



Skill Navora

COMPLETE HANDBOOK

Data Science & AI Handbook

124 lessons across Excel, SQL, Power BI, AI for analytics, Python, statistics and machine learning, generative and agentic AI, and career preparation.

124 lessons

Version 2026.10

Updated October 2026

Created by **Varun Kumar**

www.skillnavora.in

Contents

Every entry is a clickable link. The PDF also has bookmarks for each lesson and section.

Advanced Excel

- [01](#) Getting Started with Excel for Data Analysis
- [02](#) Data Entry, Data Validation & Data Cleaning
- [03](#) Number, Custom & Conditional Formatting
- [04](#) Formulas, Cell References & Aggregate Functions
- [05](#) Text Functions, Logical Functions & IF
- [06](#) Sorting, Filtering & Dynamic Arrays
- [07](#) Lookup Functions: VLOOKUP, XLOOKUP and INDEX-MATCH
- [08](#) Pivot Tables from Basic to Advanced
- [09](#) Date, Time & Number Functions and Named Ranges
- [10](#) Charts and Advanced Charts
- [11](#) Interactive Dashboard: Superstore Case Study
- [12](#) Power Query, Power Pivot, Analysis ToolPak & What-If Analysis

SQL for Data Analysis

- [01](#) Databases, RDBMS & SQL Basics
- [02](#) ER Diagrams, Relationships, Keys & Normalisation
- [03](#) Data Types, Operators, NULLs, Variables & Comments
- [04](#) DDL, DML & Constraints: CREATE, ALTER, INSERT, UPDATE, DELETE
- [05](#) SELECT, WHERE, ORDER BY & Filtering Operators
- [06](#) SQL Functions: Aggregate, Text, Date & Number
- [07](#) GROUP BY, HAVING, CASE WHEN, ROLLUP, CUBE & COALESCE
- [08](#) Joins and a Retail Case Study
- [09](#) Set Operators: UNION, UNION ALL, INTERSECT & EXCEPT
- [10](#) Subqueries, Derived Tables & EXISTS
- [11](#) OFFSET-FETCH and Window Functions I: OVER, PARTITION BY & Ranking
- [12](#) Window Functions II: Running Totals, LEAD, LAG & Frames
- [13](#) CTEs, Views, Temporary Tables, Stored Procedures & Functions
- [14](#) Transactions, Indexes, Performance & Security
- [15](#) End-to-End SQL Case Study & Interview Practice

Power BI

- [01](#) Getting Started with Power BI & Your First HR Report
- [02](#) Power Query, Star Schema & Data Modelling (Adventure Cycles)
- [03](#) DAX Fundamentals: Measures, Filter Context & CALCULATE
- [04](#) DAX Time Intelligence, Ranking, Parameters & Dynamic Measures
- [05](#) Visuals, Filters, Drill-through, Tooltips, Buttons & Report Design
- [06](#) Power BI Service, Connectivity Modes, Security & Capstone

Tableau

- [01](#) Getting Started with Tableau: Interface, Data Connections and Your First Views
- [02](#) Charts in Tableau: From Bar and Line to Maps, Dual Axis, Histograms and Pareto
- [03](#) Calculations in Tableau: Calculated Fields, Table Calculations and LOD Expressions
- [04](#) Filters, Parameters, Sets, Groups and Hierarchies in Tableau
- [05](#) Data Modelling in Tableau: Relationships, Joins, Unions, Blending, Extracts and Tableau Prep
- [06](#) Dashboards, Actions, Stories and Publishing in Tableau (Capstone)

AI for Data Analysis

- [01](#) Generative AI for Analysts: How It Works, Prompting & AI vs GenAI vs Agentic AI
- [02](#) AI with SQL: Text-to-SQL, Docker, Metabase & MinusX
- [03](#) AI in Excel & Office: Copilot, Gemini, Claude, Python in Excel & Agents
- [04](#) The AI-Assisted Analyst Workflow, Agents & Governance (Capstone)

Python for Data Science

- [01](#) Introduction to Python: Why, Install & Jupyter Notebooks
- [02](#) Variables, Data Types, Type Conversion, print(), f-strings & input()

03 Operators, Strings, Indexing, Slicing & String Methods

04 Control Flow: if, elif, else, Nested Conditions & match

05 Loops: while, for, range, break, continue & Patterns

06 Lists: Creating, Slicing, Methods, Sorting & Comprehensions

07 Tuples & Sets: Immutable Records, Unpacking and Set Algebra

08 Dictionaries: Key-Value Data, Counting and Grouping

09 Functions: Reusable Code, Arguments, Scope and Docstrings

10 Comprehensions, Lambda, map/filter and Generators

11 Object-Oriented Python: Classes, Objects, Inheritance and Dataclasses

12 Files, Exceptions, JSON and CSV

13 Modules, Packages, Virtual Environments and Git

14 Debugging, Testing, Regular Expressions and Code Quality

15 NumPy: Fast Arrays, Vectorisation and Broadcasting

16 pandas I: Series, DataFrames, Reading Data, Selecting and Filtering

17 pandas II: Data Cleaning: Missing Values, Types, Duplicates, Text and Outliers

18 pandas III: GroupBy, Pivot Tables, Merge/Join and Reshape (melt/unpivot)

19 pandas IV: Dates, Time Series, Resampling and Rolling Windows

20 Data Visualisation I: Matplotlib and Choosing the Right Chart

21 Data Visualisation II: Seaborn Statistical Charts and Interactive Plotly

22 Exploratory Data Analysis (EDA) Case Study: Telecom Customer Churn

23 Python and SQL: sqlite3, DuckDB, SQLAlchemy and pandas.read_sql

24 APIs, JSON and Web Scraping with requests and BeautifulSoup

25 Performance and Bigger Data: Vectorisation, Memory, Parquet, Polars and DuckDB

26 DSA I: Big-O, Arrays, Strings, Hashing, Two Pointers and Sliding Window

27 DSA II: Stacks, Queues, Linked Lists, Recursion, Sorting and Binary Search

28 DSA III: Trees, Heaps, Graphs (BFS/DFS) and Dynamic Programming

29 Python Capstone: End-to-End Retail Sales Analytics Project

Statistics & Machine Learning

01 Descriptive Statistics: Centre, Spread, Shape and Relationships

02 Probability and Distributions: Rules, Bayes, Binomial, Poisson and Normal

03 Sampling, the Central Limit Theorem and Confidence Intervals

04 Hypothesis Testing I: The Framework, p-values, z-tests and t-tests

05 Hypothesis Testing II: Chi-Square, ANOVA, Non-Parametric Tests and Correlation Tests

06 A/B Testing and Experimentation: Design, Power, Analysis and Pitfalls

07 Introduction to Machine Learning: Types, Business Framing and the ML Lifecycle

08 The scikit-learn Workflow: Train/Test Split, Baselines, Pipelines and Saving Models

09 Data Preprocessing and Feature Engineering for ML

10 Linear Regression: Fitting, Interpreting, Assumptions and Diagnostics

11 Gradient Descent: How Models Learn (Built from Scratch in NumPy)

12 Overfitting, Polynomial Regression and Regularisation (Ridge, Lasso, Elastic Net)

13 Logistic Regression: Probabilities, Odds Ratios and Credit-Risk Scoring

14 Classification Metrics: Confusion Matrix, Precision, Recall, ROC, PR and Business Value

15 Cross-Validation and Hyperparameter Tuning: K-Fold, GridSearchCV, RandomizedSearchCV and Optuna

16 K-Nearest Neighbours: Classifier and Regressor (Bank Loans Case Study)

17 Decision Trees: Splits, Impurity, Pruning and Interpretable Rules

18 Ensembles I: Bagging and Random Forests

19 Ensembles II: Gradient Boosting, XGBoost, LightGBM and CatBoost

20 Support Vector Machines and Naive Bayes

21 Imbalanced Classification: Fraud Detection with Class Weights, Resampling and Thresholds

22 Clustering I: K-Means and RFM Customer Segmentation

- 23** Clustering II and Dimensionality Reduction: Hierarchical, DBSCAN, Gaussian Mixtures, PCA, t-SNE and UMAP
- 24** Time-Series Forecasting: Decomposition, Baselines, ETS, ARIMA and ML with Lag Features
- 25** NLP and Text Analytics: Cleaning, TF-IDF, Sentiment Classification, Topics and Embeddings
- 26** Neural Networks and Deep Learning: From a Neuron to Transformers
- 27** Explainable AI and MLOps: SHAP, Fairness, Deployment, Monitoring and Responsible ML
- 28** ML Capstone: End-to-End Churn Prediction and Retention Strategy

Generative AI & Agentic AI

- 01** How Large Language Models Work: Tokens, Embeddings, Transformers and Training
- 02** Prompt Engineering in Depth: Patterns, Templates, Few-Shot, Reasoning and Structured Output
- 03** Calling LLM APIs from Python: Messages, Streaming, Structured Output, Costs and Reliability
- 04** Embeddings and Vector Search: Meaning as Numbers, Similarity and Vector Databases
- 05** Retrieval-Augmented Generation (RAG): Build, Evaluate and Improve a Q&A Assistant
- 06** Tool Use and Function Calling: Letting LLMs Query Data, Call APIs and Take Actions
- 07** AI Agents and Agentic AI: Loops, Planning, Memory, Multi-Agent Systems and When to Use Them
- 08** Model Context Protocol (MCP): Connecting AI Assistants to Your Data and Tools
- 09** Evaluating LLM Applications and Building Guardrails: Test Sets, LLM-as-Judge, Injection, PII and Safety
- 10** Customising Models: Prompting vs RAG vs Fine-Tuning, LoRA, Distillation and Open-Weight Models

- 11** GenAI for Data Work: Text-to-SQL, AI Coding Assistants, Data Cleaning and Automated Insights
- 12** Building and Shipping a GenAI App: Architecture, Streamlit/FastAPI, Cost, Observability and Responsible AI (Capstone)

AI Agents, MCP & APIs for PMs and BAs

- 01** APIs Without Code: How Apps Talk, REST, JSON, Authentication, Webhooks and Postman for PMs and BAs
- 02** MCP (Model Context Protocol) Without Code: Connect Claude to Your Apps and Data
- 03** How AI Agents Work and How to Build One Without Code
- 04** Connect All Your Apps with Claude, n8n, Make, Zapier and Replit
- 05** Claude Code Complete Tutorial: From First Prompt to Agents, MCP, Hooks and Automation
- 06** Voice Agents and Voice Cloning: Tools, Architecture, Build Steps and Responsible Use

AI & No-Code Tools Lab

- 01** Figma Complete Tutorial: Interface, Auto Layout, Components, Variables, Prototypes, Dev Mode and Figma AI
- 02** n8n Complete Tutorial: Workflow Automation, Data Pipelines and AI Agents

Capstone, Portfolio & Interviews

- 01** Portfolio, GitHub and Resume for Data Roles
- 02** Data Analyst Interview Preparation: SQL, Excel, Power BI, Statistics and Case Questions
- 03** Data Scientist, ML and GenAI Interview Preparation: Questions and Model Answers
- 04** Your 90-Day Plan: Learning Roadmap, Job Search Strategy and Growing in Your First Role

TRACK 5

AI for Data Analysis

Generative AI for analysts: prompting, AI vs GenAI vs agentic AI, AI over SQL (Metabase, MinusX), Copilot and Gemini in Excel, and office agents.

1. Generative AI for Analysts: How It Works, Prompting & AI vs GenAI vs Agentic AI
2. AI with SQL: Text-to-SQL, Docker, Metabase & MinusX
3. AI in Excel & Office: Copilot, Gemini, Claude, Python in Excel & Agents
4. The AI-Assisted Analyst Workflow, Agents & Governance (Capstone)

Generative AI for Analysts: How It Works, Prompting & AI vs GenAI vs Agentic AI

Understand what generative AI is and how large language models behave (tokens, context, temperature, hallucination), how AI, machine learning, deep learning, GenAI and agentic AI differ, and how to write prompts that give analysts accurate, checkable results, with reusable templates and privacy rules.

● DVA with AI · class 1

✦ Models and tools as of 2026

Beginner

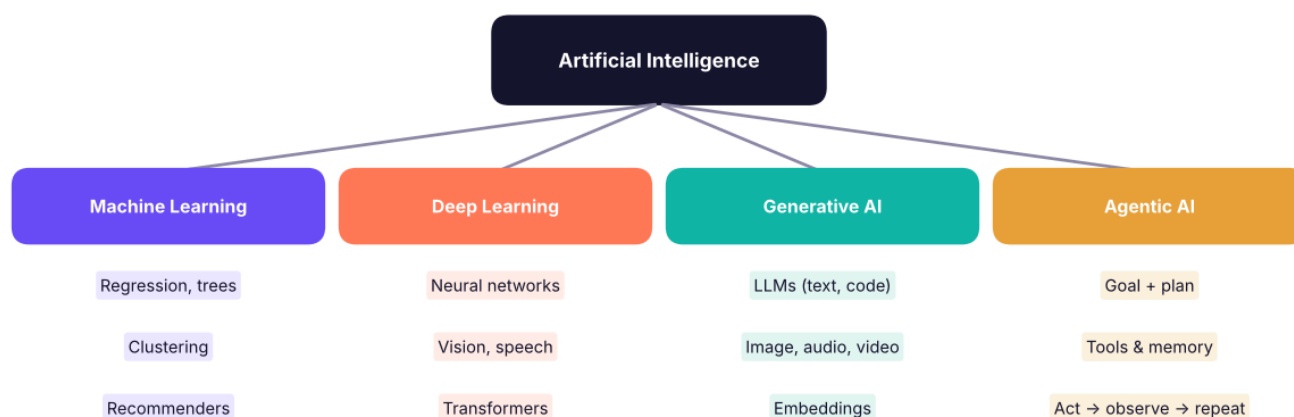
🕒 110 min

By the end of this lesson you can

- Explain the relationship between AI, ML, deep learning, generative AI and agentic AI
- Describe in plain language how an LLM generates text and why it can be confidently wrong (hallucination)
- Write prompts using role, context, task, format and constraints, with examples and step-by-step reasoning
- Use prompt templates for common analyst tasks and know what never to paste into an AI tool
- Choose a suitable tool and check an AI answer before using it

The AI family tree

AI ✓ ML ✓ Deep learning ✓ Generative AI → Agents



How the terms relate. Each ring is a subset of the one before it. Agentic AI uses generative models as its brain but adds goals, tools and loops.

Term	What it means	Example
AI	Machines doing tasks that normally need human intelligence	Rule - based chess engine, spam filter
Machine learning (ML)	Systems that learn patterns from data instead of following hand - written rules	Predict loan default from past loans
Deep learning	ML with many - layered neural networks	Image recognition, speech - to - text
Generative AI (GenAI)	Models that create new content (text, images, code, audio)	ChatGPT, Claude, Gemini, Copilot, Midjourney
LLM	Large language model: the GenAI engine for text and code	GPT, Claude, Gemini, Llama
Agentic AI	An LLM plus goals, tools and a loop : it plans, calls tools (run SQL, search, edit files), checks results and continues until the goal is met	An analyst agent that explores a database and writes the report
AGI	Hypothetical AI at human level across tasks	Not here yet; ignore the hype

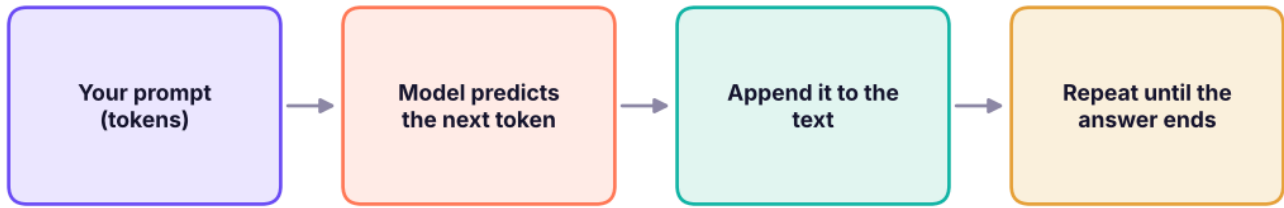
AI vs GenAI vs agentic AI

	Traditional ML	Generative AI	Agentic AI
Output	A prediction (number, label)	New content (text, code, image)	A completed multi - step task
Input	Structured data	A prompt (text, files, images)	A goal in plain language
Control	You train and deploy a model	You ask once, get one answer	The system decides the next step
Example	"Will this customer churn?" → 0.82	"Summarise these reviews" → paragraph	"Find why sales fell in Q3 and draft a memo" → runs queries, charts, writes
Main risk	Bias, drift	Hallucination	Hallucination plus actions : wrong tool calls, runaway cost, security

How an LLM works (without maths)

An LLM is trained on vast amounts of text to predict **the next token** (a word or word piece). Repeating that prediction one token at a time produces sentences, code and tables. It has learnt statistical patterns of language, facts and reasoning styles, but it does **not look things up** unless it is given a tool or documents.

One token at a time



Generating an answer: the prompt is split into tokens; the model repeatedly predicts the next token from everything so far.

Concept	Meaning	Why analysts care
Token	Roughly $\frac{3}{4}$ of an English word (about 4 characters)	Limits and costs are measured in tokens
Context window	How much text the model can consider at once (prompt + answer)	A long file may not fit; the model forgets what is outside
Temperature	Randomness of word choice	Low (0–0.3) for SQL, maths and extraction; higher for brainstorming
System prompt	Hidden instructions that set behaviour	Custom GPTs/Projects store your standing instructions
Training cut-off	The model's knowledge stops at some date	It may not know recent tools or prices; give it documents
Grounding / RAG	Supplying real documents or data in the prompt (Retrieval-Augmented Generation)	Cuts hallucination: "answer only from this file"
Tool use / function calling	The model asks the app to run a calculator, database query or code	Makes answers computed instead of guessed
Multimodal	Accepts images, PDFs, audio as well as text	Paste a chart screenshot and ask what it shows

WATCH OUT Hallucination

Because an LLM generates *plausible* text, it can state false things with complete confidence: invented statistics, non-existent functions, wrong column names, fake citations. **Never use an AI answer as a fact until you have checked it:** run the code, recompute the number, open the source.

Prompt engineering for analysts

A prompt is an instruction plus the material the model needs. The most reliable structure has five parts:

The R-C-T-F-C prompt

1. Role

Who should it act as? 'You are a senior retail data analyst.'

2. Context

Business, data and definitions: columns, grain, time period, what 'sales' means.

3. Task

One clear action: 'Find the three biggest drivers of the margin drop and quantify each.'

4. Format

Table, bullet list, SQL only, JSON; length; audience.

5. Constraints

'Use only the data given. If unsure, say so. Show assumptions. Do not invent numbers.'

+ Examples

One or two samples of the exact output you want (few-shot).

A prompt recipe: Role, Context, Task, Format, Constraints.

Weak vs strong prompt

Weak: "Analyse my sales data."

Strong:

Text

Role: You are a senior retail data analyst for an Indian omnichannel retailer.

Context: Table superstore(order_id, order_date, region, category, sub_category, sales, discount, profit),

one row per order line, Jan 2023-Dec 2025, amounts in rupees. Margin = profit / sales.

Task: Explain why overall margin is only about 9%. Look at category, sub-category and discount band.

Format: (1) a 3-bullet summary for the CFO, (2) a table of the 5 worst sub-categories (sales, profit, margin),

(3) the SQL Server queries you used.

Constraints: Use only the columns listed. State any assumption. If a number cannot be computed from the table, say so.

The strong prompt names the grain, the metric definition, the output format and the rules for uncertainty. The result is far more likely to be right *and* checkable.

