized maths.
2. axis tells NumPy which dimension to collapse.
3. Broadcasting: right to left, equal or 1.

MODULE 3 OF 4

pandas: select, filter, group & join

Load a real table, fix its gaps and combine it with another.

- DataFrames and Series
- Missing values
- groupby
- merge and join types



⏱ 3.5 hours | Area 01 · Python for ML

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

1

STEP 1

A DataFrame is a spreadsheet you program

≈ **In plain words** — pandas gives you tables with named columns. Each column has one type, like a column of prices or a column of dates.

✳ **Think of it like** — An Excel sheet you control with code: every change is written down and can be re-run.

Worked example

```
import pandas as pd
df = pd.read_csv("orders.csv")
print(df)
```

```
# output
   order_id  customer_id  price  qty    coupon
0         1          3.0  19.99    2  SPRING10
1         2          1.0   5.50   10       NaN
2         3          NaN  42.00    1       NaN
3         4          2.0   7.25    4  SPRING10
```

The word for it

DataFrame (the table) and **Series** (one column).

Why it matters

Almost every dataset you train on starts as a DataFrame.

STEP 2

2 Missing values show up as NaN

In plain words

When a cell is empty, pandas writes NaN. You must decide: drop that row, or fill the gap.

Think of it like

A form with blanks: if the name is missing you can't process it; if the optional coupon box is blank, that's fine.

Worked example

```
import pandas as pd
df = pd.read_csv("orders.csv")
print(df.isna().sum())
print(len(df.dropna()), "vs", len(df.dropna(subset=["custo
```

```
# output
order_id       0
customer_id    1
price          0
qty            0
coupon         2
dtype: int64
2 vs 3
```

The word for it

NaN, `isna()`, `dropna(subset=…)`, `fillna()`.

Why it matters

Dropping too much data biases your model; keeping broken rows crashes it.

PAUSE & TRY

Order 3 has no customer_id; orders 2 and 3 have no coupon. Which rows should `dropna(subset=["customer_id", "price"])` drop? Answer

Only order 3. The coupon is optional.

STEP 3

3 groupby answers “per something” questions

In plain words

To get a total or average per category, group the rows by that category and aggregate each group.

Think of it like

Sorting receipts into piles by month, then adding up each pile.

✎ **Worked example**

```
import pandas as pd
df = pd.DataFrame({"region": ["east", "west", "east"], "re
print(df.groupby("region")["revenue"].sum())
```

```
# output
region
east    95
west    29
Name: revenue, dtype: int64
```

Aa **The word for it** — **groupby** — split, apply, combine.

! **Why it matters** — "Average purchase per customer", "churn rate per plan" — these features come from groupby.

4 STEP 4 Joins combine two tables

≈ **In plain words** — A join matches rows from two tables using a shared key column, like customer_id.

✳ **Think of it like** — Matching name tags to a guest list: some guests aren't on the list, some names on the list never arrived. You choose who to keep.

✎ **Worked example**

how=	keeps
inner	only matches
left	every row of the left table
right	every row of the right table
outer	everything

Aa **The word for it** — **merge** with `on=` and `how=`.

Why it matters An inner join silently drops rows whose key is missing on the other side — lab bug 3.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

DataFrame
A table: labelled rows (the index) and named columns, each column one dtype.

Series
One column (or row) of a DataFrame: values plus an index.

loc / iloc
`loc` selects by label, `iloc` by position.

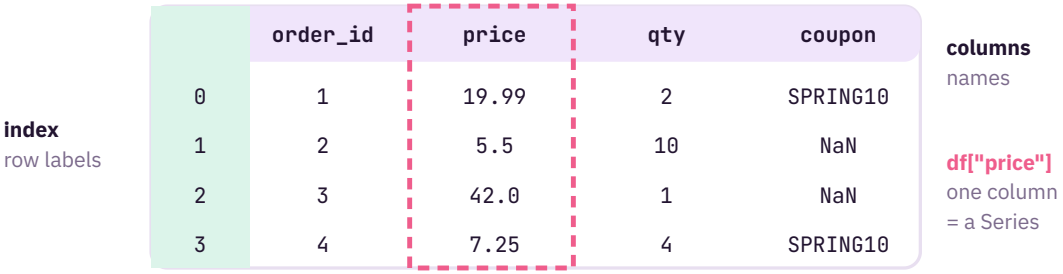
NaN
“Not a Number”: pandas' marker for a missing value.

groupby
Split rows by a key, apply an aggregation to each group, combine the results.

merge / join
Combine two tables on a key column. `how` decides which rows survive.

dtype object
pandas' type for text (or mixed) columns.

Method chaining
Writing steps as `df.dropna(...).assign(...).groupby(...)` — one readable pipeline.



A DataFrame is labelled columns of NumPy arrays. Each column is a Series with its own dtype; the index labels rows and drives alignment.

```python
import pandas as pd

df = pd.read_csv("orders.csv")
print(df.shape)
print(df.dtypes)
print(df.isna().sum())
```

```
# output
(7, 6)
order_id         int64
customer_id    float64
month              str
price          float64
qty              int64
coupon             str
dtype: object
order_id       0
customer_id    1
month          0
price          1
qty            0
coupon         5
dtype: int64
```

Select, filter, add columns

```
import pandas as pd

df = pd.read_csv("orders.csv")
cheap = df.loc[df["price"] < 10, ["order_id", "price", "qty"]]   # rows by mask,
print(cheap)
df = df.assign(revenue=df["price"] * df["qty"])                  # new column
print(df[["order_id", "revenue"]].head(3))
print(df.iloc[0, :3].to_dict())                                  # first row, fi
```

```
# output
   order_id  price  qty
1         2   5.50   10
3         4   7.25    4
   order_id  revenue
0         1    39.98
1         2    55.00
2         3    42.00
{'order_id': 1, 'customer_id': 3.0, 'month': '2026-01'}
```

Missing values: drop only what you must

```
import pandas as pd

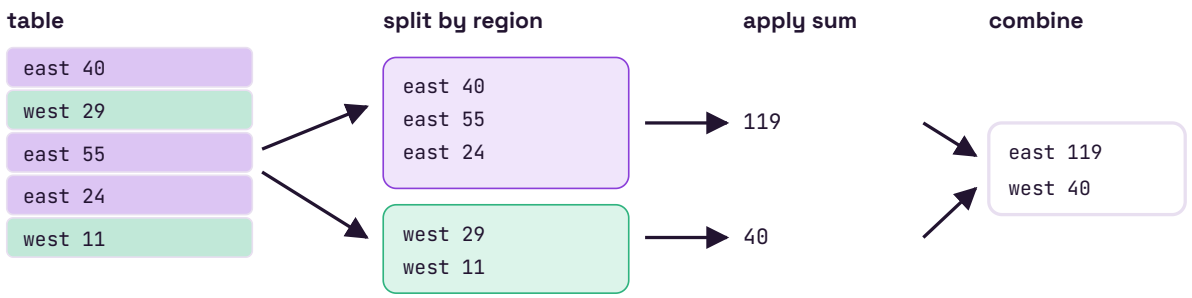
df = pd.read_csv("orders.csv")
print("dropna():                ", len(df.dropna()), "rows")       # drops rows
print("dropna(subset=[...]):    ", len(df.dropna(subset=["customer_id", "price"]
print(df["coupon"].fillna("none").value_counts().to_dict())
```

```
# output
dropna():                 2 rows
dropna(subset=[...]):     5 rows
{'none': 5, 'SPRING10': 2}
```

dropna() with no subset

`coupon` is optional, so most rows have NaN there. `df.dropna()` throws those rows away along with the genuinely broken ones — 7 orders become 1. Always say which columns are required: `dropna(subset=["customer_id", "price"])`. Planted bug 1.

groupby: split, apply, combine



One line replaces a loop and a dictionary. `df.groupby("region")["revenue"].sum()`.

```python
import pandas as pd

df = pd.read_csv("orders.csv").dropna(subset=["customer_id", "price"])
df = df.assign(revenue=df["price"] * df["qty"])
print(df.groupby("month")["revenue"].agg(["count", "sum", "mean"]).round(2))
```

```
# output
         count    sum   mean
month
2026-01      2  94.98  47.49
2026-02      1  29.00  29.00
2026-03      2  54.00  27.00
```

Joins: which rows survive



how="inner" only matching keys

how="left" all left rows

how="right" all right rows

how="outer" everything

Left table = orders, right = customers. An inner join silently drops orders from unknown customers.