Core techniques

Technique	How	Use for
Be specific	Name columns, periods, units, audience	Everything
Few-shot examples	Show 2–3 input→output pairs	Classification, extraction, tone matching
Step by step	"First list the assumptions, then compute, then summarise"	Multi-step analysis; makes errors visible
Ask for checks	"Then list three ways this answer could be wrong"	Self-critique
Structured output	"Return JSON with keys..."	Feeding results into code
Delimiters	Put data in <code>"""triple quotes"""</code> or XML tags	Stops the model confusing instructions with data
Iterate	Treat the first answer as a draft; refine with follow-ups	Always
Give it the schema, not the data	Column names and 5 sample rows, not the whole file	Privacy, token limits

Prompt templates for common analyst jobs

Job	Template (replace brackets)
Explain a formula	"Explain this Excel formula step by step for a beginner and give a simpler alternative: [formula]"
Write SQL	"Dialect: SQL Server. Tables: [schema with keys and grain]. Write a query that [question]. Return only SQL, then list assumptions."
Debug code/SQL	"This [language] query gives [wrong result / this error]. Expected: [x]. Query: [code]. Find the cause and show the fix."
Clean data	"Here are 10 sample values of column [x]: [list]. Propose normalisation rules and a mapping table; flag ambiguous cases."
Interpret results	"Here is a result table: [table]. Write 3 insights for a non-technical manager, each with a number, and 2 caveats."
Choose a chart	"I want to show [message] to [audience] with [data]. Recommend a chart and what to label."
Draft a report	"Turn these findings into a one-page memo with: situation, key findings, recommendation, risks. Keep it under 250 words."
Interview practice	"Act as an interviewer for a data analyst role; ask one SQL question at a time and give feedback on my answer."

EXAMPLE From vague to checkable

Prompt: "Explain this formula: `=IFERROR(INDEX($C:$C,MATCH(1,($A:$A=F2)*($B:$B=G2),0)), "Not found")` " A good answer says it looks up C where A equals F2 **and** B equals G2 (a two-condition lookup), returns "Not found" if none, and notes it needs array evaluation in older Excel. You can verify by testing it on 3 rows.

Choosing and using AI tools

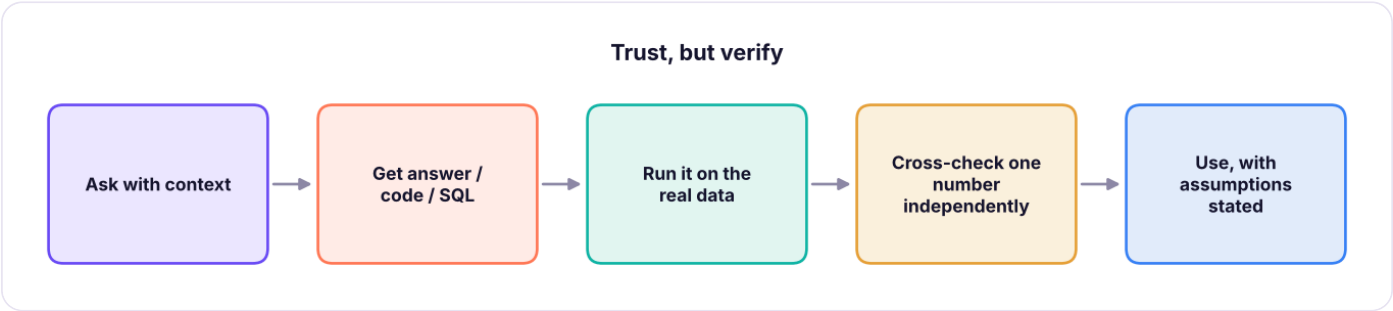
Tool type	Examples	Good for	Watch out
General chat assistants	Claude, ChatGPT, Gemini, Copilot, Grok, Perplexity	Explaining, drafting, code, brainstorming	Data privacy; hallucination
Search-grounded	Perplexity, Gemini with search, Copilot with web	Fresh facts with citations	Check the cited pages; sources vary in quality
Document/notebook tools	NotebookLM, Projects, Custom GPTs	Answering only from your files	Quality depends on the files
Embedded in apps	Copilot in Excel/Power BI, Gemini in Sheets, Metabase/ MinusX, Claude in Excel	Work where the data is	Permissions; see next lessons
Coding assistants and agents	Claude Code, Cursor, GitHub Copilot, Codex	Writing and running code, analysis scripts	Review diffs; run tests; secrets

Never paste

1. **Personal data** (names with phone numbers, emails, Aadhaar/PAN, health, salary of identifiable people) unless your company's approved tool and policy allow it.
2. **Confidential business data** (unreleased financials, customer lists, contracts, source code, credentials, API keys) into public tools.
3. **Anything you are not allowed to share**: client data under NDA.

Use **enterprise versions** with data-protection terms, **mask or sample** the data (replace names with IDs, share the schema and 5 fake rows), and follow your organisation's AI policy. In India, personal data is also covered by the Digital Personal Data Protection Act, 2023.

A verification habit for every AI answer



The analyst's rule: generate with AI, verify with data.

Here is why verification matters. Suppose an AI writes the following query for "average order value per region". It looks right, runs without error and returns plausible numbers, but it averages order **lines**, not **orders**:

Python

```
import duckdb, pandas as pd
con = duckdb.connect()
con.execute("CREATE TABLE superstore AS SELECT * FROM read_csv_auto('superstore_sales.csv')")
wrong = con.execute("SELECT region, ROUND(AVG(sales), 0) AS avg_order_value FROM superstore GROUP BY region ORDER BY region").df()
right = con.execute("""SELECT region, ROUND(AVG(order_total), 0) AS avg_order_value
                      FROM (SELECT region, order_id, SUM(sales) AS order_total FROM superstore GROUP
BY region, order_id) GROUP BY region ORDER BY region""").df()
pd.DataFrame({"region": wrong.region, "AI query (per line)": wrong.avg_order_value, "Correct (per order)": right.avg_order_value})
```

OUTPUT

	region	AI query (per line)	Correct (per order)
0	East	30,521	60,684
1	North	32,038	61,892
2	South	32,580	64,925
3	West	32,192	64,067

4 rows × 3 columns

The AI's version understates order value by roughly half, because an average order has about two lines. A one-line sanity check (total sales ÷ number of orders) would have caught it:

Python

```
tot = con.execute("SELECT SUM(sales) / COUNT(DISTINCT order_id) FROM superstore").fetchone()[0]
print(f"Total sales / distinct orders = {tot:,.0f}")
```

OUTPUT

Total sales / distinct orders = 62,942

COMMON MISTAKES Common AI-assistant failure modes

1. **Wrong grain:** counting lines as orders, double-counting after a join (Lesson 8 in SQL).
2. **Invented columns or functions:** the code references a field that does not exist.
3. **Silent assumptions:** it picks a date range, currency or definition without saying.
4. **Plausible but unverified numbers:** especially when asked to "estimate" or "recall" data it cannot see.
5. **Over-agreeing:** if you assert something wrong, it may follow you. Ask it to challenge your assumption.
6. **Stale knowledge:** new tool versions, prices, laws.

Responsible use

- **Accountability stays with you:** you sign off on the analysis, not the model.
- **Bias:** models mirror training data; check outputs for unfair assumptions about people or regions.
- **Transparency:** tell stakeholders when AI helped, and keep the prompt and verification notes.
- **Copyright and attribution:** do not present generated text or images as someone else's work; check licences.
- **Security:** treat prompts and uploaded files as data leaving your machine; beware prompt injection when an AI reads untrusted documents or web pages.

DO IT WITH AI Do it with AI

Pick a task you did earlier in this course (for example the SQL discount analysis). Ask an AI assistant to produce the answer from the schema, run its SQL, and compare with the result in this handbook. Where it differs, work out *why* (grain? date range? definition?). This single exercise teaches more about AI limits than any lecture.

Summary

KEY TAKEAWAYS

- AI $\supset$ ML $\supset$ deep learning $\supset$ generative AI; **agentic AI** = an LLM with goals, tools and a loop.
- LLMs predict the next token; they do not look things up unless given documents or tools. They can **hallucinate**, so verify every answer.
- Good prompts have **role, context, task, format and constraints**, use examples and step-by-step reasoning, and give the schema (not the sensitive data).
- Use templates for SQL, formulas, cleaning, insights, memos and interview practice; iterate.
- Never paste personal or confidential data into public tools; mask or sample, and follow your company policy.
- The analyst's loop: ask with context $\rightarrow$ run on real data $\rightarrow$ cross-check one number $\rightarrow$ state assumptions.

Quick quiz

6 questions · tap an answer to check it

1. Which statement best describes agentic AI?

- A** A model that only classifies images
- B** An LLM-based system that plans, uses tools and repeats steps to achieve a goal
- C** Any chatbot
- D** A rule-based macro

Answer: B. Agents add goals, tools and a decision loop on top of a language model.

2. Why can an LLM state wrong facts confidently?

- A** It lacks memory
- B** It generates plausible text from patterns and does not verify facts unless given sources or tools
- C** It is connected to the internet
- D** It always copies its training data

Answer: B. Fluency is not accuracy.

3. Which setting is best for generating SQL?

A High temperature

B Low temperature

C No context

D Maximum length

Answer: B. Low temperature reduces random variation for precise tasks.

4. What makes a prompt more reliable for analysis?

A Shorter wording

B Role, context (schema and definitions), a clear task, an output format and constraints

C Asking for "everything"

D Hiding the table names

Answer: B. Specifics reduce guesswork and make the answer checkable.

5. A teammate wants to paste a customer list with phone numbers into a public chatbot. Best advice?

A Go ahead

B Do not; mask or sample the data, or use an approved enterprise tool

C Delete the email column only

D Convert it to PDF first

Answer: B. Personal data should not go into tools not approved for it.

6. An AI query returns an "average order value" half the size of total sales ÷ orders. The likely error is:

- A Wrong chart
- B Averaging order lines instead of order totals (wrong grain)
- C Too many decimals
- D A missing index

Answer: B. Always sanity-check results against a simple independent calculation.

EXERCISE Prompt rewrite

Rewrite this weak prompt into a strong one and list three checks you will run on the answer: *"Tell me about churn in my telecom data."* The data is `telecom_churn.csv` with columns `customer_id`, `senior_citizen`, `tenure_months`, `contract`, `internet_service`, `payment_method`, `monthly_charges`, `total_charges`, `support_calls`, `churn` (Yes/No).

Show solution (worked answer)

Strong prompt: "Role: senior telecom data analyst. Context: table `telecom_churn(...columns...)`, one row per subscriber, `churn` = Yes/No. Task: find which contract types, tenure bands and support-call counts have the highest churn rate and quantify each. Format: a table (group, customers, churners, churn %), three bullets for a retention manager, and the SQL/pandas code used. Constraints: use only these columns, state assumptions, do not infer causes you cannot test, flag small groups (< 50 customers)." Checks: (1) recompute the overall churn rate by hand (count of Yes ÷ rows) and match it; (2) confirm each group's customer counts add up to the total; (3) spot-check one group with an independent query.

Q: Difference between AI, ML and generative AI? AI is the broad field; ML learns patterns from data to predict or classify; generative AI (a deep-learning subset) creates new content such as text, code and images.

Q: What is a hallucination and how do you reduce it? Confident but false output. Reduce it by grounding answers in provided documents or data (RAG), using tools that compute results, lowering temperature, asking for sources and assumptions, and verifying independently.

Q: How do you write a good prompt for a data task? State the role, context (schema, grain, definitions), a single clear task, the required output format, and constraints (use only given data, state assumptions); give examples; iterate.

Q: What is agentic AI? Systems where an LLM plans and executes multi-step tasks using tools, observes results and continues until a goal is met, with human oversight for risky actions.

AI with SQL: Text-to-SQL, Docker, Metabase & MinusX

Use AI to write, explain and debug SQL safely. Learn how text-to-SQL works and where it fails, give the model the right schema, protect your database with read-only access, install Docker, run Metabase on your own data, connect an AI assistant such as MinusX, and validate every generated query.

● DVA with AI · class 2

✦ Docker, Metabase and AI assistants, 2026

Intermediate

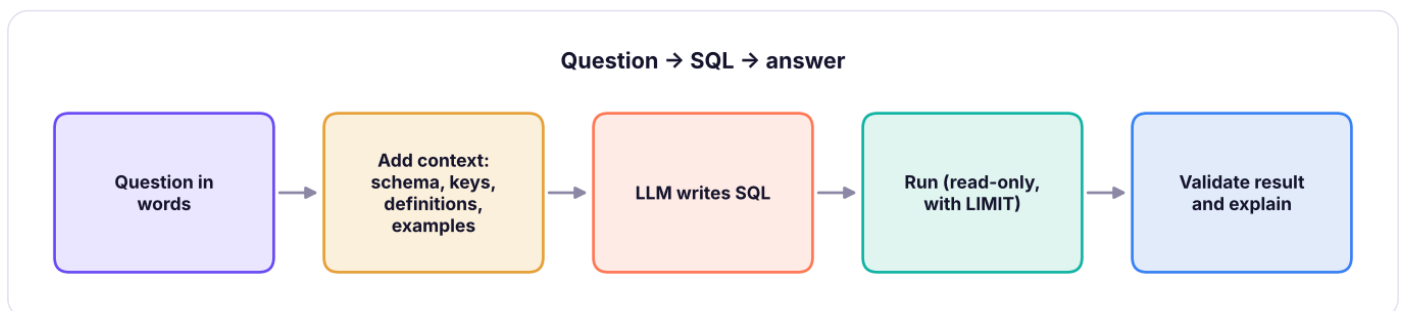
🕒 130 min

By the end of this lesson you can

- Explain how text-to-SQL works and why context (schema, definitions, examples) decides quality
- Write a schema-aware prompt and review generated SQL for the usual errors
- Install Docker and run Metabase locally; connect a database; build questions, models and dashboards
- Describe how an in-browser AI assistant such as MinusX works with Metabase, and its privacy trade-offs
- Apply guardrails: read-only roles, row limits, query review and result validation

What is text-to-SQL?

Text-to-SQL turns a business question in plain language into a SQL query: "Which region had the highest profit in 2025?" → `SELECT region, SUM(profit) ...`. LLMs are good at SQL syntax, but **correct SQL for your database** needs your context.



Text-to-SQL pipeline. Quality depends mostly on the context you supply, not on the model alone.

What goes wrong without context

Problem	Example	Prevention
Wrong tables/columns	Uses <code>customer_name</code> in <code>orders</code> , where it does not exist	Provide the schema (tables, columns, types, keys)
Wrong grain / fan-out	Joins orders to items and then <code>COUNT(*)</code> for "orders"	State the grain: "one row per order line"; add "count distinct order_id" rule
Wrong definitions	"Active customer", "net sales", "this year"	Provide a metric dictionary
Dialect mismatch	<code>LIMIT</code> instead of <code>TOP</code> in SQL Server	Name the database and version
Dates	Uses <code>YEAR(date) = 2025</code> (slow) or the wrong fiscal year	State calendar vs fiscal year and preferred patterns
Hallucinated values	<code>WHERE region = 'Northern'</code> when data says <code>North</code>	Provide sample values for categorical columns
Unsafe actions	<code>DELETE / DROP</code> suggestions	Read-only connection; review before running

A schema prompt that works

Text

```
You are a senior analytics engineer. Write SQL for Microsoft SQL Server 2022.

SCHEMA (all in dbo):
customers(customer_id PK, customer_name, segment, city, state, region)
  - segment in ('Consumer','Corporate','Home Office'); region in ('North','South','East','West')
orders(order_id PK, order_date DATE, ship_date DATE, ship_mode, customer_id FK → customers)
order_items(item_id PK, order_id FK → orders, product_id FK → products, quantity, discount, sales, profit)
  - one row per product line; sales and profit in rupees
products(product_id PK, product_name, category, sub_category, list_price)

DEFINITIONS:
- "Orders" = COUNT(DISTINCT orders.order_id). "Margin %" = SUM(profit) / SUM(sales).
- Financial year runs April-March. Use half-open date ranges ( $\geq$  start AND  $<$  next start).

QUESTION: For FY2025 (Apr 2025-Mar 2026), show margin % by category and by discount band (0, 1-20%, >20%).

RULES: Return only SQL; add comments; no SELECT *; list any assumptions after the query.
```

Notice the pieces: dialect, schema with keys, valid values, grain, definitions, date convention and output rules. This is the difference between a lucky answer and a reliable one.

TIP Keep a reusable "context file"

Save your schema, definitions and sample queries in one text file (or in an AI **Project** / custom instructions). Paste it into every session. Many BI tools (Metabase, Power BI Copilot, Snowflake Cortex, Looker) call this a **semantic layer** or "AI context": descriptions, synonyms and verified queries.

Reviewing AI-generated SQL: a checklist

1. **Does every table/column exist?** (Run it; fix any "invalid column name".)
2. **Is the grain right?** After each join, did rows multiply? Are distinct counts used for orders and customers?
3. **Are filters correct?** Date ranges half-open? Status filters? NULL handling?
4. **Do aggregates match the definition?** `SUM(profit)/SUM(sales)` not `AVG(margin)` .
5. **Does the total reconcile?** Compare with a simple query: `SELECT SUM(sales) FROM order_items` .
6. **Is it safe and cheap?** No writes; a `TOP` / `LIMIT` while testing; no accidental cross join.
7. **Is it readable?** CTEs, aliases and comments you could explain in an interview.

Let's see this in practice. A typical AI answer to "profit margin by discount band" that forgets `SUM(profit)/SUM(sales)` and averages row margins instead:

SQL

```
SELECT CASE WHEN discount = 0 THEN '0%' WHEN discount ≤ 0.2 THEN '1-20%' ELSE '>20%' END AS
discount_band,
       ROUND(AVG(profit / sales) * 100, 1) AS ai_margin_pct           -- average of row margins
(unweighted)
FROM order_items
GROUP BY CASE WHEN discount = 0 THEN '0%' WHEN discount ≤ 0.2 THEN '1-20%' ELSE '>20%' END
ORDER BY discount_band;
```

RESULT

discount_band	ai_margin_pct
0%	18.2
1-20%	2.0
>20%	-20.5

3 rows × 2 columns

SQL

```
SELECT CASE WHEN discount = 0 THEN '0%' WHEN discount ≤ 0.2 THEN '1-20%' ELSE '>20%' END AS
discount_band,
        ROUND(100.0 * SUM(profit) / SUM(sales), 1) AS correct_margin_pct      -- weighted: total profit /
total sales
FROM order_items
GROUP BY CASE WHEN discount = 0 THEN '0%' WHEN discount ≤ 0.2 THEN '1-20%' ELSE '>20%' END
ORDER BY discount_band;
```

RESULT

discount_band	correct_margin_pct
0%	17.5
1-20%	1.2
>20%	-20.6

3 rows × 2 columns

The two queries differ in every band (18.2% vs 17.5% and 2.0% vs 1.2% for the first two) because large order lines matter more in the weighted version than small ones. Which one the business wants must be written in the **definitions** part of your prompt. The checklist question "does the aggregate match the definition?" catches this.

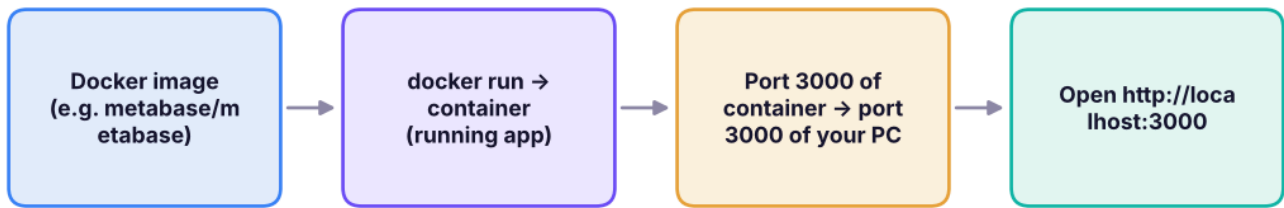
Guardrails for letting AI near a database

Guardrail	How
Read-only login	Create a database user with SELECT only on approved views (GRANT SELECT ON ...); never give AI tools db_owner
Expose views, not raw tables	A vw_sales_for_ai view with clean names, no personal data columns
Row and time limits	Timeouts, TOP 1000, query cost limits
Masking	Hide phone, email, ID columns; use dynamic data masking
Human review	Show the SQL before running it; log queries
Verified questions	Keep a library of approved queries for key metrics
No secrets in prompts	Connection strings and passwords stay in the tool's secret store

Docker: running tools in containers

Docker packages an application and everything it needs into a **container**, so it runs the same on any machine. Instead of installing Java, a database and an app by hand, you run one command.

From image to running app



An image is a read-only template; a container is a running instance of it. Many containers can run from one image.

STEP BY STEP Install Docker Desktop

1. Download **Docker Desktop** for Windows, Mac or Linux from the Docker website and install it. On Windows it uses **WSL 2** (Windows Subsystem for Linux); the installer enables it, and you may need to restart.
2. Start Docker Desktop and wait until it says "Engine running".
3. Open a terminal (PowerShell, Terminal) and check: `docker --version` and `docker run hello-world`.
4. Note your machine needs virtualisation enabled in BIOS and about 4 GB of free RAM for small tools.

Command	Meaning
<code>docker pull metabase/metabase:latest</code>	Download an image
<code>docker run -d -p 3000:3000 --name metabase metabase/metabase</code>	Run it in the background (-d), map port 3000, name it
<code>docker ps / docker ps -a</code>	List running / all containers
<code>docker logs -f metabase</code>	Follow its logs (see start-up progress)
<code>docker stop metabase / docker start metabase</code>	Stop / start again
<code>docker rm metabase</code>	Delete the container (not the image)
<code>docker images</code>	List downloaded images
<code>docker compose up -d</code>	Start everything defined in a <code>compose.yaml</code> file

Metabase: open-source BI for everyone

Metabase is an open-source business-intelligence tool: connect a database, ask questions with a point-and-click **question builder** (no SQL needed), or write SQL, then save charts into **dashboards**. It is popular in start-ups because non-technical colleagues can self-serve.

STEP BY STEP Run Metabase and connect your data

1. Start it: `docker run -d -p 3000:3000 --name metabase metabase/metabase`. First start takes a minute or two (check `docker logs -f metabase`).
2. Open **`http://localhost:3000`**, create the admin account, and choose **Add your data**.
3. Pick your database type (PostgreSQL, MySQL, SQL Server, BigQuery, Snowflake, SQLite files via drivers, CSV upload in newer versions), enter host, port, database name, username and password.
Use the read-only user. If the database also runs on your PC, use the host `host.docker.internal` from inside the container.
4. Metabase scans the tables. Browse them under **Browse data**.
5. **New → Question**: choose a table, pick columns, add **Filter**, **Summarize** (Sum of sales by region), choose a visualisation, **Save**.
6. **New → Dashboard**: add saved questions, add filters (date, region) and connect them to the question fields.

For anything you want to keep, run Metabase with a proper application database instead of the default embedded one, and mount a volume, for example:

YAML

```
# compose.yaml
services:
  metabase:
    image: metabase/metabase:latest
    ports: ["3000:3000"]
    environment:
      MB_DB_TYPE: postgres
      MB_DB_DBNAME: metabase
      MB_DB_PORT: 5432
      MB_DB_USER: metabase
      MB_DB_PASS: ${METABASE_DB_PASSWORD}      # from a .env file, never hard-coded
      MB_DB_HOST: postgres
    depends_on: [postgres]
  postgres:
    image: postgres:16
    environment:
      POSTGRES_USER: metabase
      POSTGRES_DB: metabase
      POSTGRES_PASSWORD: ${METABASE_DB_PASSWORD}
    volumes: ["pgdata:/var/lib/postgresql/data"]
volumes:
  pgdata:
```

Metabase concepts

Concept	Meaning
Question	A saved query (builder or SQL) with a visualisation
Model	A curated, cleaned table (a saved query with documented columns) for people to build on
Metric	A named, reusable calculation such as "Net revenue"
Dashboard	A page of questions with shared filters
Collection	Folders with permissions
Segment / filters	Reusable row filters
Permissions	Which groups see which databases and tables; row-level sandboxing (paid plans)
Metabot / AI features	Built-in AI assistance for questions and SQL in recent versions (availability depends on the edition and plan; check Metabase's current documentation)

Try it on the retail data: connect to your SQL Server RetailDB, then build three questions: *Sales by region*, *Monthly sales* (summarise by order_date: Month), and *Top 10 customers*. Then verify each one against this handbook's tables:

SQL

```
SELECT region, ROUND(SUM(sales), 0) AS sum_of_sales FROM superstore GROUP BY region ORDER BY sum_of_sales DESC;
```

RESULT

region	sum_of_sales
South	53,952,769
West	51,958,521
North	49,018,161
East	46,484,119

4 rows × 2 columns

MinusX: an AI assistant inside your BI tool

MinusX is an open-source AI assistant delivered as a **Chrome extension** that adds a chat sidebar to tools such as **Metabase** and **Jupyter notebooks**. Instead of copying your data into a chatbot, you work in Metabase and ask the sidebar things like *"Why did sales drop in March?"* or *"Write a query for monthly profit by category"*. According to its public description, it reads a simplified view of the page (metadata such as table and column names), then operates the tool by clicking, typing and writing queries, and lets you accept or reject its edits.

Aspect	How it behaves	What you should still do
Context	Sees the page, selected tables/columns and the query editor	Make sure the right database and tables are selected
Actions	Writes and runs SQL, edits questions, explains results	Read the SQL before accepting
Models	Supports several LLM providers (you may supply your own API key)	Check the provider's data policy
Privacy	Vendor says raw data is not sent; metadata and your prompts are	Verify the current policy; never expose sensitive tables
Cost	Free and paid plans; model API usage may cost	Check current pricing

STEP BY STEP Using an AI assistant with Metabase

1. Install the Chrome extension from the Chrome Web Store (search "MinusX") and pin it; open your Metabase tab and open the sidebar.
2. Add your model/API key in settings if required.
3. Select the database/table in Metabase so the assistant knows the context.
4. Ask a precise question ("Monthly sales and profit by category for 2025, as a line chart; sales = SUM(sales), profit = SUM(profit)").
5. Review the generated SQL, run it, check totals against a simple query, then save it as a question.
6. Ask follow-ups: "Explain the biggest drop", "Break it down by region", "Why is Furniture negative?"

WATCH OUT AI assistants and sensitive data

Anything the assistant can see, the model provider may receive. Use a **read-only role limited to non-sensitive views**, keep personal columns out of the connected schema, use an enterprise model agreement for company data, and keep a record of which prompts were run. If your company bans browser extensions that read pages, ask IT before installing any.

Other ways to use AI with SQL

Use	How
Explain a query	Paste the query (not the data) and ask for a line-by-line explanation and the business meaning
Optimise	Paste query + execution plan + table sizes; ask for rewrites and indexes; test each
Convert dialects	"Convert this SQL Server query to PostgreSQL; list functions that differ"
Generate test data	"Create INSERT statements for 20 fictional customers across four regions" (no real data)
Write documentation	Column descriptions, data dictionaries, comments
Data quality checks	"Write queries to find nulls, duplicates, orphans and out-of-range values for these tables"
Platform AI	Snowflake Cortex Analyst, BigQuery Gemini, Databricks Genie, SSMS Copilot, Azure Data Studio/VS Code Copilot, Fabric Copilot: all follow the same principle: context in, SQL out, you verify

2026 UPDATE Trend: the semantic layer

Teams that succeed with text-to-SQL do not rely on the raw schema: they define metrics (net sales, active customer) once in a **semantic layer** (dbt Semantic Layer, Cube, LookML, Power BI semantic models, Metabase models/metrics) and let AI query *that*. The AI then picks from governed metrics instead of reinventing definitions, which removes most wrong-number errors.

Summary

KEY TAKEAWAYS

- Text-to-SQL works when you provide **dialect, schema with keys, valid values, grain, metric definitions and date rules**.
- Review every query: columns exist, **grain** is right, filters correct, aggregates match the definition, totals reconcile, nothing writes.
- Protect the database with **read-only logins on curated views**, row limits and human review; never expose secrets or personal columns.
- **Docker** runs tools in containers (`docker run -d -p 3000:3000 metabase/metabase`); **Metabase** gives self-service questions, models and dashboards.
- An assistant such as **MinusX** adds an AI sidebar to Metabase; treat it as a fast junior analyst whose SQL you review, and check privacy terms.
- A governed **semantic layer** makes AI answers consistent.

Quick quiz

6 questions · tap an answer to check it

1. Why does giving the AI your schema and metric definitions improve results?

- A** It makes the model larger
- B** The model can use real table and column names and your definitions instead of guessing
- C** It speeds up the database
- D** It removes the need to review the SQL

Answer: B. Context is the main driver of text-to-SQL accuracy; review is still required.

2. Which command starts Metabase in a container on port 3000?

- A** `docker build metabase`
- B** `docker run -d -p 3000:3000 --name metabase metabase/metabase`
- C** `docker stop metabase`
- D** `metabase install`

Answer: B. `-d` runs detached and `-p 3000:3000` maps the port.

3. Which login is safest for an AI tool?

- A** The database owner
- B** A read-only user restricted to curated views
- C** The admin account
- D** An anonymous login

Answer: B. Least privilege limits damage if the AI makes a mistake.

4. A generated query averages profit/sales row by row to get a margin. Why is that risky?

- A** It is invalid SQL
- B** It is an unweighted average of ratios and may not match the business definition
 $\text{SUM}(\text{profit})/\text{SUM}(\text{sales})$
- C** It is slower
- D** It drops NULLs

Answer: B. Weighted vs unweighted aggregates differ; define the metric in the prompt.

5. What is a semantic layer?

- A** A type of chart
- B** A governed set of definitions for tables, metrics and relationships that BI and AI tools use
- C** A SQL function
- D** A backup system

Answer: B. It standardises metrics so every query uses the same logic.

6. What should you do before saving an AI-generated Metabase question?

- A** Nothing
- B** Read the SQL and reconcile at least one total against a simple independent query
- C** Share it with everyone
- D** Delete the database

Answer: B. Validation protects you from plausible but wrong numbers.

EXERCISE Schema prompt and review

(1) Write a schema prompt for the employees table (HR data: emp_id, gender, age, department, job_title, education, hire_date, years_at_company, monthly_income, job_satisfaction, overtime, performance_rating, city, attrition) including at least three definitions. (2) Ask any AI assistant for a query that returns attrition % by department and overtime. (3) Review the result with the checklist and reconcile the overall attrition % (25.9%) with a simple query.

Show solution (worked answer)

Definitions to include: "Attrition %" = employees with attrition = 'Yes' ÷ all employees; monthly_income is in rupees; "tenure band" = 0–2, 3–5, 6+ years. Reconciliation query: `SELECT ROUND(100.0 * SUM(CASE WHEN attrition = 'Yes' THEN 1 ELSE 0 END) / COUNT(*), 1) FROM employees;` should return 25.9. The weighted total from the grouped result (sum of leavers ÷ sum of employees across all groups) must also equal 25.9.

INTERVIEW CORNER Questions you may be asked

Q: How would you use AI to speed up SQL work safely? Provide schema, definitions and dialect; generate and explain queries; review for grain, filters and definitions; run read-only with limits; reconcile totals; keep verified queries in a library.

Q: What is Docker and why is it useful for data tools? A container platform that packages apps with their dependencies so the same setup runs anywhere; useful to spin up Metabase, databases or Jupyter in one command and share reproducible environments.

Q: What are the risks of connecting an AI assistant to a database? Data exposure to a model provider, destructive queries, wrong answers presented as facts, and prompt injection; mitigate with read-only access to curated views, masking, review and logging.

Q: What is Metabase? An open-source BI tool where users can ask questions with a visual builder or SQL, build dashboards and share them, with permission control by group.

AI in Excel & Office: Copilot, Gemini, Claude, Python in Excel & Agents

Use AI where analysts work every day. Work with Copilot in Excel (formulas, insights, charts, the COPILOT function), Gemini in Google Sheets and Excel add-ins, Claude and other assistants for formulas and files, Python in Excel, and office agents that perform multi-step tasks, and learn how to prompt, verify and stay safe.

● DVA with AI · class 3

✦ Microsoft 365 Copilot, Gemini and agent features as of 2026

Intermediate

🕒 120 min

By the end of this lesson you can

- Prepare a workbook so AI tools understand it (Tables, clear headers, no merged cells)
- Use Copilot and Gemini-style assistants to explain, write and debug formulas, create pivots and charts, and summarise data
- Use the Excel COPILOT function and Python in Excel for text, classification and analysis inside cells
- Describe what office agents do and what controls they need
- Verify AI-generated formulas and numbers with independent checks

The AI-in-Excel landscape

AI options for spreadsheet work

Built-in assistant

Copilot in Excel (Microsoft 365), Gemini in Google Sheets, Excel for the web insights: chat sidebar that reads your workbook

AI functions in cells

COPILOT() function, Gemini AI() function in Sheets: classify, summarise or extract per row

Add-ins & extensions

Gemini, Claude, ChatGPT and other add-ins that run in a task pane or via an API key

Python & code

Python in Excel (=PY()), Office Scripts, Google Apps Script; AI writes the code

Office agents

Multi-step assistants that clean data, build models, create decks and take actions across Office apps

External assistants

Upload the file to Claude, ChatGPT or Gemini; ask for analysis and formulas

Four ways AI shows up in spreadsheets.

Step 0: make your workbook AI-ready

AI assistants can only reason about data they can *see and parse*. The same rules that make pivot tables work (Excel Lesson 1) make AI work:

- 1. **Convert data to an Excel Table** (`Ctrl + T`) with a clear name (`tblSales`) and one header row.
- 2. **Descriptive column names** (`order_date` , `sales_inr`) instead of `Col1` ; add units.
- 3. No merged cells, blank rows or notes inside the data; keep totals outside.
- 4. One sheet per dataset; a small **Data dictionary** sheet that defines columns and metrics (AI reads it too).
- 5. Remove or mask sensitive columns before connecting an AI service.
- 6. Save to **OneDrive/SharePoint** (Copilot needs cloud-saved files with AutoSave on).

Copilot in Excel

Needs a Microsoft 365 Copilot licence and your data formatted as a Table or range with headers.

Task	Example prompt
Explain	"Explain what column H does and how it is calculated"
Formulas	"Add a column that shows profit margin %" / "Write a formula that looks up the manager's name from the Managers table"
Insights	"Show me insights about sales by region" → Copilot adds pivot tables and charts
Highlight	"Highlight rows where discount > 20% and profit is negative" (conditional formatting)
Sort / filter	"Sort by sales descending and filter to Technology"
Pivot & charts	"Create a pivot table of sales by category and year with YoY %"
Clean up	"Fix the inconsistent city names" / "Split full name into first and last name"
Analyse with Python	"Run a correlation between discount and profit and plot it" (uses Python in Excel)

STEP BY STEP

A safe Copilot workflow

1. Open your workbook (saved in OneDrive) → **Home** → **Copilot**.

2. Select the Table (or let Copilot detect it).

3. Ask one thing at a time, with column names: "Add a column `margin_pct = profit / sales`, formatted as a percentage."

4. **Read the proposed formula** and its explanation before clicking **Insert column**.

5. Check three rows by hand, and one total against a plain `SUM` .

6. Undo (`Ctrl + Z`) if wrong, and rephrase with more detail.

The COPILOT() function

Microsoft 365 also provides a `COPILOT` **function** that sends text and cell ranges to the model and returns results into cells that **spill and recalculate**:

Excel formula

```
=COPILOT("Classify this customer review as Positive, Neutral or Negative", D2:D100)
=COPILOT("Summarise the main complaint in 6 words or fewer", D2)
=COPILOT("Extract the city name from this address", A2)
```

Use it for text tasks (classify, summarise, extract, translate, tag) over **small to moderate** ranges, not for arithmetic: use normal formulas for numbers. Results can change between recalculations and each call consumes model capacity; availability, limits and licensing change, so check Microsoft's current documentation. Convert important results to values (copy → paste values) once checked.

Here is the same classify-sentiment idea applied to the product reviews dataset, with the (non-AI) ground-truth labels already in the file, so you can see how to *measure* an AI's accuracy:

Python

```
import pandas as pd
rev = pd.read_csv("product_reviews.csv")
print(rev.shape)
rev[["review_id", "product", "rating", "review_text", "sentiment"]].head(6)
```

OUTPUT

(1200, 5)

	review_id	product	rating	review_text	sentiment
0	1	Speaker	4	Very comfortable to use all day	positive
1	2	Smartwatch	5	Very comfortable to use all day	positive
2	3	Smartphone	4	Delivery was quick and packaging was neat. great value for money, totally worth it	positive
3	4	Earbuds	1	Battery drains too fast	negative
4	5	Laptop	2	Delivery was late and the box was damaged	negative
5	6	Laptop	2	Refund process is very slow. heats up a lot while charging	negative

6 rows × 5 columns

Python

```
# How would you evaluate an AI classifier? Compare its labels with a trusted label set.
# Here a deliberately weak keyword rule plays the role of "the AI" so you can see the evaluation
method.
# (These reviews are synthetic and templated, so real reviews will be much harder to classify.)
neg_words = ("stopped", "late", "poor", "crashing") # a deliberately small word list: an
imperfect classifier
pos_words = ("excellent", "great", "amazing", "premium")
def guess(t):
    t = t.lower()
    n = sum(w in t for w in neg_words); p = sum(w in t for w in pos_words)
    return "negative" if n > p else ("positive" if p > n else "neutral")
rev["pred"] = rev.review_text.apply(guess)
acc = (rev.pred == rev.sentiment).mean()
print(f"Agreement with trusted labels: {acc:.1%}\n")
print(pd.crosstab(rev.sentiment, rev.pred, rownames=["actual"], colnames=["predicted"]))
```

OUTPUT

Agreement with trusted labels: 56.0%

predicted	negative	neutral	positive
actual			
negative	204	206	0
neutral	0	175	0
positive	0	322	293

The cross-tab (a **confusion matrix**, ML track Lesson 15) shows *where* a classifier goes wrong. Do the same with an AI column: take 100 reviews you labelled yourself, compare, and only then trust it on 10,000.

Gemini in Google Sheets and with Excel

- **Gemini in Google Sheets** (Workspace plans): a side panel that can *create tables, write formulas, summarise, generate charts and analyse trends*, plus an **AI()** function to run a prompt per row (classify, summarise, extract) similar to COPILOT().
- **Gemini and other LLM add-ins for Excel**: some marketplaces offer add-ins or custom functions (for example =GEMINI("prompt", A2)) that call a model through **your own API key**. They are useful for batch text tasks, but you must protect the key, watch the **cost per call**, and avoid sending sensitive data.
- **Apps Script / Office Scripts**: AI can write the script; you review and run it.