how= decides what you keep. For “every order, plus its region if known”, the orders table is on the left and the join is `how="left"`.

```
import pandas as pd

orders = pd.read_csv("orders.csv").dropna(subset=["customer_id", "price"])
customers = pd.read_csv("customers.csv")
for how in ["inner", "left"]:
    j = orders.merge(customers, on="customer_id", how=how)
    print(f"{how:5}: {len(j)} rows, regions = {j['region'].fillna('unknown').uni
```

```
# output
inner: 4 rows, regions = ['east', 'west']
left : 5 rows, regions = ['east', 'west', 'unknown']
```

Try it: join explorer

order_id	customer_id	revenue	region
1	3	39.98	east
2	1	55	east
4	2	29	west
6	1	24	east

4 rows. Order 7 (customer 9) and customer 5 both vanish. Revenue counted: **147.98** of 177.98.

Task	pandas
Rows and columns	`df.shape` , `df.info()`
Summary stats	`df.describe()`
Count values	`df["col"].value_counts()`
Rename / drop	`df.rename(columns={...})` , `df.drop(columns=[...])`
Sort	`df.sort_values("revenue", ascending=False)`
Dates	`pd.to_datetime(df["month"])`
Save	`df.to_parquet("clean.parquet")` or `to_csv(index=False)`

★ **Check row counts after every join**

Print `len(df)` before and after each merge. Fewer rows than you expected means an inner join dropped keys; more rows means duplicate keys on the right side multiplied your data.

Common mistakes

`dropna()` with no subset · inner joins that silently drop rows (planted bug 3) · chained assignment `df[df.x > 0]["y"] = 1` (use `.loc`) · `apply` with a Python function where a vectorized column operation exists · forgetting `index=False` when saving CSV.

INTERVIEW CORNER

Q. loc vs iloc? **A.** loc selects by label (index values, column names); iloc by integer position.

Q. Why did my row count change after merge? **A.** An inner join drops keys missing on either side; duplicate keys on the right side multiply rows. Check counts before and after.

Key takeaways

1. A DataFrame is named columns of arrays.
2. Say which columns are required when dropping NaN.
3. Pick the join on purpose and check the row count.

MODULE 4 OF 4

Plots and a clean notebook workflow

See your data, and make your results reproducible.

- Histogram, scatter, line
- Notebooks and hidden state
- Project layout and seeds
- Lab walkthrough



2.5 hours | Area 01 · Python for ML

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

Look before you model

In plain words

Before any model, plot each column. You'll spot typos, outliers, gaps and skew that numbers alone hide.

Think of it like

A doctor looks at an X-ray before operating.

Worked example

Plot	Shows
Histogram	the shape of one column
Scatter	how two columns relate
Line	change over time

The word for it

Exploratory data analysis (EDA).

Why it matters — Most model problems are data problems you could have seen in a plot.

2 STEP 2 Notebooks remember what you ran

In plain words — A notebook keeps every variable from every cell you ran — in the order you ran them, not the order on the page.

Think of it like — A whiteboard that never erases: old scribbles stay and can confuse the next person.

Worked example

1. Run cell 1: load data.
2. Run cell 3: count rows → 400.
3. Edit and run cell 2: filter to 300 rows.
4. Cell 3 still shows 400 until you rerun it.

The word for it — **Kernel** state and **Restart & Run All**.

Why it matters — A result you can't reproduce isn't a result. Interviewers ask about this.

PAUSE & TRY

Your teammate's notebook works for them but crashes for you at cell 5. Most likely cause?

Answer

Hidden state: they defined something in a cell they later deleted or ran out of order. Restart & Run All reproduces the crash.

3 STEP 3 Seeds make random numbers repeatable

In plain words — Random number generators start from a seed. The same seed gives the same “random” numbers every time.

Think of it like

Shuffling cards in exactly the same way every time you say “42”.

Worked example

```
import numpy as np
print(np.random.default_rng(42).integers(0, 100, 5))
print(np.random.default_rng(42).integers(0, 100, 5))
```

```
# output
[ 8 77 65 43 43]
[ 8 77 65 43 43]
```

The word for it

Seed; `np.random.default_rng(seed)`.

Why it matters

Train/test splits and model initialisation are random. Seeds let you and reviewers get the same numbers.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Notebook

A document of code cells, outputs and notes (Jupyter), run one cell at a time.

Kernel

The Python process behind a notebook; it keeps variables between cells.

Hidden state

Variables left over from cells run earlier or out of order.

Figure / Axes

Matplotlib's canvas and one plot area on it.