COMMON MISTAKES API-key hygiene for add-ins

Never paste an API key into a shared workbook or a cell others can read; use the add-in's secure settings, set a **spending limit** on the key, and rotate it if exposed. A key in a shared file is a billing and security incident waiting to happen.

Claude, ChatGPT and other assistants with your files

Upload a CSV or XLSX to a general assistant that can **run code** (analysis/code-execution tools). It reads the file, writes Python, runs it, and shows charts and tables: closer to an analyst than a chatbot that only talks.

A good first message with a file:

Text

```
I've attached superstore_sales.csv (one row per order line, Jan 2023-Dec 2025, amounts in rupees).
1) Profile the data: row count, columns and types, missing values, duplicates, date range.
2) Compute total sales, profit and margin by year and category.
3) Find the three biggest drivers of low margin; quantify each.
4) Show your Python code, and tell me which assumptions you made.
Do not guess: if something cannot be computed from the file, say so.
```

Typical good behaviour: it prints the profile, shows code, makes charts. What you still do: spot-check numbers, make sure it used the right columns, and ask for the code so you can re-run it yourself. Assistants that run code produce **reproducible** analysis; those that only "think" may fabricate numbers.

Assistant strengths	Check
Writing and explaining formulas (even LET, LAMBDA, XLOOKUP, array formulas)	Test on sample rows
Cleaning plans and Power Query M code	Preview the result; keep a backup
Building DAX measures	Validate against a pivot
Summarising and drafting commentary	Verify each number quoted
Designing dashboards and chart choices	Compare with your audience's needs

Python in Excel

=PY(turns a cell into a Python editor (Formulas → Insert Python). It uses **pandas, matplotlib, seaborn, statsmodels and scikit-learn**, runs in Microsoft's secure cloud (no local Python needed), and reads Excel ranges with xl().

Python

```
df = xl("tblSales[#All]", headers=True) # Excel Table → pandas DataFrame
df.groupby("region")["profit"].sum().sort_values(ascending=False)
```

Ask Copilot: "Use Python to show the correlation between discount and profit and draw a scatter plot." Behind the scenes it writes the =PY() code. Check what it computed against this handbook's result:

Python

```
sup = pd.read_csv("superstore_sales.csv")
print(f"Correlation discount vs profit margin (lines): {sup.discount.corr(sup.profit /
sup.sales):.2f}")
print(sup.groupby(sup.discount.map(lambda d: f"{d:.0%}")).apply(lambda g: round(g.profit.sum() /
g.sales.sum() * 100, 1)).rename("margin_").to_string())
```

OUTPUT

Correlation discount vs profit margin (lines): -0.91	
discount	
0%	17.5

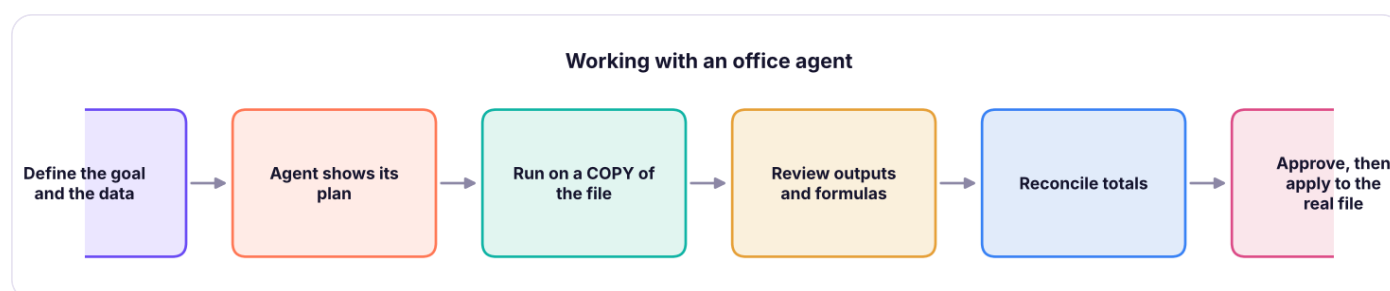
10%	5.9
15%	0.9
20%	-4.6
30%	-19.0
40%	-26.3

A strong negative correlation, and margin turning negative from a 20% discount upwards: the same story as the SQL and Power BI lessons, arrived at a third way. Cross-checking tools against each other is how you build trust.

Office agents: the next step

Agents go beyond answering: given a goal ("Clean this export, build a pivot by region, add a chart, and draft a three-slide summary"), they **plan, use the spreadsheet or Office apps as tools, check results and iterate**, then present the outcome for your approval. Microsoft, Google, Anthropic, OpenAI and others have all released agent modes for office work; names and capabilities change quickly, so evaluate any specific product (including whichever "Frontier"-style agent your organisation provides) against the questions below.

Question to ask about any agent	Why it matters
What can it read and write ? (files, mail, calendar, web)	Scope of risk
Does it show its plan and steps and ask before acting?	Oversight
Can I undo or review changes (version history)?	Recoverability
Where does data go (tenant, region, retention, training use)?	Compliance
How does it handle untrusted content (prompt injection in emails/web pages)?	Security
Does it produce formulas and sources or just pasted values?	Auditability
What does it cost per task ?	Budget



A safe agent workflow: small steps, visible plan, human approval at risky points, independent checks.

What AI does well and badly in spreadsheets

Strong	Weak / risky
Explaining unfamiliar formulas and files	Large-scale precise calculation (use formulas/Python)
Drafting formulas, DAX, SQL, M code, VBA	Knowing your business rules unless told
Text tasks: classify, summarise, extract, translate	Consistent results every time (non-deterministic)
Suggesting charts and structure	Hidden assumptions (date ranges, definitions)
Debugging errors (#N/A, #SPILL!)	Anything involving sensitive data in the wrong tool
Documentation and commentary	Replacing review and sign-off

A checklist before you trust an AI-built spreadsheet

1. Totals reconcile with the source (SUM of raw vs summary).
2. Row counts match (no lost or duplicated rows).
3. Formulas are **visible** (not pasted values) and use absolute references correctly.
4. Spot-check five random rows by hand.
5. Edge cases: blanks, zero, negative, text in number columns, duplicates.
6. Units and currency are labelled; dates are real dates.
7. Assumptions are written on the sheet.
8. A second tool or person confirms the headline number.

Summary

KEY TAKEAWAYS

- Prepare data as **Tables** with clear headers, no merged cells, and a data dictionary; save in the cloud for Copilot.
- **Copilot** explains and writes formulas, creates pivots, charts, highlights and cleans data; `COPILOT()` and Sheets' `AI()` apply prompts per row for text tasks.
- General assistants that can **run code** make reproducible analyses from uploaded files; always request the code and assumptions.
- **Python in Excel** (`=PY()`) brings pandas into cells; AI can write it, you verify it.
- **Agents** plan and act across apps: work on a copy, require visible plans, approvals, undo, audit trail and clear data-handling terms.
- Verify with reconciliation, spot checks, edge cases and a second tool; protect keys and sensitive data.

Quick quiz

6 questions · tap an answer to check it

1. What should you do first to help an AI understand your Excel data?

- A Merge headers
- B Convert it to a clearly named Excel Table with descriptive headers and no merged cells**
- C Colour the cells
- D Hide the columns

Answer: B. Structured, well-named data is easier for AI (and pivot tables) to use.

2. The COPILOT() function is best used for:

- A Precise financial arithmetic
- B Text tasks such as classifying, summarising or extracting over a range**
- C Replacing SUM
- D Sorting

Answer: B. Use normal formulas for numbers; use AI functions for language tasks.

3. Why ask an assistant to show its Python code?

- A To make the answer longer
- B So the analysis is reproducible and can be checked**
- C Because code is always correct
- D To avoid cleaning data

Answer: B. Visible code lets you verify and re-run the work.

4. Which is a sensible control for an office agent?

- A** Let it edit the live file without review
- B** Run it on a copy, review its plan and outputs, and keep version history
- C** Give it all passwords
- D** Disable logging

Answer: B. Human oversight and recoverability are essential.

5. An AI classifier labels reviews; how do you measure its quality?

- A** Read three results
- B** Compare its labels with a trusted labelled sample and look at accuracy and the confusion matrix
- C** Use temperature 1.0
- D** Ask it if it is right

Answer: B. Evaluate against ground truth, not self-reports.

6. Where should an API key for an Excel add-in be stored?

- A** In a visible cell
- B** In the add-in's secure settings with a spending limit
- C** In the sheet name
- D** In an email

Answer: B. Keys in shared files can be abused and billed to you.

EXERCISE AI-assisted analysis, verified

Using `superstore_sales.csv`: (1) ask an AI assistant with code execution to profile the data and compute margin by category and discount band; (2) write the pivot table version yourself; (3) compare the two and list any differences; (4) create three `COPILOT()`-style prompts (or `Gemini AI()` prompts) for a column of customer comments, and say how you would evaluate them on 50 labelled examples.

Show solution (worked answer)

Expected margins by discount band: 0% → 17.5%, 1–20% → 1.2%, >20% → -20.6% (overall 9.1%).

Differences usually come from averaging margins instead of total profit ÷ total sales, or from using order lines versus orders. Prompts: "Classify as Positive, Neutral or Negative (one word)"; "Extract the product mentioned (or 'none')"; "Summarise the complaint in under 8 words". Evaluation: label 50 comments yourself, run the prompt, compute accuracy and a confusion matrix, inspect errors, refine the prompt, and re-test on a fresh sample.

INTERVIEW CORNER Questions you may be asked

Q: How would you use AI tools to speed up Excel analysis responsibly? Prepare structured data, ask specific questions, review generated formulas and pivots, reconcile totals, document assumptions, keep sensitive data out of unapproved tools, and keep a human sign-off.

Q: What is Python in Excel? A feature that runs Python (pandas, matplotlib, scikit-learn) in cells via `=PY()`, in a secure cloud, reading Excel ranges with `xl()`, useful for analysis and charts that are hard with formulas.

Q: What is an AI agent in an office context and what are its risks? A system that plans and executes multi-step tasks using apps and files; risks include wrong actions, data leakage, prompt injection and lack of audit; mitigate with scoped permissions, approvals, copies and logs.

Q: How do you check an AI-generated formula? Test on a few rows including edge cases, compare totals with a simple independent calculation, read the formula's explanation, and make sure references and units are right.

The AI-Assisted Analyst Workflow, Agents & Governance (Capstone)

Combine everything into one repeatable workflow: frame a question, brief the AI, analyse with code, verify, visualise and write the story. See how an analyst agent works (plan, tools, observe, repeat) with a runnable toy example, learn MCP and tool-calling basics, and finish with governance, ethics and a telecom-churn capstone.

● DVA with AI · wrap-up

★ 2026

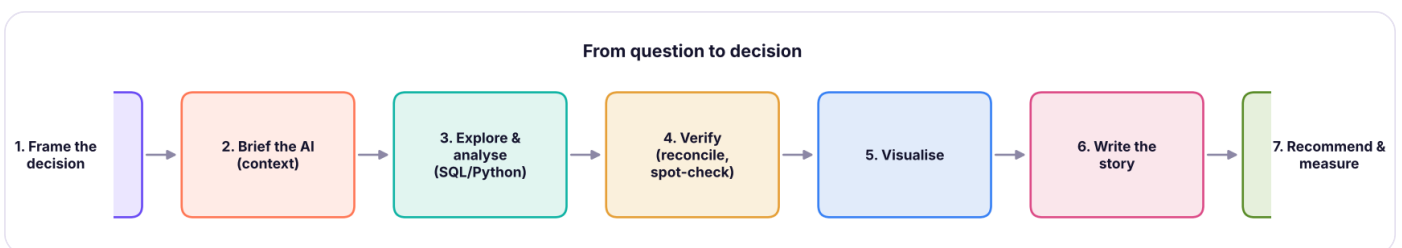
Intermediate

🕒 140 min

By the end of this lesson you can

- Apply a seven-step AI-assisted analysis workflow from question to recommendation
- Explain the agent loop, tool calling and the Model Context Protocol (MCP) at a conceptual level
- Build a tiny agent-style loop that plans, runs SQL tools and reports, and describe where an LLM would sit
- Design human-in-the-loop controls, logging and evaluation for AI analytics
- Complete an end-to-end churn analysis and explain it to a business audience

The AI-assisted analyst workflow



Seven steps. AI accelerates several of them, but the analyst owns the first and the last two.

Step	What you do	Where AI helps	Your responsibility
1. Frame	Decision, audience, metric definitions, constraints	Brainstorm questions, critique your framing	Own the decision question
2. Brief	Schema, definitions, sample rows, rules	Turn notes into a clean brief	No sensitive data; correct definitions
3. Analyse	Profile, clean, aggregate, model	Draft SQL/Python, explain code, suggest checks	Run it; read it
4. Verify	Reconcile totals, test edge cases, second method	Write test queries, find anomalies	Sign off the numbers
5. Visualise	Choose and label charts	Suggest chart types, write plotting code	Honest, readable charts
6. Write	Insight, caveat, action	Draft commentary, simplify language	Every claim traceable to a number
7. Recommend	Action, owner, metric, date	Pre-mortem: "how could this fail?"	Business judgement

Case: why do telecom customers leave?

CASE STUDY Brief

A telecom company has 3,000 subscribers in `telecom_churn.csv` (tenure, contract type, internet service, payment method, charges, support calls, churn Yes/No). The retention head asks: **Who is most likely to churn, and what should we do about it?** We will follow the workflow and use AI-style assistance where it helps, checking everything.

Steps 1–2: frame and brief

Text

```
Decision: choose which customers get a retention offer next quarter (budget: 300 offers).
Metric: churn rate = churners / customers in the group. Report groups with at least 50 customers only.
Schema: telecom_churn(customer_id, senior_citizen 0/1, tenure_months, contract {Month-to-month, One
year, Two year},
        internet_service {Fiber, DSL, No internet}, payment_method, monthly_charges, total_charges,
support_calls, churn {Yes, No})
Rules: use only this table; state assumptions; do not claim causes; show code.
```

Step 3: profile and analyse

Python

```
import pandas as pd, numpy as np
tc = pd.read_csv("telecom_churn.csv")
tc["churned"] = (tc.churn == "Yes").astype(int)
print(f"{len(tc):,} customers | overall churn {tc.churned.mean():.1%} | missing values: {int(tc.isna().sum().sum())} | duplicate IDs: {int(tc.customer_id.duplicated().sum())}")
tc.describe(include="all").T[["count", "unique", "top", "mean", "min", "max"]].round(1).head(10)
```

OUTPUT

3,000 customers | overall churn 25.7% | missing values: 0 | duplicate IDs: 0

	count	unique	top	mean	min	max
customer_id	3000	3000	T00001	NaN	NaN	NaN
senior_citizen	3,000	NaN	NaN	0.17	0	1
tenure_months	3,000	NaN	NaN	36.06	0	72
contract	3000	3	Month-to-month	NaN	NaN	NaN
internet_service	3000	3	Fiber	NaN	NaN	NaN
payment_method	3000	4	Card	NaN	NaN	NaN
monthly_charges	3,000	NaN	NaN	67.07	20	131.74
total_charges	3,000	NaN	NaN	2,418.32	19.58	8,724.35
support_calls	3,000	NaN	NaN	1.28	0	6
churn	3000	2	No	NaN	NaN	NaN

10 rows × 6 columns

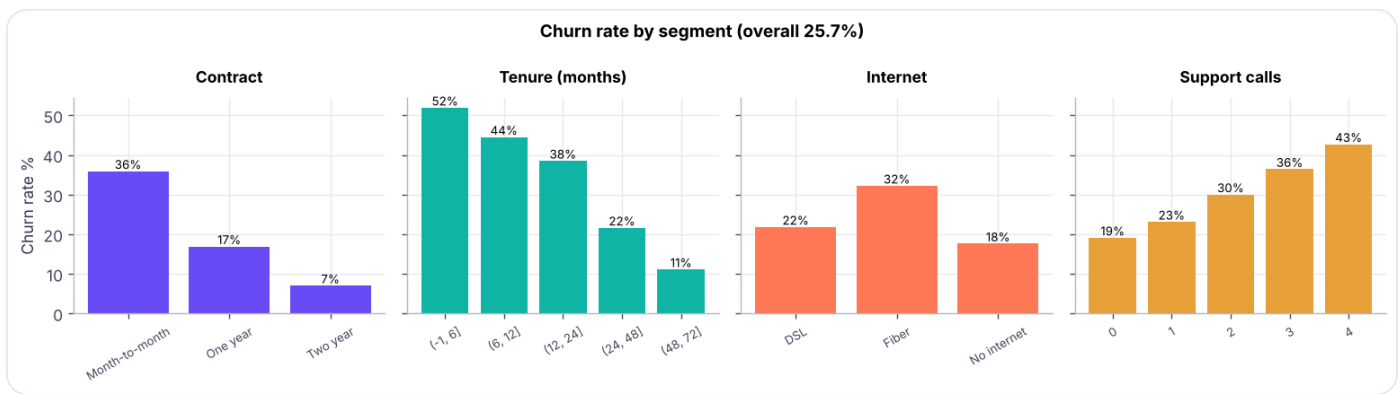
Churn by one factor at a time (the same GROUP BY thinking as SQL Lesson 7):

Python

```
def churn_by(col, bins=None, min_n=50):
    x = tc[col] if bins is None else pd.cut(tc[col], bins)
    g = tc.groupby(x, observed=True).agg(customers=("churned", "size"), churners=("churned", "sum"))
    g["churn_%"] = (g.churners / g.customers * 100).round(1)
    return g[g.customers >= min_n]
print(churn_by("contract").to_string()); print(); print(churn_by("internet_service").to_string());
print()
print(churn_by("support_calls").to_string()); print(); print(churn_by("tenure_months", bins=[-1, 6, 12,
24, 48, 72]).to_string())
```

OUTPUT

	customers	churners	churn_%
contract			
Month-to-month	1690	607	35.9
One year	739	125	16.9
Two year	571	40	7.0
internet_service			
DSL	1047	228	21.8
Fiber	1368	441	32.2
No internet	585	103	17.6
support_calls			
0	822	157	19.1
1	1101	254	23.1
2	649	195	30.0
3	294	107	36.4
4	110	47	42.7
tenure_months			
(-1, 6]	318	165	51.9
(6, 12]	238	106	44.5
(12, 24]	452	174	38.5
(24, 48]	1011	218	21.6
(48, 72]	981	109	11.1



Churn rate by contract type, tenure band, internet service and support calls. Month-to-month customers, new customers, fibre users and frequent callers churn most.

Step 4: verify before you believe

Python

```
# 1. Reconcile: group counts add up; 2. a second method gives the same answer; 3. check small groups
g = churn_by("contract", min_n=0)
print("Group customers sum to total :", g.customers.sum() == len(tc))
print("Group churners sum to total :", g.churners.sum() == tc.churned.sum())
import duckdb
sql = duckdb.sql("SELECT contract, COUNT(*) n, ROUND(100.0*AVG(CASE WHEN churn='Yes' THEN 1 ELSE 0
END),1) churn_pct FROM tc GROUP BY contract ORDER BY contract").df()
print("SQL cross-check:\n", sql.to_string(index=False))
print("\nCombinations with fewer than 50 customers are excluded from rankings; check:",
(tc.groupby(['contract', 'internet_service']).size() < 50).sum(), "such groups")
```

OUTPUT

```
Group customers sum to total : True
Group churners sum to total : True
SQL cross-check:
```

	contract	n	churn_pct
Month-to-month	1690	35.9	
One year	739	16.9	
Two year	571	7.0	

```
Combinations with fewer than 50 customers are excluded from rankings; check: 0 such groups
```

Associations are **not causes**: month-to-month customers may also be newer or on fibre. To look at several factors together, combine them:

Python

```
tc["tenure_band"] = pd.cut(tc.tenure_months, [-1, 12, 72], labels=["≤12 months", ">12 months"])
combo = tc.groupby(["contract", "tenure_band", observed=True]).agg(customers=("churned", "size"),
churn_pct=("churned", lambda s: round(s.mean() * 100, 1)))
combo[combo.customers ≥ 50]
```

OUTPUT

		customers	churn_pct
contract	tenure_band		
Month-to-month	≤12 months	306	67.0
	>12 months	1384	29.0
One year	≤12 months	151	33.8
	>12 months	588	12.6
Two year	≤12 months	99	15.2
	>12 months	472	5.3

6 rows × 4 columns

Steps 5–7: prioritise and recommend

With a budget of 300 offers, focus on the **highest-risk group that is also large enough to matter**. A simple **risk × size** table:

Python

```
seg = tc.groupby(["contract", "internet_service"], observed=True).agg(customers=("churned", "size"),
churn_pct=("churned", "mean"), avg_charges=("monthly_charges", "mean"))
seg["expected_churners"] = (seg.customers * seg.churn_pct).round(0)
seg["churn_pct"] = (seg.churn_pct * 100).round(1); seg["avg_charges"] = seg.avg_charges.round(0)
seg = seg[seg.customers ≥ 50].sort_values("expected_churners", ascending=False)
seg.head(6)
```

OUTPUT

		customers	churn_pct	avg_charges	expected_churners
contract	internet_service				
Month-to-month	Fiber	743	43.5	87	323
	DSL	617	30.8	61	190
	No internet	330	28.5	31	94
One year	Fiber	341	26.7	86	91
	DSL	252	11.1	61	28
Two year	Fiber	284	9.5	87	27

6 rows × 4 columns

KEY TAKEAWAYS

A recommendation a manager can act on

1. **Target** the top segments by *expected churners*, not only by churn rate: they hold most of the preventable loss.

2. **Offer** a discounted switch to a 1-year contract for month-to-month customers in their first year, and a service check for fibre customers with 2+ support calls.

3. **Measure** with an A/B test: random half of the target list gets the offer; compare churn after 90 days (the test design is in the ML track, hypothesis-testing lessons).

4. **Limit:** this analysis is descriptive; it does not prove that contracts *cause* churn or estimate the offer's effect.

How an AI agent does analysis

An agent turns the workflow above into a loop that an LLM drives:



The agent loop. The model decides the next action, a tool executes it, and the result goes back into the model's context.

Piece	Meaning	Analytics example
Goal	What to achieve	"Find the top churn drivers and draft a memo"
Tools	Functions the model can call	<code>run_sql(query)</code> , <code>run_python(code)</code> , <code>plot(spec)</code> , <code>search_docs(text)</code>
Memory / context	What it has seen so far	Previous queries and results
Planner (LLM)	Chooses the next action	"Check churn by contract next"
Guardrails	Limits	Read-only DB, max queries, no network, human approval for writes
Evaluation	How you know it worked	Unit tests on metrics, golden questions, human review

A toy agent you can run (no LLM)

To make the loop concrete, here is a miniature "analyst agent" in which a hard-coded planner stands in for the LLM. It has **one tool** (read-only SQL) and a rule: stop when it has found the top three churn drivers.

Python

```
import duckdb
db = duckdb.connect(); db.execute("CREATE TABLE tc AS SELECT * FROM
read_csv_auto('telecom_churn.csv')")

def run_sql(q):
    # TOOL: read-only SQL with a row limit
    assert q.strip().lower().startswith("select"), "only SELECT statements are allowed"
    return db.execute(q).df().head(20)

def planner(memory):
    # stands in for the LLM: decides the next tool call
    todo = [c for c in ["contract", "internet_service", "payment_method", "support_calls"] if c not in
memory]
    if todo and len(memory) < 4:
        c = todo[0]
        return f"SELECT {c} AS grp, COUNT(*) AS n, ROUND(100.0*AVG(CASE WHEN churn='Yes' THEN 1 ELSE 0
END),1) AS churn_pct FROM tc GROUP BY {c} HAVING COUNT(*) ≥ 50", c
    return None, None

memory, log = {}, []
while True:
    q, key = planner(memory)
    if q is None: break
    res = run_sql(q); memory[key] = res; log.append(f"ran query on '{key}' → best group:
{res.sort_values('churn_pct').iloc[-1].grp} ({res.churn_pct.max()}%)")
    print("\n".join(log))
    spread = {k: v.churn_pct.max() - v.churn_pct.min() for k, v in memory.items()}
    print("\nDrivers ranked by spread in churn rate (percentage points):")
    for k, v in sorted(spread.items(), key=lambda x: -x[1]): print(f" {k:18s} {v:5.1f}")
```

OUTPUT

```
ran query on 'contract' → best group: Month-to-month (35.9%)
ran query on 'internet_service' → best group: Fiber (32.2%)
ran query on 'payment_method' → best group: Net banking (28.5%)
ran query on 'support_calls' → best group: 4.0 (42.7%)
```

Drivers ranked by spread in churn rate (percentage points):

contract	28.9
support_calls	23.6
internet_service	14.6
payment_method	5.0

A real agent replaces `planner` with an LLM call that reads the memory and writes the next SQL, adds many more tools, and decides when to stop. Everything else (read-only guard, row limit, log) is the same safety scaffolding you should demand from any agent.

Tool calling and MCP

- **Tool (function) calling:** the model returns a structured request such as `{"tool": "run_sql", "arguments": {"query": "SELECT ..."}}`; your code runs it and sends the result back.
- **Model Context Protocol (MCP)** is an open standard for connecting AI applications to **tools and data sources** through a common interface, so one connector (a "server") for your database, files or SaaS app works with many AI clients. For analysts it means an assistant can query a **governed** database or semantic layer through an approved server instead of you pasting data around.

- **RAG (retrieval-augmented generation)** gives the model *documents* (metric definitions, past reports) to ground its answers (GenAI track).
- **Multi-agent systems** split work among specialised agents (a SQL agent, a chart agent, a reviewer); more moving parts mean more to test.

Evaluating AI analytics

What to evaluate	How
Correctness of numbers	A set of "golden questions" with known answers (total sales by year, churn by contract) run on every change
SQL quality	Execution accuracy (does the result match the reference result?), not just text similarity
Faithfulness of narrative	Every number in the text must appear in the tables; automatic check by extracting numbers
Safety	Attempted writes blocked, personal data not exposed, prompt-injection tests
Cost and latency	Tokens, tool calls and time per task
Usefulness	Do users make better decisions? Track adoption and feedback

Governance, ethics and your career

- **Policy:** know your organisation's approved tools, data-classification rules and retention.
- **Human in the loop** for decisions that affect people (credit, hiring, pricing, health): AI assists; accountable humans decide.
- **Fairness:** test whether results differ unfairly across groups (gender, age, region) before acting.
- **Explainability:** record the data, prompt, code and checks behind each insight; be able to re-run it.
- **Skills that stay valuable:** asking the right question, domain knowledge, data modelling, SQL/Python fluency to *read and verify* AI output, statistics and experimentation, storytelling and stakeholder management. The analysts who thrive are those who use AI for speed **and** can prove their answers.

2026 UPDATE What is changing in 2026

Analytics agents are moving into the BI tools themselves (Copilot in Power BI/Fabric, Gemini in Looker and BigQuery, Snowflake Cortex, Databricks Genie, Metabase AI features). Vendors compete on accuracy with governed semantic layers, so expect "ask a question of certified metrics" to become a standard feature. Learn the concepts here and re-check each product's current capabilities, pricing and data-handling terms before adopting it.

Summary

KEY TAKEAWAYS

- Follow a **seven-step workflow**: frame, brief, analyse, verify, visualise, write, recommend; AI accelerates but you own framing, verification and judgement.
- Always **verify**: reconcile totals, cross-check with a second method, exclude tiny groups, and remember associations are not causes.
- An **agent** = goal + LLM planner + tools + memory + guardrails in a loop; tool calling and **MCP** connect models to tools and governed data.
- Evaluate AI analytics with golden questions, execution accuracy, faithfulness checks, safety tests and cost tracking.
- Keep humans accountable, log the work, check fairness and follow data policy.

Quick quiz

6 questions · tap an answer to check it

1. In the agent loop, which part executes a SQL query?

A The planner

B A tool

C The memory

D The prompt

Answer: B. The LLM plans; tools perform actions such as running SQL.

2. What is MCP?

A A chart type

B An open protocol for connecting AI applications to tools and data sources

C A database engine

D A prompt format

Answer: B. The Model Context Protocol standardises how AI clients use external tools and data.

3. Why report churn only for groups with at least 50 customers?

- A To shorten the report
- B Small groups give unstable percentages that can mislead**
- C Because SQL requires it
- D To hide data

Answer: B. Small samples produce noisy rates.

4. Which statement about our churn findings is correct?

- A Contracts cause churn
- B Month-to-month customers churn more, but this is an association that needs testing before claiming cause**
- C Fibre causes churn
- D Support calls prove dissatisfaction

Answer: B. Descriptive analysis shows patterns, not causes; use an experiment to test an intervention.

5. What is a "golden question" set?

- A Questions that cost the most
- B Questions with known correct answers used to test an AI analytics system after each change**
- C A list of dashboards
- D Training data

Answer: B. They enable regression testing of answers.

6. Which control best limits damage if an agent writes a bad query?

- A** A longer prompt
- B** A read-only database role with row limits and logging
- C** A bigger model
- D** A louder warning

Answer: B. Technical guardrails beat instructions alone.

EXERCISE Run the workflow on a new question

Using `superstore_sales.csv`, follow the seven steps for: *"Which customer segment × region combinations should sales focus on next quarter?"* Write the brief, run the analysis (use `groupby` or SQL), verify with two methods, produce one chart and write a five-sentence recommendation that includes one limitation.

Show solution (worked answer)

Brief: define "focus" (for example profit, not just sales; minimum 100 order lines). Analysis: group by segment and region, compute sales, profit, margin and lines; filter groups with ≥ 100 lines. Verify: group totals equal the overall totals; repeat with SQL; check one group manually. Chart: bubble or bar of profit by segment × region. Recommendation example: "Focus on [top profit segment-region] (profit ₹X, margin Y%); protect [healthy group]; reduce discounts in [low-margin group]. Limitation: three years of data, no cost-to-serve or competitor information; profits include discounts that may not repeat."

Q: Walk me through how you use AI in your analysis workflow. I frame the decision, give the AI a clear brief with schema and definitions, use it to draft and explain SQL or Python, run everything myself, reconcile totals and cross-check with a second method, use AI for chart and narrative drafts, then I validate each number and take responsibility for the recommendation.

Q: What is an AI agent and how is it different from a chatbot? A chatbot responds once to a prompt; an agent pursues a goal across multiple steps by calling tools, observing results and deciding what to do next, so it needs stronger guardrails and evaluation.

Q: How do you stop an AI analytics tool from giving wrong numbers? Governed semantic layer, read-only access to curated data, golden-question tests, execution-based evaluation, human review of important outputs, and logging.

Q: A model says "contract type drives churn". What do you say? The data shows an association; to claim causation we would need an experiment or a causal method, controlling for tenure and other factors.

Python for Data Science

Python from zero: data types, control flow, data structures, functions, OOP, files, NumPy, pandas, visualisation, dates, SQL connections, EDA and DSA for data roles.

1. Introduction to Python: Why, Install & Jupyter Notebooks
2. Variables, Data Types, Type Conversion, print(), f-strings & input()
3. Operators, Strings, Indexing, Slicing & String Methods
4. Control Flow: if, elif, else, Nested Conditions & match
5. Loops: while, for, range, break, continue & Patterns
6. Lists: Creating, Slicing, Methods, Sorting & Comprehensions
7. Tuples & Sets: Immutable Records, Unpacking and Set Algebra
8. Dictionaries: Key-Value Data, Counting and Grouping
9. Functions: Reusable Code, Arguments, Scope and Docstrings
10. Comprehensions, Lambda, map/filter and Generators
11. Object-Oriented Python: Classes, Objects, Inheritance and Dataclasses
12. Files, Exceptions, JSON and CSV
13. Modules, Packages, Virtual Environments and Git
14. Debugging, Testing, Regular Expressions and Code Quality
15. NumPy: Fast Arrays, Vectorisation and Broadcasting
16. pandas I: Series, DataFrames, Reading Data, Selecting and Filtering
17. pandas II: Data Cleaning: Missing Values, Types, Duplicates, Text and Outliers
18. pandas III: GroupBy, Pivot Tables, Merge/Join and Reshape (melt/unpivot)
19. pandas IV: Dates, Time Series, Resampling and Rolling Windows
20. Data Visualisation I: Matplotlib and Choosing the Right Chart
21. Data Visualisation II: Seaborn Statistical Charts and Interactive Plotly
22. Exploratory Data Analysis (EDA) Case Study: Telecom Customer Churn
23. Python and SQL: sqlite3, DuckDB, SQLAlchemy and pandas.read_sql
24. APIs, JSON and Web Scraping with requests and BeautifulSoup
25. Performance and Bigger Data: Vectorisation, Memory, Parquet, Polars and DuckDB
26. DSA I: Big-O, Arrays, Strings, Hashing, Two Pointers and Sliding Window
27. DSA II: Stacks, Queues, Linked Lists, Recursion, Sorting and Binary Search
28. DSA III: Trees, Heaps, Graphs (BFS/DFS) and Dynamic Programming
29. Python Capstone: End-to-End Retail Sales Analytics Project

Introduction to Python: Why, Install & Jupyter Notebooks

Learn what programming and Python are, how Python compares with other languages, how code runs (interpreters, compilers, bytecode), the properties of Python, and set up your environment (Anaconda, VS Code, Jupyter, Google Colab). Write and run your first cells, with markdown, magic commands and good notebook habits.

● Python class 1

✦ Python 3.13, Jupyter, VS Code, Google Colab (2026)

Beginner

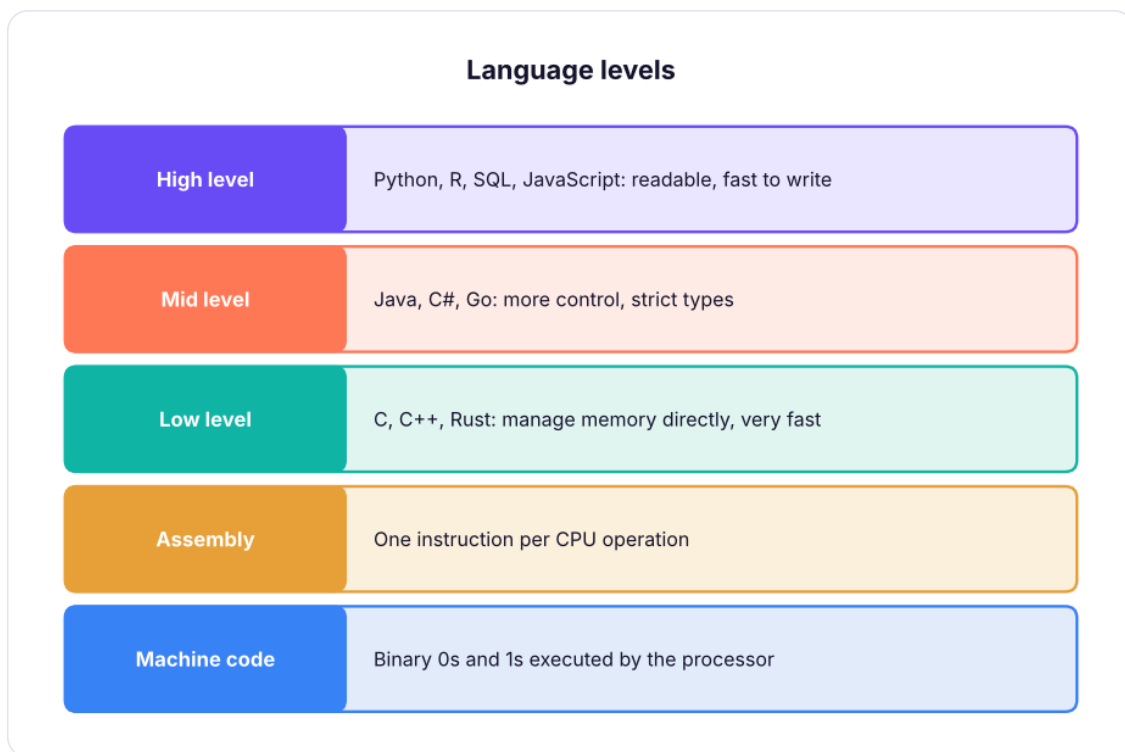
🕒 90 min

By the end of this lesson you can

- Explain what a programming language is and why Python is the default language for data work
- Describe the difference between compiled and interpreted languages and how Python runs your code
- Install Python and Jupyter (or use Google Colab) and run code in cells
- Use Markdown cells, keyboard shortcuts, magic commands and restart-and-run-all
- Follow good habits: naming, comments, reproducibility and virtual environments

What is programming?

A **program** is a set of precise instructions a computer follows. Computers only understand machine code (0s and 1s), so we write instructions in a **programming language** that humans can read, and a translator turns them into something the machine can run.



Languages sit on levels. High-level languages like Python are close to human language; low-level ones are close to the machine.

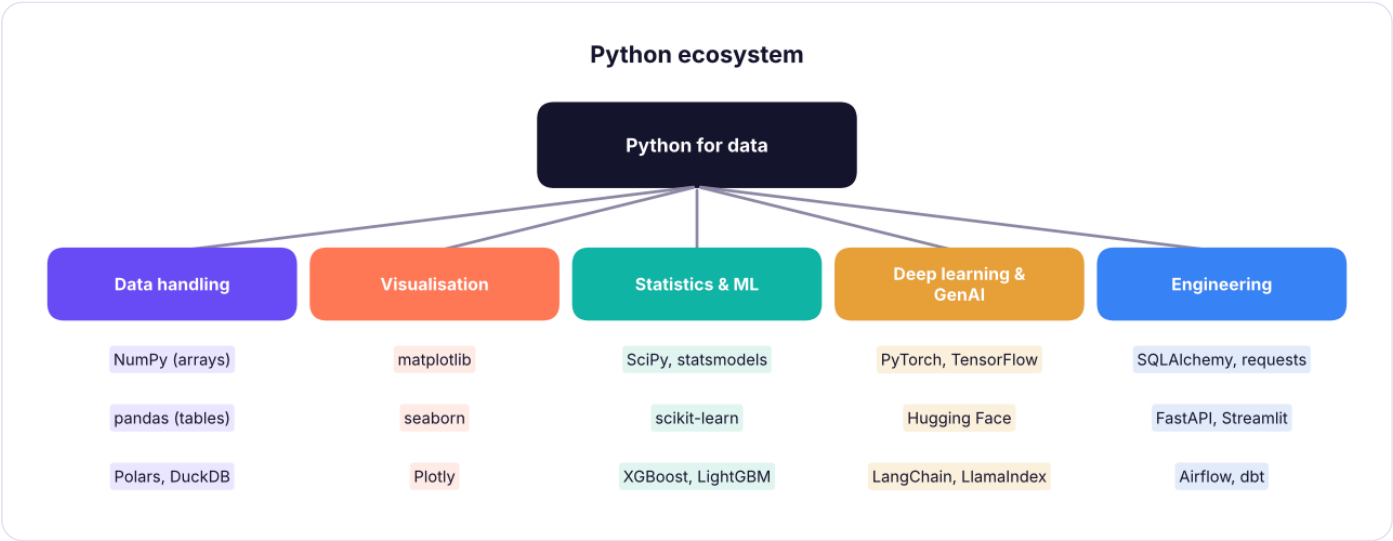
Compiled vs interpreted

	Compiled (C, C++, Rust, Go)	Interpreted (Python, JavaScript, R)
How it runs	Whole program translated to machine code before running	Translated and executed line by line while running
Speed	Usually faster at run time	Slower per operation, much faster to develop
Errors	Many caught before running	Some appear only when that line runs
Output	A separate executable file	Source file runs under an interpreter

Python is usually called **interpreted**. In reality CPython (the standard version) first compiles your code to **bytecode** (`.pyc` files) and then a virtual machine executes it. Python code that needs speed calls libraries written in C (NumPy, pandas), so data work is fast anyway.