Histogram

Counts of values in ranges (bins): the shape of one column.

Reproducibility

Anyone can rerun your code from scratch and get the same numbers.

Three plots you will make every week

```
import matplotlib.pyplot as plt
import numpy as np

rng = np.random.default_rng(0)
price = rng.normal(40, 12, 500)
qty = rng.integers(1, 6, 500)

fig, axes = plt.subplots(1, 3, figsize=(12, 3.5))
axes[0].hist(price, bins=30)                    # distribution of one column
axes[0].set_title("price")
axes[1].scatter(price, price * qty, s=6)        # relationship of two columns
axes[1].set_xlabel("price"); axes[1].set_ylabel("revenue")
axes[2].plot(np.cumsum(price * qty))            # trend over time / order
axes[2].set_title("cumulative revenue")
fig.tight_layout()
fig.savefig("eda.png", dpi=120)
print("saved eda.png", fig.get_size_inches())
```

```
# output
saved eda.png [12.   3.5]
```

Question	Plot	Matplotlib
What does this column look like?	histogram	`ax.hist(x, bins=30)`
Do two columns move together?	scatter	`ax.scatter(x, y)`
How does it change over time?	line	`ax.plot(t, y)`
How do groups compare?	bar	`ax.bar(labels, values)`
Outliers per group?	box plot	`ax.boxplot(groups)`

```
[1]  df = pd.read_csv('orders.csv')
[4]  df = df[df.price > 10]
[2]  print(len(df)) # says 7
[3]  model.fit(df)
```

Execution order ≠ page order

Cell 2 printed 7 before cell 4 filtered. Reading top to bottom, you would think the model saw filtered data. It did not.

Fix: Kernel → Restart & Run All

before you trust or share any result.

Notebooks remember everything you ran, in the order you ran it. The page can lie; the kernel does not.

A clean workflow

Step	Do
1	One folder per project: `data/` , `notebooks/` , `src/` , `requirements.txt`
2	Explore in a notebook; move code you reuse into functions in `src/`
3	Set seeds: `rng = np.random.default_rng(42)`
4	Restart & Run All before sharing a result
5	Save cleaned data to a file; never edit raw data in place
6	Commit code (not big data files) to git

i

Notebook for exploring, .py for keeping

Notebooks are great for looking at data. Code that must run again — cleaning, features, training — belongs in `.py` files with functions you can test, exactly like `prep.py` in the lab.

The lab: three bugs in a data-prep script



400 orders, 55 customers, 6 rules. Every bug runs without an error — only the checks reveal them.

Common mistakes

Trusting a notebook without Restart & Run All · plots without labels or titles · editing raw CSVs by hand · no seed, so numbers change every run · huge notebooks that nobody can rerun.

INTERVIEW CORNER

Q. Why can't you trust a notebook that ran fine? **A.** Cells may have run out of order, leaving hidden state; a clean Restart & Run All proves the result is reproducible.

Q. First plots for a new dataset? **A.** A histogram per numeric column, value counts for categories, and scatter plots of features against the target.

Key takeaways

1. Histogram, scatter, line: shape, relationship, trend.
2. Restart & Run All before you trust a number.
3. Explore in notebooks, keep code in .py files.

PRACTICE

Check yourself

Answer in your head, then open each card. The area test has 20 more.

`np.array([[1,2],[3,4]]).sum(axis=0)` ?

`[4 6]` — the rows collapse, one sum per column.

Can shapes (5,3) and (3,) broadcast? (5,3) and (5,)?

(5,3)+(3,) yes; (5,3)+(5,) no — reshape to (5,1) first.

What does `df.dropna()` drop?

Every row with a NaN in any column — including optional ones.

Keep every order and add the region when known: which join?

```
orders.merge(customers, on="customer_id", how="left").
```

loc or iloc for “rows where price > 10, column qty”?

`df.loc[df.price > 10, "qty"]` — labels and a mask.

Revenue per month in one line?

```
df.groupby("month")["revenue"].sum()
```

Your notebook gives a different answer after a restart. Why?

Hidden state: cells had run out of order or depended on deleted cells.

Why is `a * 2` on an array faster than a list comprehension?

One compiled loop over contiguous memory instead of a Python-level loop over objects.

REFERENCE

Interview questions

Asked in ML engineer and data scientist interviews. Try an answer out loud before opening.



How do you inspect a new dataset quickly?