—

Why Python for data?



The Python data ecosystem. You will learn each of these libraries in this track.

Property of Python	Meaning
Readable	Indentation and English - like keywords; fewer lines than most languages
Interpreted, dynamically typed	No compile step; variables take the type of the value they hold
High level	Memory is managed for you (garbage collection)
Multi - paradigm	Procedural, object - oriented and functional styles
Huge ecosystem	500,000+ packages on PyPI; pandas, scikit - learn, PyTorch lead data and AI
Open source and portable	Free; runs on Windows, Mac, Linux, cloud notebooks
Great glue	Calls SQL, APIs, C/C++ code, Excel, Spark

NOTE

Python versions

Use Python **3.11, 3.12 or 3.13** (the current stable series in 2026). Python 2 ended in 2020 and must not be used. Check your version with `python --version` .

Getting set up: three options

Option	Best for	Steps
Google Colab (free, in the browser)	Starting today, no installation, GPU for ML	Go to <code>colab.research.google.com</code> → New notebook → sign in with Google
Anaconda / Miniconda	Full local data - science setup with packages and environments	Install Anaconda Distribution → open Anaconda Navigator or a terminal → <code>jupyter lab</code>
Python + VS Code	Professional workflow, notebooks and scripts in one editor	Install Python from <code>python.org</code> (tick "Add to PATH") → install VS Code → add the Python and Jupyter extensions

STEP BY STEP

Local install with pip (any OS)

1. Install Python 3.12+ from `python.org` (Windows: tick **Add python.exe to PATH**).

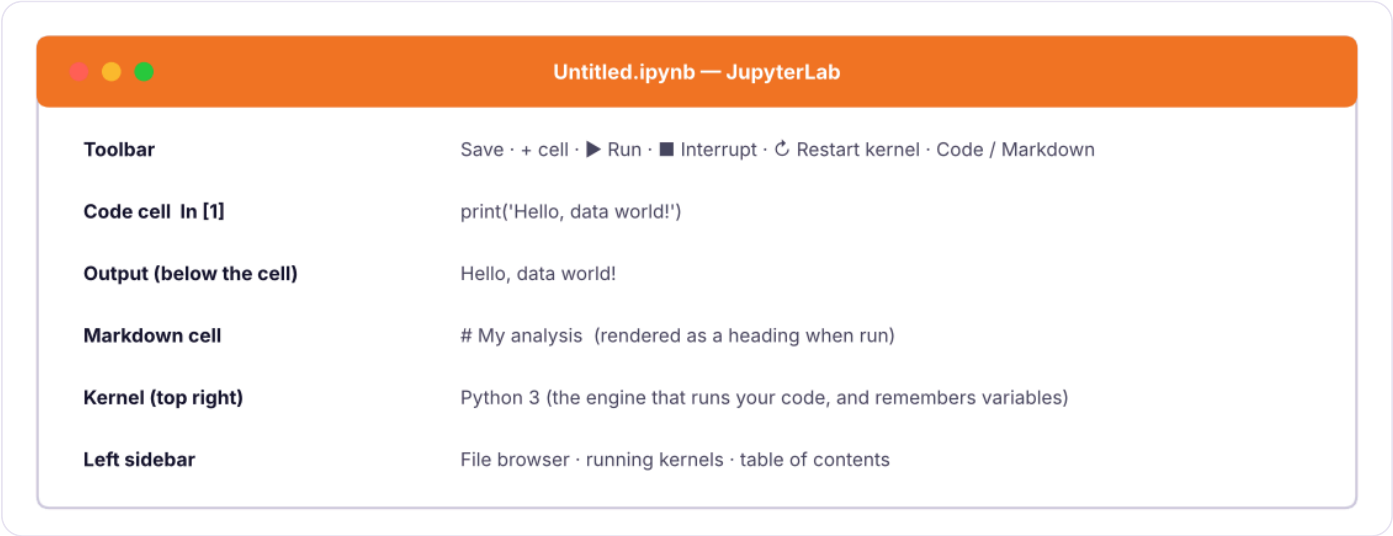
2. Open a terminal and create a **virtual environment** (an isolated set of packages for one project):
`python -m venv .venv` then activate it: Windows `.venv\Scripts\activate` ; Mac/Linux `source .venv/bin/activate` .

3. Install the course packages: `pip install jupyterlab numpy pandas matplotlib seaborn scikit-learn scipy statsmodels`

4. Start Jupyter: `jupyter lab` (opens in your browser).

Jupyter notebooks

A **Jupyter notebook** (`.ipynb`) mixes **code cells**, **Markdown cells** (text, headings, images) and **outputs** (tables, charts) in one document. It is the standard tool for exploring data and sharing analysis. The name comes from Julia, Python and R.



Anatomy of a Jupyter notebook.

The kernel

The **kernel** is the Python process behind the notebook. Variables you create stay in its memory until you restart it. Cells can be run in any order, which is flexible but dangerous: a notebook can look right while depending on a cell you deleted. **Before sharing, always Restart Kernel and Run All** to prove the notebook works top to bottom.

Essential shortcuts (Jupyter / Colab)

Action	Shortcut
Run cell, stay	Ctrl + Enter
Run cell, go to next	Shift + Enter
Insert cell above / below (command mode)	A / B
Delete cell (command mode)	D, D
Convert to Markdown / Code	M / Y
Command mode / edit mode	Esc / Enter
Comment / uncomment lines	Ctrl + /
Show function help	Shift + Tab (inside the brackets)
Autocomplete	Tab
Undo cell delete	Z (command mode)

Your first code

Python

```
print("Hello, data world!")
```

OUTPUT

Hello, data world!

Python is also a calculator:

Python

```
2 + 3 * 4
```

OUTPUT

14

Python

```
(2 + 3) * 4, 10 / 4, 10 // 4, 10 % 4, 2 ** 10
```

OUTPUT

```
(20, 2.5, 2, 2, 1024)
```

The **last expression** in a notebook cell is displayed automatically; to show several things use `print()`.

Python

```
name = "Varun"
learners = 25
print("Welcome to the course,", name)
print(f"This batch has {learners} learners")
```

OUTPUT

```
Welcome to the course, Varun
This batch has 25 learners
```

Comments

Python

```
# This is a comment: Python ignores everything after the hash
price = 120 # you can also put a comment at the end of a line
"""A triple-quoted string on its own line is often used
as a multi-line comment or docstring"""
```

Write comments to explain **why**, not what. `x = x + 1 # add one` is noise; `# compensate for the header row` is useful.

Errors are normal

Every programmer sees errors daily. Learn to read them from the **bottom line up**:

Python

```
print("Total:", total_amount)
```

OUTPUT

```
NameError: name 'total_amount' is not defined
```

`NameError: name 'total_amount' is not defined` says exactly what happened: we used a name we never created. Common beginner errors:

Error	Usual cause
<code>NameError</code>	Typo, or the cell that defines the variable has not been run
<code>SyntaxError</code>	Missing bracket, quote or colon; wrong indentation
<code>TypeError</code>	Using a value of the wrong type (<code>"5" + 3</code>)
<code>ValueError</code>	Right type, wrong value (<code>int("abc")</code>)
<code>IndexError</code> , <code>KeyError</code>	Asking for a position or key that does not exist
<code>FileNotFoundError</code>	Wrong path or the file is not where you think

TIP How to ask for help

Copy the **last line** of the error, search it, or paste the code and the error into an AI assistant (without secrets) and ask: *"What does this error mean and how do I fix it? Explain step by step."* Then fix it **yourself** and re-run: you learn by doing.

Markdown cells

Change a cell to **Markdown** to write notes next to your code:

MARKDOWN

```
# Title
## Section
Some bold and italic text, a [link](https://example.com), and a list:
- item one
- item two

| Column | Value |
|---|---|
| A | 1 |
```

Good notebooks tell a story: a title, the question, the data, the steps with short explanations, and a conclusion.

Magic commands and useful tools

Jupyter adds special commands beginning with `%` :

Command	Purpose
<code>%timeit expr</code>	Time a short expression precisely
<code>%%time</code>	Time a whole cell
<code>%pwd</code> , <code>%ls</code>	Current folder, list files
<code>%matplotlib inline</code>	Show charts in the notebook (default now)
<code>%who</code>	List variables
<code>!pip install package</code>	Run a shell command (install a package) from a cell

Reproducibility habits

1. **Folder per project:** `data/` , `notebooks/` , `src/` , `README.md` .
2. **Virtual environment per project** and a `requirements.txt` (`pip freeze > requirements.txt`).
3. **Use relative paths** (`data/sales.csv`), not `C:\Users\you\Desktop\...` .
4. **Set random seeds** (`np.random.seed(42)`) so results repeat.
5. **Restart and Run All** before sharing; keep notebooks short and focused.
6. **Version control:** commit to Git; clear outputs of big notebooks.
7. **Do not commit** data with personal information or secrets (API keys); use environment variables.

DO IT WITH AI Do it with AI

Ask an assistant: *"I'm a beginner. Give me a 7-day practice plan for Python with 20 minutes a day, each day with three small exercises and answers I can check."* Use it to **practise**, not to skip the struggle: type the code, break it on purpose, and read the errors.

Summary

KEY TAKEAWAYS

- A program is a set of instructions; Python is a **high-level, interpreted, readable** language with a huge data ecosystem (NumPy, pandas, scikit-learn, PyTorch).
- CPython compiles source to bytecode and runs it on a virtual machine; speed-critical libraries use C underneath.
- Start with **Google Colab** or install **Anaconda** or **Python + VS Code**; isolate projects with **virtual environments**.
- A Jupyter notebook mixes code, Markdown and output; the **kernel** holds state, so *Restart and Run All* before sharing.
- Read error messages from the bottom; learn the common error types.
- Use clear comments, relative paths, seeds and version control for reproducible work.

Quick quiz

6 questions · tap an answer to check it

1. Which best describes Python?

- A** A low-level compiled language
- B** A high-level, interpreted, general-purpose language with a rich data ecosystem
- C** A database language
- D** A spreadsheet tool

Answer: B. Python is readable, dynamically typed and widely used for data.

2. What does the kernel do in a notebook?

- A** Formats Markdown
- B** Runs your code and keeps variables in memory
- C** Stores the file
- D** Draws charts only

Answer: B. The kernel is the Python process behind the notebook.

3. Why run "Restart kernel and Run All" before sharing a notebook?

- A** To make it smaller
- B** To confirm the notebook works from top to bottom without hidden state
- C** To delete output
- D** To change the Python version

Answer: B. Cells can be run out of order; this test removes that risk.

4. What does `NameError: name 'x' is not defined` mean?

- A x is a bad name
- B Python has no variable called x yet (typo or the defining cell was not run)
- C x must be a number
- D A syntax mistake

Answer: B. You used a name before creating it.

5. Which command installs a package from inside a notebook cell?

- A `install pandas`
- B `!pip install pandas`
- C `%pip.exe pandas`
- D `import install`

Answer: B. A leading `!` runs a shell command.

6. What is a virtual environment for?

- A Faster internet
- B Keeping the packages of one project separate from others
- C Running Python 2
- D Backing up data

Answer: B. Isolated environments avoid version conflicts.

EXERCISE Set up and run

(1) Open Colab or Jupyter and create a notebook called `day1.ipynb`. (2) Add a Markdown cell with a title and your goals. (3) In code cells, print your name, compute the area of a rectangle 12×7.5 , and print the Python version (`import sys; print(sys.version)`). (4) Cause a `NameError` on purpose and fix it. (5) Restart and Run All.

Show solution (worked answer)

Python

```
print("Asha Rao")
length, width = 12, 7.5
print("Area:", length * width)           # 90.0
import sys; print(sys.version)
print(undefined_name)                   # NameError: name 'undefined_name' is not defined
undefined_name = 5; print(undefined_name)
```

INTERVIEW CORNER Questions you may be asked

Q: Why is Python popular for data science? Readable syntax, a huge ecosystem (NumPy, pandas, scikit-learn, PyTorch), strong community, easy integration with SQL, APIs and cloud, and notebooks for exploration.

Q: Is Python compiled or interpreted? Both: CPython compiles source to bytecode, which the Python virtual machine interprets. It is called interpreted because there is no separate manual compile step.

Q: What is a virtual environment? An isolated Python installation with its own packages for one project (created with `venv` or `conda`), preventing dependency conflicts.

Q: Notebook vs script? Notebooks are interactive and great for exploration and communication; scripts (`.py`) are better for production code, testing and automation.

Variables, Data Types, Type Conversion, print(), f-strings & input()

Store and name values with variables, meet Python's core data types (int, float, str, bool, None, complex), convert between types, format output with print(), f-strings and .format(), and read user input with input() and sys.argv.

● Python class 2

✦ Python 3.13

Beginner

🕒 110 min

By the end of this lesson you can

- Create variables and follow Python naming rules and conventions
- Identify and check types with type() and isinstance(); explain mutable vs immutable
- Convert between int, float, str and bool, and predict what int(), float(), bool() and str() do
- Format output with print() options, f-strings (including number formats) and str.format()
- Read and validate input with input(), and avoid float rounding surprises

Variables

A **variable** is a name that refers to a value stored in memory. You create it by assigning with `=`: no declaration, no type needed.

Python

```
price = 1250          # an int
tax_rate = 0.18       # a float
product = "Laptop"    # a str
in_stock = True       # a bool
total = price * (1 + tax_rate)
print(product, "costs", total)
```

OUTPUT

```
Laptop costs 1475.0
```

Names point to objects



A variable is a label attached to an object. Re-assigning moves the label; it does not change the old object.

Naming rules and conventions

Rule	Example
Letters, digits and <code>_</code> only; cannot start with a digit	<code>sales_2025</code> ✓, <code>2025_sales</code> ✗
Case-sensitive	<code>Total</code> and <code>total</code> are different
Cannot be a keyword (<code>if</code> , <code>for</code> , <code>class</code> , <code>None</code> , <code>True</code> ...)	<code>class</code> ✗, <code>class_name</code> ✓
Convention: <code>snake_case</code> for variables and functions, <code>PascalCase</code> for classes, <code>UPPER_CASE</code> for constants	<code>monthly_income</code> , <code>CustomerOrder</code> , <code>GST_RATE</code>
Use meaningful names	<code>avg_order_value</code> not <code>aov2</code> or <code>x</code>
Avoid shadowing built-ins	Do not name variables <code>list</code> , <code>sum</code> , <code>max</code> , <code>str</code> , <code>id</code> , <code>input</code>

Python

```
a = b = c = 0           # chained assignment
x, y, z = 1, 2.5, "three" # multiple assignment (tuple unpacking)
x, y = y, x             # swap without a temp variable
print(a, b, c, x, y, z)
```

OUTPUT

```
0 0 0 2.5 1 three
```

Data types

Everything in Python is an **object** with a **type**. Check it with `type()` .

Type	Name	Example	Notes
int	Integer	42 , -7 , 1_000_000	Unlimited size
float	Decimal number	3.14 , 2e-3	64-bit binary floating point
str	Text	"Pune" , 'Pune'	Immutable sequence of characters
bool	Boolean	True , False	Subclass of int (True = 1)
NoneType	"nothing"	None	Used for missing/absent values
complex	Complex number	3 + 4j	Rare in data work
list , tuple , dict , set	Collections	Lessons 8–10	

Python

```
values = [42, 3.14, "Pune", True, None, 3 + 4j, [1, 2], (1, 2), {"a": 1}, {1, 2}]
for v in values:
    print(f"{str(v):10s} → {type(v).__name__}")
```

OUTPUT

```
42          → int
3.14        → float
Pune        → str
True        → bool
None        → NoneType
(3+4j)      → complex
[1, 2]      → list
(1, 2)      → tuple
{'a': 1}    → dict
{1, 2}      → set
```

Python

```
print(isinstance(42, int), isinstance(True, int), isinstance(3.0, (int, float))) # True True True
print(type(10 / 2), type(10 // 2))      # / always returns float; // with ints returns int
```

OUTPUT

```
True True True
<class 'float'> <class 'int'>
```

Mutable vs immutable

- **Immutable:** cannot be changed after creation: int , float , str , bool , tuple , frozenset . "Changing" creates a new object.
- **Mutable:** can be changed in place: list , dict , set .

Python

```
s = "data"
try:
    s[0] = "D"                # strings are immutable
except TypeError as e:
    print("Error:", e)
s2 = "D" + s[1:]              # build a new string instead
print(s2, "| id changed:", id(s) != id(s2))
```

OUTPUT

```
Error: 'str' object does not support item assignment
Data | id changed: True
```

Numbers: int and float

Python

```
big = 12_345_678_901_234_567_890 * 10    # ints never overflow
print(big)
print(0.1 + 0.2)                        # floats are approximate!
print(0.1 + 0.2 == 0.3, abs((0.1 + 0.2) - 0.3) < 1e-9)
print(round(2.675, 2), round(0.5), round(1.5), round(2.5)) # banker's rounding: halves go to even
```

OUTPUT

```
123456789012345678900
0.30000000000000004
False True
2.67 0 2 2
```

WATCH OUT Floating-point surprises

Computers store decimals in binary, so `0.1 + 0.2` is `0.30000000000000004`. For **money** use `decimal.Decimal` (or integers of paise); for comparisons use `math.isclose(a, b)` or `abs(a - b) < tolerance`. Python's `round()` rounds halves to the nearest even number (`round(2.5) = 2`), unlike Excel's `ROUND`.

Python

```
from decimal import Decimal
print(Decimal("0.1") + Decimal("0.2"))    # exact decimal arithmetic
import math; print(math.isclose(0.1 + 0.2, 0.3))
```

OUTPUT

```
0.3
True
```

Booleans and None

Python

```
print(True + True, 5 > 3, 5 == 5.0, "a" < "b", not False)
print(bool(0), bool(0.0), bool(""), bool([]), bool(None))
print(bool(1), bool(-3), bool("0"), bool([0]))
print(None is None, type(None))
```

all False: "falsy" values
all True: anything else is "truthy"

OUTPUT

```
2 True True True True
False False False False False
True True True True
True <class 'NoneType'>
```

Falsy values: `0`, `0.0`, `""`, `[]`, `{}`, `()`, `set()`, `None`, `False`. Everything else is truthy. `None` represents "no value" (like SQL NULL); test with `is None`, never `= None`.

Type conversion

Convert with the type's name: `int()`, `float()`, `str()`, `bool()`, `list()` ...

Python

```
print(int("42"), int(3.99), int(-3.99), int(True))
print(float("3.5"), float(7), float("1e3"))
print(str(123), str(3.0), str([1, 2]))
print(int("0b10", 2), int("ff", 16), bin(10), hex(255))
print(round(3.7), round(3.14159, 2), int(round(3.7)))
```

int() truncates towards zero
other bases

OUTPUT

```
42 3 -3 1
3.5 7.0 1000.0
123 3.0 [1, 2]
10 255 0b1010 0xff
4 3.14 4
```

Python

```
print("5" + 3)
```

TypeError: can only concatenate str to str

OUTPUT

```
TypeError: can only concatenate str (not "int") to str
```

Python

```
print("5" + str(3), int("5") + 3, "5" * 3)      # explicit conversion; repetition for strings
```

OUTPUT

53 8 555

Python

```
int("abc")                                     # ValueError: invalid literal for int()
```

OUTPUT

`ValueError: invalid literal for int() with base 10: 'abc'`

Conversion	Result	Note
<code>int("42")</code>	42	String must look like an integer
<code>int("3.7")</code>	ValueError	Convert to float first: <code>int(float("3.7"))</code>
<code>int(3.99)</code>	3	Truncates (not rounds)
<code>float("abc")</code>	ValueError	
<code>str(None)</code>	<code>'None'</code>	
<code>bool("False")</code>	True	Any non-empty string is True!
<code>list("abc")</code>	<code>['a', 'b', 'c']</code>	

Implicit conversion happens in mixed arithmetic: `int + float → float` , `bool + int → int` .

Python

```
print(1 + 2.0, True + 1, type(1 + 2.0).__name__)
```

OUTPUT

3.0 2 float



print(): showing output

Python

```
print("Order", 1001, "shipped")           # items separated by a space
print("Order", 1001, "shipped", sep=" | ") # change the separator
print("Loading", end="...")               # change the line ending (no newline)
print("done")
print("Line one\nLine two\tTabbed")       # escape sequences: \n newline, \t tab
print('He said "hi"', "It's fine", 'It\'s fine') # quotes inside strings
print(r"C:\new\table")                    # raw string: backslashes kept
print("""A multi-line
string with triple quotes""")
```

OUTPUT

```
Order 1001 shipped
Order | 1001 | shipped
Loading...done
Line one
Line two      Tabbed
He said "hi" It's fine It's fine
C:\new\table
A multi-line
string with triple quotes
```

f-strings: the modern way to format

Put `f` before the quotes and `{expression:format}` inside:

Python

```
name, qty, price, share = "Asha", 3, 1234.5678, 0.2345
print(f"{name} bought {qty} items")
print(f"Price: {price:.2f}")           # 2 decimals
print(f"Price: {price:,.2f}")          # thousands separator
print(f"Share: {share:.1%}")           # percentage
print(f"ID: {qty:03d}")                 # zero-padded to 3 digits
print(f"[{name:<10}] [{name:>10}] [{name:^10}]") # left, right, centre in width 10
print(f"{price:>12,.1f}|")              # right-aligned width 12
print(f"{qty * price:.2f} {qty = }")    # expressions work; {x = } prints name and value
print(f"{1234567:,} {255:#x} {255:b} {0.000123:.2e}")
```

OUTPUT

```
Asha bought 3 items
Price: 1234.57
Price: 1,234.57
Share: 23.4%
ID: 003
[Asha      ] [      Asha] [   Asha   ]
      1,234.6|
```

```
3,703.70  qty = 3
1,234,567  0xff  11111111  1.23e-04
```

Format spec	Meaning
<code>.2f</code>	2 decimal places (fixed)
<code>,</code>	Thousands separator
<code>.1%</code>	Percentage with 1 decimal
<code>05d</code>	Integer padded with zeros to width 5
<code><10</code> , <code>>10</code> , <code>^10</code>	Left / right / centre in width 10
<code>e</code>	Scientific notation
<code>x</code> , <code>b</code> , <code>o</code>	Hex, binary, octal

.format() and % (older styles)

Python

```
print("{} bought {} items at {:.1f}".format("Asha", 3, 12.345))
print("{1} before {0}".format("B", "A"))
print("{name} paid {amt:,.0f}".format(name="Ravi", amt=1234567))
print("%s bought %d items at %.1f" % ("Asha", 3, 12.345))    # legacy, still seen in old code
```

OUTPUT

```
Asha bought 3 items at 12.3
A before B
Ravi paid 1,234,567
Asha bought 3 items at 12.3
```

Prefer **f-strings** in new code; you will meet `.format()` and `%` in older code and logs.

Reading input

`input()` reads **text** from the keyboard and always returns a `str` :

Python

```
name = input("Your name: ")          # waits for the user
age = int(input("Your age: "))        # convert to a number
height = float(input("Height in metres: "))
print(f"Hello {name}, you are {age} and {height:.2f} m tall.")
```

Because this handbook runs without a keyboard, here is the same logic with the typed values supplied in code:

Python

```
typed = ["Asha", "29", "1.62"]          # imagine the user typed these three answers
name = typed[0]; age = int(typed[1]); height = float(typed[2])
print(f"Hello {name}, you are {age} and {height:.2f} m tall. In ten years: {age + 10}.")
```

OUTPUT

Hello Asha, you are 29 and 1.62 m tall. In ten years: 39.

Validate input

Python

```
while True:
    text = input("Enter quantity (1-100): ")
    if text.isdigit() and 1 ≤ int(text) ≤ 100:
        quantity = int(text); break
    print("Please type a whole number between 1 and 100.")
```

Other ways to get values into a program: **command-line arguments** (`sys.argv` , `argparse`), **files** (Lesson 13), **databases and APIs** (later lessons), and **environment variables** (`os.environ["API_KEY"]`) for secrets.

Useful built-in functions

Python

```
nums = [4, 8, 15, 16, 23, 42]
print(len(nums), sum(nums), min(nums), max(nums), sorted(nums, reverse=True))
print(abs(-7), pow(2, 5), divmod(17, 5), round(sum(nums) / len(nums), 2))
print(type(nums).__name__, id(nums) > 0, dir(3)[:3])          # introspection helpers
help(len) if False else None                                  # help(len) prints documentation
```

OUTPUT

```
6 108 4 42 [42, 23, 16, 15, 8, 4]
7 32 (3, 2) 18.0
list True ['__abs__', '__add__', '__and__']
```

Mini project: a bill calculator

Python

```
items = {"Notebook": (45.0, 10), "Pen pack": (60.0, 25), "Stapler": (180.0, 4)}    # name: (price, quantity)
GST = 0.18
subtotal = 0
print(f"{'Item':<12}{'Price':>8}{'Qty':>5}{'Amount':>10}")
for name, (price, qty) in items.items():
    amount = price * qty; subtotal += amount
    print(f"{name:<12}{price:>8.2f}{qty:>5}{amount:>10,.2f}")
print("-" * 35)
print(f"{'Subtotal':<25}{subtotal:>10,.2f}")
print(f"{'GST (18%)':<25}{subtotal * GST:>10,.2f}")
print(f"{'Total':<25}{subtotal * (1 + GST):>10,.2f}")
```

OUTPUT

Item	Price	Qty	Amount
Notebook	45.00	10	450.00
Pen pack	60.00	25	1,500.00
Stapler	180.00	4	720.00

Subtotal			2,670.00
GST (18%)			480.60
Total			3,150.60

(The `for` loop and dictionary are explained in Lessons 6 and 9; notice how f-string alignment makes a tidy table.)

Summary

KEY TAKEAWAYS

- A **variable** is a name pointing to an object; names are `snake_case`, case-sensitive, and must not be keywords or shadow built-ins.
- Core types: `int`, `float`, `str`, `bool`, `NoneType`; `list/tuple/dict/set` come later. Check with `type()` and `isinstance()`.
- Strings, numbers and tuples are **immutable**; lists, dicts and sets are **mutable**.
- Floats are approximate (`0.1 + 0.2`): use `Decimal` for money and `math.isclose` for comparisons. `round()` uses banker's rounding.
- Convert with `int()`, `float()`, `str()`, `bool()`; `int("3.7")` fails, `int(3.99)` truncates, `bool("False")` is `True`.
- Format with **f-strings** (`{x:,.2f}`, `{p:.1%}`, `{n:05d}`, `{s:<10}`); `print` has `sep` and `end`.
- `input()` returns a string: convert and validate it.

Quick quiz

6 questions · tap an answer to check it

1. What does `type(10 / 2)` return?

A `int`

B `float`

C `str`

D `bool`

Answer: B. `/` always returns a float in Python 3 (5.0).

2. What is `int(-3.9)` ?

A `-4`

B `-3`

C `3`

D `-3.9`

Answer: B. `int()` truncates towards zero.

3. Which is NOT falsy?

A `0`

B `""`

C `"0"`

D `None`

Answer: C. A non-empty string, including `"0"`, is truthy.

4. What does `print(f"{0.2345:.1%}")` print?

A 0.2

B 23.4%

C 2.3%

D 0.23

Answer: B. `.1%` multiplies by 100 and shows one decimal. 0.2345 displays as 23.4% because 0.2345 is stored as a binary value just below 0.2345.

5. `x = input("Age: ")` and the user types 25. What is `type(x)` ?

A `int`

B `str`

C `float`

D `bool`

Answer: B. `input()` always returns text.

6. Why is `0.1 + 0.2 == 0.3` False?

A Python is buggy

B Decimal fractions cannot be represented exactly in binary floating point

C Because of integer division

D Because of rounding to even

Answer: B. Use `math.isclose` or `Decimal`.

EXERCISE Practice

(1) Ask for a price and quantity (simulate the input), compute the total with 18% GST and print it as ₹1,23,456.78 -style with thousands separators and 2 decimals. (2) Convert "1010" from binary to int and 255 to hex. (3) Explain the output of `print(round(2.5), round(3.5))` .

Show solution (worked answer)

Python

```
price, qty = float("1250.5"), int("3")
total = price * qty * 1.18
print(f"Total: ₹{total:,.2f}")           # ₹4,426.77
print(int("1010", 2), hex(255))        # 10 0xff
print(round(2.5), round(3.5))          # 2 4 : halves round to the nearest even number
```

INTERVIEW CORNER Questions you may be asked

Q: What is the difference between mutable and immutable types? Give examples. Immutable objects cannot change after creation (int, float, str, tuple); mutable ones can (list, dict, set). It matters for function arguments, dictionary keys (must be immutable/hashable) and copying.

Q: What is the output of `bool("False")` and why? True: any non-empty string is truthy, regardless of its content.

Q: How do you compare floats safely? `math.isclose(a, b)` or `abs(a - b) < tol` ; use `Decimal` for exact decimal arithmetic such as money.

Q: `is` vs `=` ? `=` compares values; `is` compares identity (same object). Use `is None` for None.

Operators, Strings, Indexing, Slicing & String Methods

Master Python's operators (arithmetic, comparison, logical, assignment, membership, identity, bitwise) and work with text: string indexing and slicing, immutability, every string method an analyst needs (split, join, strip, replace, find, startswith, case methods, isdigit...), escape sequences, and cleaning real messy text.

● Python classes 3-4

✦ Python 3.13

Beginner

🕒 120 min

By the end of this lesson you can

- Use arithmetic, comparison, logical, membership and identity operators and explain operator precedence
- Index and slice strings forwards and backwards, including steps, and explain immutability
- Apply string methods to clean, split, search, join and format text
- Build small text-cleaning functions and check results on real data
- Recognise common string pitfalls (encoding, whitespace, case, escape characters)

Operators

Arithmetic

Operator	Meaning	Example	Result
<code>+</code> <code>-</code> <code>*</code>	Add, subtract, multiply	<code>7 * 3</code>	21
<code>/</code>	True division (always float)	<code>7 / 2</code>	3.5
<code>//</code>	Floor division	<code>7 // 2</code> , <code>-7 // 2</code>	3, -4
<code>%</code>	Remainder (modulo)	<code>7 % 2</code> , <code>-7 % 3</code>	1, 2
<code>**</code>	Power	<code>2 ** 10</code>	1024

Python

```
print(7 / 2, 7 // 2, -7 // 2, 7 % 2, -7 % 3, 2 ** 10, 9 ** 0.5)
print(divmod(17, 5))           # (quotient, remainder) in one call
hours = 135; print(f"{hours // 60} h {hours % 60} min")      # split minutes into hours and minutes
print(10 % 3 == 1, 12 % 2 == 0)  # test odd/even and divisibility
```

OUTPUT

```
3.5 3 -4 1 2 1024 3.0
(3, 2)
2 h 15 min
True True
```

Comparison and logical

Python

```
x = 15
print(x > 10, x == 15, x != 15, 10 < x <= 20)      # chained comparison: 10 < x and x <= 20
print(x > 10 and x < 20, x < 5 or x > 12, not x > 10)
print("apple" < "banana", "Zebra" < "apple")      # strings compare by character codes: uppercase
                                                    # before lowercase
```

OUTPUT

```
True True False True
True True False
True True
```

Short-circuit evaluation: `A and B` stops at A if A is falsy; `A or B` stops at A if A is truthy. They return the **deciding operand**, not just True/False:

Python

```
print(0 or "default", "" or "n/a", "x" and "y", None and 1)
name = ""; print(name or "Anonymous")      # a common idiom for default values
print(5 > 3 and print("evaluated") is None) # right side runs only when needed
```

OUTPUT

```
default n/a y None
Anonymous
evaluated
True
```

Assignment, membership, identity

Python

```
count = 10; count += 5; count *= 2; count -= 1; count /= 4; count **= 2
print(count)
print(3 in [1, 2, 3], "py" in "python", "z" not in "python", "a" in {"a": 1}) # membership
a = [1, 2]; b = [1, 2]; c = a
print(a == b, a is b, a is c, None is None) # equality vs
identity
```

OUTPUT

```
49
True True True True
True False True True
```

Bitwise (awareness)

& AND, | OR, ^ XOR, ~ NOT, << >> shifts: used for flags and low-level work; in pandas/NumPy & and | combine boolean arrays (use brackets).

Python

```
print(5 & 3, 5 | 3, 5 ^ 3, 5 << 1, 5 >> 1, bin(5))
```

OUTPUT

```
1 7 6 10 2 0b101
```

Precedence

From highest to lowest: ** → unary -x → * / // % → + - → comparisons (< = in is) → not → and → or . When in doubt, use brackets.

Python

```
print(2 + 3 * 4 ** 2, (2 + 3) * 4 ** 2, -2 ** 2, (-2) ** 2, not 1 == 2 or False)
```

OUTPUT

```
50 80 -4 4 True
```

Walrus operator (3.8+)

:= assigns inside an expression: if (n := len(items)) > 10: print(f"{n} items") . Useful in loops and comprehensions; do not overuse.

Strings

A **string** is an immutable sequence of Unicode characters. Create with single, double or triple quotes.

Python

```
city = "Bengaluru"
quote = 'She said "hello"'
multi = """Line 1
Line 2"""
hindi = "नमस्ते" # Unicode works everywhere
print(len(city), len(hindi), multi.splitlines(), city.upper())
print("Ha" * 3, "data" + " " + "science", "-" * 20)
```

OUTPUT

```
9 6 ['Line 1', 'Line 2'] BENGALURU
HaHaHa data science -----
```

Indexing

Characters are numbered from 0 on the left and from -1 on the right.

index	0	1	2	3	4	5	6	7	8
	B	E	N	G	A	L	U	R	U
neg.	-9	-8	-7	-6	-5	-4	-3	-2	-1

Indexing a string: positive indexes count from the left (starting at 0), negative from the right (starting at -1).

Python

```
w = "BENGALURU"
print(w[0], w[3], w[-1], w[-3])
try: print(w[20])
except IndexError as e: print("IndexError:", e)
```

OUTPUT

```
B G U U
IndexError: string index out of range
```

Slicing: `s[start:stop:step]`

`start` is included, `stop` is **excluded**; omit a part to mean "from the beginning / to the end"; a negative step goes backwards.

Python

```
w = "BENGALURU"
print(w[0:3], w[:3], w[3:], w[3:6], w[-4:], w[:-1])
print(w[::2], w[::-1], w[1:3], w[8:2:-2])
print(w[2:100], repr(w[5:2]))           # out-of-range slices are safe; an empty slice is ''
```

OUTPUT

```
BEN BEN GALURU GAL LURU BENGALUR
BNAUU URULAGNEB EAR UUA
NGALURU ''
```

Slice	Meaning
<code>s[:n]</code>	First n characters
<code>s[-n:]</code>	Last n characters
<code>s[a:b]</code>	From a up to (not including) b; length b – a
<code>s[::-1]</code>	Reversed
<code>s[::2]</code>	Every second character

Python

```
order_id = "IN-2025-00042"
country, year, number = order_id[:2], int(order_id[3:7]), int(order_id[-5:])
print(country, year, number)           # same as LEFT/MID/RIGHT in Excel and SQL
```

OUTPUT

```
IN 2025 42
```

String methods

Strings are immutable, so every method **returns a new string**; the original stays unchanged.

Case and cleaning

Python

```
s = "  new DELHI  "
print(repr(s.strip()), repr(s.lstrip()), repr(s.rstrip()))
print(s.strip().lower(), s.strip().upper(), s.strip().title(), s.strip().capitalize(),
s.strip().swapcase())
print("xxHixx".strip("x"), "2025-04-27".replace("-", "/"), "a,b,,c".replace(",", ", "))
print("Bengaluru".casefold() == "BENGALURU".casefold())           # robust case-insensitive comparison
```

OUTPUT

```
'new DELHI' 'new DELHI' ' ' 'new DELHI'
new delhi NEW DELHI New Delhi New delhi NEW delhi
Hi 2025/04/27 a,b,c
True
```


Searching and testing

Python

```
t = "Delhi, Mumbai, Delhi, Pune"
print(t.find("Delhi"), t.rfind("Delhi"), t.find("Goa"), t.count("Delhi"), t.index("Mumbai"))
print(t.startswith("Delhi"), t.endswith("Pune"), "Mumbai" in t)
print("12345".isdigit(), "12.5".isdigit(), "abc".isalpha(), "abc123".isalnum(), " ".isspace(),
      "Abc".istitle(), "abc".islower())
```

OUTPUT

```
0 15 -1 2 7
True True True
True False True True True True True
```

`find` returns `-1` when not found; `index` raises `ValueError`. `isdigit` is False for decimals and negatives, so use `try: float(s)` to test numbers.

Splitting and joining

Python

```
csv_line = "E101,Kavya Iyer,HR,97000"
parts = csv_line.split(",")
print(parts, len(parts))
print("one two three".split(), "a-b-c".split("-", 1), "line1\nline2".splitlines(),
      "path/to/file.csv".rsplit("/", 1))
print(", ".join(["Pune", "Delhi", "Goa"]), "".join(["a", "b"]), " | ".join(str(n) for n in [1, 2, 3]))
print("abc".partition("b"), "IN-2025-00042".partition("-"))
```

OUTPUT

```
['E101', 'Kavya Iyer', 'HR', '97000'] 4
['one', 'two', 'three'] ['a', 'b-c'] ['line1', 'line2'] ['path/to', 'file.csv']
Pune, Delhi, Goa ab 1 | 2 | 3
('a', 'b', 'c') ('IN', '-', '2025-00042')
```

`split()` with no argument splits on any whitespace and ignores repeats: the best way to tokenise text. `join` is called on the **separator**, with a list of strings.