`df.shape`, `df.dtypes`/`info()`, `df.isna().sum()`, `df.describe()`, value counts for categories, then histograms.

Explain broadcasting with an example.

Subtracting a (4,) column-mean vector from a (3,4) matrix: shapes compared right to left, 4 matches 4, the vector is reused for each row.

How do you handle missing values?

Understand why they are missing; drop rows only when a required field is missing; otherwise impute (Area 03) and possibly add a was-missing flag.

What is the difference between merge and concat?

merge joins on key columns (like SQL); concat stacks tables along an axis.

Why avoid apply() on large DataFrames?

It calls a Python function per row; vectorized column operations run in compiled code and are much faster.

View vs copy in NumPy?

Basic slices are views sharing memory; fancy indexing and .copy() make copies. Writing to a view changes the original.

How do you make an analysis reproducible?

Pinned environment, seeds, code in scripts/functions, raw data untouched, Restart & Run All.

What does groupby().agg() return?

One row per group key, with one column per aggregation requested.

REFERENCE

Glossary

Every term used on this page, A to Z.

A a

Array
NumPy's n-dimensional, single-dtype container.

axis
Dimension an operation runs along; it disappears in reductions.

Boolean mask
True/False array used to select values.

Broadcasting
Combining different shapes by stretching size-1 axes.

Comprehension
One-line loop that builds a list/dict.

CSV
Comma-separated text table.

DataFrame
pandas table of named columns.

dict
Key-value mapping.

dtype
Element type of an array or column.

f-string
String with embedded expressions.

groupby
Split-apply-combine aggregation.

iloc
Position-based selection.

Index
Row labels of a DataFrame/Series.

Jupyter
Notebook environment for interactive code.

Kernel
Process that runs a notebook's code.

loc
Label-based selection.

merge
Join two tables on keys.

NaN
Missing value marker.

pip
Python package installer.

Series
One labelled column.

shape
Sizes along each axis.

Vectorization
Whole-array operations instead of loops.

venv
Per-project Python environment.

View
Array that shares memory with another.

REFERENCE

FAQ & troubleshooting

The problems learners hit most in this area, and the fix for each.



You see	Why	Fix
ModuleNotFoundError: No module named 'pandas'	Package not installed in the active environment	Activate the venv (`source ~/ml/bin/activate`) then `pip install pandas`
ValueError: operands could not be broadcast together	Shapes don't line up right-to-left	Print `.shape` of both; add an axis with `[:, None]`
SettingWithCopyWarning	Chained indexing assigned to a temporary copy	Use `df.loc[mask, "col"] = value`
Row count jumped after a merge	Duplicate keys in the right table	Check `customers.customer_id.is_unique` ; deduplicate first
Column of numbers has dtype object	A stray text value (e.g. "n/a")	`pd.to_numeric(col, errors="coerce")` , then inspect the NaNs
Plot window never appears (WSL/SSH)	No display	Save with `fig.savefig("plot.png")` or use Jupyter
pip: externally-managed-environment	Installing into system Python on new Ubuntu/macOS	Create and activate a venv first

REFERENCE

Further reading

Go deeper when you are ready.



Resource	Why read it
Wes McKinney, *Python for Data Analysis* (3rd ed., O'Reilly) — free online at wesmckinney.com/book	By the author of pandas: NumPy and pandas in depth
NumPy user guide — Broadcasting (numpy.org/doc)	The exact broadcasting rules with diagrams
pandas user guide — 10 minutes to pandas; Merge, join, concatenate	Official quick start and the join reference
Jake VanderPlas, *Python Data Science Handbook* — free online	NumPy, pandas and Matplotlib chapters with examples

HANDS-ON LAB · ABOUT 2 HOURS

Lab 01 · Fix a data-prep script

`prep.py` cleans 400 orders, standardizes three numeric columns and totals revenue per region. It runs without errors — and three of its answers are wrong. `check.py` tests six rules from the specification.

1. `pip install -r requirements.txt`, then `python check.py`. Read every FAIL line: which rule, and what number did you get?
2. Bug 1 (clean_orders): the row count is far too low. Which rows should survive?
3. Bug 2 (standardize): column means are not 0. Which argument is missing?
4. Bug 3 (revenue_by_region): regions don't add up to the total and there is no “unknown”. Which join keeps every order?
5. Fix one bug at a time and re-run until every line says ok.

Done when you see `PASS: all 6 checks passed — prep.py matches the specification`.