Alignment, padding and formatting

Python

```
print("42".zfill(6), "7".rjust(4, "0"), "ab".ljust(5, ".") + "|", "ab".center(8, "*"))
print("Total".ljust(10) + "1,500".rjust(10))
print("{:>8} {:<8} {:^8}".format("right", "left", "mid"))
print("tab\tseparated", "new\\nline stays literal", "unicode: \u20b9 \u2713")
```

OUTPUT

```
000042 0007 ab... | ***ab***
Total          1,500
    right left      mid
tab      separated new\nline stays literal unicode: ₹ ✓
```

Translating and encoding

Python

```
table = str.maketrans({"₹": "Rs ", ",": ""})
print("₹1,25,000".translate(table))
print("naïve café".encode("utf-8"), "abc".encode("ascii"), b"abc".decode())
print(ord("A"), chr(97), "é".encode("utf-8"), len("é".encode("utf-8")))
```

OUTPUT

```
Rs 125000
b'na\xc3\xafve caf\xc3\xa9' b'abc' abc
65 a b'\xc3\xa9' 2
```

NOTE Encoding

Text on disk is **bytes**. Most files today are **UTF-8**; if you see `Ã©` or `💎`, the file was read with the wrong encoding. In pandas use `pd.read_csv(path, encoding="utf-8")` or `"latin-1" / "cp1252"` for old Windows exports.

Method cheat sheet

Goal	Methods
Change case	<code>lower()</code> , <code>upper()</code> , <code>title()</code> , <code>capitalize()</code> , <code>casefold()</code>
Trim	<code>strip()</code> , <code>lstrip()</code> , <code>rstrip()</code> , <code>removeprefix()</code> , <code>removesuffix()</code>
Replace	<code>replace(old, new, count)</code> , <code>translate()</code>
Search	<code>find()</code> , <code>rfind()</code> , <code>index()</code> , <code>count()</code> , <code>in</code> , <code>startswith()</code> , <code>endswith()</code>
Test	<code>isdigit()</code> , <code>isalpha()</code> , <code>isalnum()</code> , <code>isspace()</code> , <code>isupper()</code> , <code>islower()</code>
Split / join	<code>split()</code> , <code>rsplit()</code> , <code>splitlines()</code> , <code>partition()</code> , <code>join()</code>
Pad / align	<code>zfill()</code> , <code>ljust()</code> , <code>rjust()</code> , <code>center()</code>
Format	f-strings, <code>format()</code> , <code>format_map()</code>

Cleaning real text

Customer names from the course's `customers.csv`, cleaned with the same rules you used in Excel (`TRIM`, `PROPER`) and SQL:

Python

```
import pandas as pd
cust = pd.read_csv("customers.csv")
raw = [" asha RAO ", "ravi verma", "MEERA\tIYER", "kabir singh "] # messy examples
def clean_name(name):
    parts = name.split() # split() collapses all whitespace
    return " ".join(p.capitalize() for p in parts)
print([clean_name(n) for n in raw])
print(cust.customer_name.head(5).tolist())
first_names = [n.split()[0] for n in cust.customer_name]
print(len(first_names), "names; most common first names:",
pd.Series(first_names).value_counts().head(5).to_dict())
```

OUTPUT

```
['Asha Rao', 'Ravi Verma', 'Meera Iyer', 'Kabir Singh']
['Saanvi Khan', 'Meera Bose', 'Riya Shah', 'Diya Fernandes', 'Neha Sharma']
800 names; most common first names: {'Arjun': 36, 'Vikram': 31, 'Aarav': 31, 'Sneha': 30,
'Priya': 30}
```

Python

```
def initials(name): return "".join(w[0].upper() for w in name.split())
def mask_email(email):
    user, _, domain = email.partition("@")
    return user[0] + "*" * (len(user) - 2) + user[-1] + "@" + domain
print(initials("kavya iyer"), mask_email("kavya.iyer@abc.in"))
print("palindrome" if (s := "Level").lower() == s.lower()[::-1] else "not")
```

OUTPUT

```
KI k*****r@abc.in
palindrome
```

Counting characters and words

Python

```
text = "data science is fun and data engineering is fun too"
words = text.split()
freq = {}
for w in words: freq[w] = freq.get(w, 0) + 1
print(freq)
print(max(freq, key=freq.get), len(words), len(set(words)))
```

OUTPUT

```
{'data': 2, 'science': 1, 'is': 2, 'fun': 2, 'and': 1, 'engineering': 1, 'too': 1}
data 10 7
```

(`collections.Counter(words)` does this in one line: Lesson 8.)

Common pitfalls

Pitfall	Example	Fix
Forgetting strings are immutable	<code>s.upper()</code> without assigning	<code>s = s.upper()</code>
Whitespace/case mismatches	<code>"Delhi " != "Delhi"</code>	<code>strip()</code> , <code>lower()</code> before comparing
Using <code>+</code> with numbers	<code>"Total: " + 5</code>	f-string or <code>str(5)</code>
<code>isdigit</code> for decimals	<code>"3.5".isdigit()</code> → False	try: <code>float(x)</code>
Off-by-one in slices	<code>s[0:3]</code> has 3 characters	stop is excluded
Backslashes in Windows paths	<code>"C:\new"</code> → newline!	raw string <code>r"C:\new"</code> or forward slashes
Huge strings in loops	<code>s += x</code> in a loop	build a list and <code>"".join(list)</code>
Wrong encoding	Ã© characters	<code>encoding="utf-8"</code>

DO IT WITH AI

Do it with AI

Ask an assistant to produce 10 messy phone numbers (fake) and then write a function to standardise them to `+91-XXXXX-XXXXX` . Test it yourself on edge cases (spaces, dashes, leading 0, +91 already present) and compare approaches: string methods first, then regular expressions (Lesson 14).

Summary

KEY TAKEAWAYS

- Operators: `/` true division, `//` floor division, `%` remainder, `**` power; comparisons chain (`10 < x ≤ 20`); and `/` or short-circuit and return an operand; use `in` , `not in` , `is` .
- Strings are **immutable** sequences: index from 0 or `-1`, slice `[start:stop:step]` (stop excluded).
- Methods return new strings: `strip` , `lower/upper/title` , `replace` , `split` / `join` , `find` / `count` , `startswith` , `zfill` , `partition` .
- Clean text by trimming, normalising case/whitespace and replacing variants before comparing or counting.
- Watch pitfalls: forgetting to assign, off-by-one slices, wrong encodings, Windows backslashes.

Quick quiz

6 questions · tap an answer to check it

1. What does `"BENGALURU"[2:5]` return?

A NGA

B ENG

C NGAL

D GAL

Answer: A. Index 2 is N, 3 is G and 4 is A; the stop index 5 is excluded.

2. What is `-7 // 2` in Python?

A -3

B -4

C 3

D -3.5

Answer: B. Floor division rounds down towards minus infinity.

3. What does `0 or "default"` return?

A 0

B True

C "default"

D None

Answer: C. `or` returns the first truthy operand (or the last operand).

4. Which statement is true about strings?

- A `s[0] = "X"` changes the first character
- B Strings are immutable, so methods like `upper()` return a new string
- C `split()` changes the original string
- D `strip()` removes only spaces inside the text

Answer: B. Assign the result to keep it.

5. How do you split text on any whitespace and ignore repeated spaces?

- A `text.split(" ")`
- B `text.split()`
- C `text.partition()`
- D `text.strip()`

Answer: B. `split()` without an argument splits on runs of whitespace.

6. `"3.5".isdigit()` returns:

- A True
- B False
- C 3.5
- D An error

Answer: B. `isdigit()` is True only for strings of digits; use `float()` inside try/except for decimals.

EXERCISE String practice

(1) From "IN-2025-00042" extract the country, year (int) and sequence number (int) with slicing. (2) Write `is_valid_pin(text)` that returns True for six-digit PIN codes not starting with 0. (3) Reverse the words in "data science is fun" (word order, not letters). (4) Turn " rAHuL VERMA " into "Rahul Verma" .

[Show solution \(worked answer\)](#)

Python

```
oid = "IN-2025-00042"; print(oid[:2], int(oid[3:7]), int(oid[-5:]))
def is_valid_pin(t): return len(t) == 6 and t.isdigit() and t[0] != "0"
print(is_valid_pin("411001"), is_valid_pin("041100"), is_valid_pin("4110a1"))
print(" ".join("data science is fun".split()[::-1]))          # fun is science data
print(" ".join(w.capitalize() for w in " rAHuL VERMA ".split()))
```

INTERVIEW CORNER Questions you may be asked

Q: How do you reverse a string in Python? `s[::-1]` (slice with step -1) or `"".join(reversed(s))` .

Q: Why use `"".join(list)` instead of `+=` in a loop? Strings are immutable; each `+=` creates a new string (quadratic time). Building a list and joining once is linear.

Q: Difference between `find` and `index` ? `find` returns -1 if the substring is missing; `index` raises `ValueError` .

Q: What does `//` do with negative numbers? Floors toward minus infinity: `-7 // 2 = -4` , and `-7 % 3 = 2` , so `(a // b) * b + a % b = a` always holds.

Control Flow: if, elif, else, Nested Conditions & match

Make programs decide. Learn indentation, if/elif/else, comparison and logical conditions, truthiness, conditional expressions (ternary), nested decisions, the match statement, and how Python's if maps to Excel's IF/IFS and SQL's CASE WHEN, with data-flavoured examples.

● Python class 3

✦ Python 3.13

Beginner

🕒 90 min

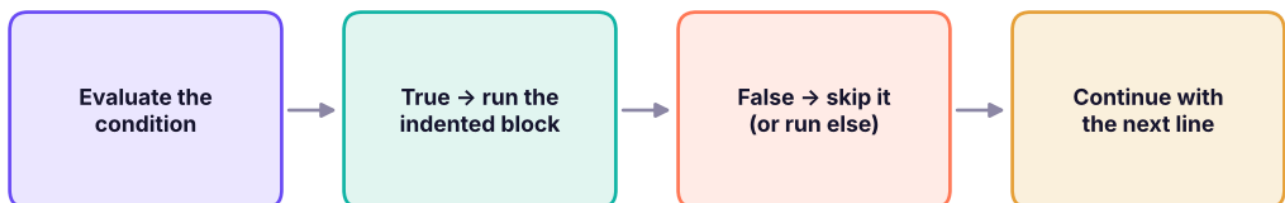
By the end of this lesson you can

- Write correct if / elif / else blocks and explain why indentation matters
- Combine conditions with and, or, not, in, chained comparisons and truthiness
- Use conditional expressions and the match statement
- Order conditions correctly and avoid common branching bugs
- Translate a spreadsheet or SQL rule into Python

Decisions and indentation

A program normally runs line by line. **Control flow** lets it choose between paths. In Python, a **colon** `:` starts a block and **indentation** (4 spaces by convention) says which lines belong to it.

if / else flow



An if statement tests a condition. If it is True the indented block runs; otherwise Python moves on (or runs else).

Python

```
marks = 82
if marks ≥ 40:
    print("Pass")
    print("Well done!")
print("This line always runs")
```


OUTPUT

Pass
Well done!
This line always runs

Python

```
if True:  
    print("not indented")           # IndentationError: expected an indented block after 'if' statement
```

COMMON MISTAKES Indentation rules

Use **4 spaces** (never mix tabs and spaces). Everything in the same block has the **same indentation**. A missing colon or wrong indentation causes `SyntaxError` / `IndentationError`.

if / elif / else

Python

```
marks = 82  
if marks ≥ 90:  
    grade = "A+"  
elif marks ≥ 75:  
    grade = "A"  
elif marks ≥ 60:  
    grade = "B"  
elif marks ≥ 40:  
    grade = "C"  
else:  
    grade = "Fail"  
print(marks, "→", grade)
```

OUTPUT

82 → A

Python checks the conditions **top to bottom** and runs the **first** one that is true, then skips the rest. This is Excel's nested `IF` / `IFS` and SQL's `CASE WHEN` (Excel Lesson 5, SQL Lesson 7).

COMMON MISTAKES Order matters

If you test `marks ≥ 40` first, an 82 gets "C" because the first true branch wins. Put the most specific / strictest condition first.

Python

```
for m in (95, 82, 61, 45, 12):
    g = "A+" if m ≥ 90 else "A" if m ≥ 75 else "B" if m ≥ 60 else "C" if m ≥ 40 else "Fail"    #
    chained conditional expression
    print(m, g)
```

OUTPUT

```
95 A+
82 A
61 B
45 C
12 Fail
```

Conditional expression (ternary)

`value_if_true if condition else value_if_false` is Python's one-line `IF()` :

Python

```
salary = 97000
band = "High" if salary ≥ 100000 else "Standard"
bonus = salary * 0.1 if band == "High" else 0
print(band, bonus)
```

OUTPUT

```
Standard 0
```

Conditions

Comparison and logical operators

Python

```
age, city, income = 29, "Pune", 85000
print(age ≥ 18 and city == "Pune")           # both must be true
print(city in ("Pune", "Mumbai") or income > 100000) # at least one
print(not (age < 18))
print(18 ≤ age < 60)                           # chained comparison
if 18 ≤ age < 60 and income ≥ 50000 and city in ("Pune", "Mumbai", "Delhi"):
    print("Eligible for the metro premium plan")
```

OUTPUT

```
True
True
True
```

```
True
Eligible for the metro premium plan
```

Truthiness

Python treats "empty" and zero-like values as False (Lesson 2), so you can test them directly:

Python

```
name = ""
items = [3, 5]
note = None
if not name: print("name is empty")
if items: print("items has", len(items), "elements")
if note is None: print("note is missing")           # use `is None`, not `== None`
if 0 < len(items) <= 5: print("small list")
```

OUTPUT

```
name is empty
items has 2 elements
note is missing
small list
```

Membership and string tests

Python

```
status = "Shipped"
if status in ("Shipped", "Delivered"): print("order has left the warehouse")
email = "kavya.iyer@abc.in"
if "@" in email and email.endswith(".in"): print("Indian email address")
```

OUTPUT

```
order has left the warehouse
Indian email address
```

Nested conditions

Python

```
is_member, cart_value = True, 3200
if cart_value > 0:
    if is_member:
        discount = 0.1 if cart_value >= 3000 else 0.05
    else:
        discount = 0.02 if cart_value >= 3000 else 0
    print(f"Discount: {discount:.0%}")
else:
    print("Cart is empty")
```

OUTPUT

Discount: 10%

Prefer **flat** conditions with `and` to deeply nested ones when they read clearly:

Python

```
def shipping_fee(value, is_member):
    if value ≤ 0: return 0
    if is_member and value ≥ 500: return 0
    if value ≥ 1500: return 0
    return 49
print([shipping_fee(v, m) for v, m in [(0, True), (600, True), (600, False), (1600, False)]])
```

OUTPUT

[0, 0, 49, 0]

Returning early (**guard clauses**) keeps code flat and readable.

match: structural pattern matching (3.10+)

`match` compares one value with several patterns: a clean alternative to a long chain of `=` tests.

Python

```
def describe_status(code):
    match code:
        case 200 | 201:
            return "success"
        case 301 | 302:
            return "redirect"
        case 404:
            return "not found"
        case int(c) if 500 ≤ c < 600:
            return "server error"
        case _:
            return "unknown"
print([describe_status(c) for c in (200, 404, 503, 999)])
```

OUTPUT

['success', 'not found', 'server error', 'unknown']

Python

```
def handle(command):
    match command.split():
        case ["load", filename]: return f"loading {filename}"
        case ["save", filename]: return f"saving {filename}"
        case ["quit"]:           return "bye"
        case _:                  return "unknown command"
print(handle("load sales.csv"), "|", handle("quit"), "|", handle("fly away"))
```

OUTPUT

loading sales.csv | bye | unknown command

Data examples

Rule-based bonus (the same as an Excel IF + AND)

Python

```
import pandas as pd
emp = pd.read_csv("employees_small.csv")
def bonus(salary, rating, dept):
    if rating >= 5: rate = 0.15
    elif rating >= 4: rate = 0.10
    elif rating >= 3: rate = 0.05
    else: rate = 0
    if dept == "Sales": rate += 0.02          # extra for Sales
    return round(salary * rate)
emp["bonus"] = [bonus(s, r, d) for s, r, d in zip(emp.salary, emp.rating, emp.department)]
emp[["name", "department", "salary", "rating", "bonus"]].head(8)
```

OUTPUT

	name	department	salary	rating	bonus
0	Kavya Iyer	HR	97000	4	9700
1	Imran Reddy	Finance	157000	3	7850
2	Harpreet Kulkarni	Finance	131000	4	13100
3	Diya Iyer	HR	90000	4	9000
4	Simran Iyer	HR	139000	4	13900
5	Saanvi Rao	HR	83000	4	8300
6	Isha Mehta	Sales	57000	3	3990
7	Aarav Kulkarni	HR	101000	2	0

8 rows × 5 columns

(In pandas you would use `np.where`, `np.select` or `.apply`: Lesson 19. Writing the rule once as a function makes it testable.)

A small decision tree: loan pre-check

Python

```
def loan_precheck(age, income, existing_emi, credit_score):
    reasons = []
    if age < 21 or age > 60: reasons.append("age outside 21-60")
    if income < 25000: reasons.append("income below ₹25,000")
    if existing_emi > 0.5 * income: reasons.append("EMIs exceed 50% of income")
    if credit_score < 650: reasons.append("credit score below 650")
    return ("Eligible", []) if not reasons else ("Review", reasons)
print(loan_precheck(34, 70000, 20000, 730))
print(loan_precheck(19, 22000, 15000, 610))
```

OUTPUT

```
('Eligible', [])
('Review', ['age outside 21-60', 'income below ₹25,000', 'EMIs exceed 50% of income', 'credit score below 650'])
```

Collecting **all** failing reasons is more useful than stopping at the first, and shows how a series of independent `if` statements (not `elif`) is sometimes right.

if/elif vs several ifs

Pattern	Use when
<code>if ... elif ... else</code>	Choices are mutually exclusive : only one branch should run
Several separate <code>if</code>	Each condition is independent : more than one may apply (collecting reasons, applying several rules)

Common mistakes

Mistake	Fix
<code>if x = 5:</code> (assignment)	<code>if x == 5:</code>
<code>if x == 3 or 4:</code> (always true: 4 is truthy)	<code>if x in (3, 4):</code>
<code>if x == None:</code>	<code>if x is None:</code>
Comparing floats with <code>==</code>	<code>math.isclose</code>
Forgetting <code>elif</code> ordering	strictest first
<code>if len(items) > 0:</code> (verbose)	<code>if items:</code>
Deep nesting	guard clauses / and / helper functions

DO IT WITH AI Do it with AI

Take an Excel rule from your own work, such as a nested IF for commission, and ask an assistant to convert it to Python with test cases. Then write five edge-case tests yourself (boundaries like exactly 50,000, blanks, negative values) and see if the code survives.

Summary

KEY TAKEAWAYS

- A colon starts a block; **indentation** (4 spaces) defines it. `if` / `elif` / `else` runs the first true branch.
- Combine conditions with `and`, `or`, `not`, `in`, chained comparisons; `empty/zero/None` are falsy; test `None` with `is None`.
- Use the **ternary** `a if cond else b` for simple choices and the `match` statement for many exact cases and patterns.
- Order conditions from strictest to loosest; prefer flat code with guard clauses; use separate `ifs` when several rules can apply.
- Wrap rules in functions so they can be tested and reused.

Quick quiz

6 questions · tap an answer to check it

1. What does `x if cond else y` do?

- ☐ A Runs a loop
- ☒ B Evaluates to `x` when `cond` is `True`, otherwise `y`
- ☐ C Defines a function
- ☐ D Compares `x` and `y`

Answer: B. It is Python's conditional (ternary) expression.

2. Which condition is correct for "x is 3 or 4"?

A $x = 3$ or 4

B $x \text{ in } (3, 4)$

C $x = 3$ or 4

D $x \text{ is } (3, 4)$

Answer: B. $x = 3$ or 4 is always truthy because 4 is truthy.

3. With marks = 82, which grade does `if marks ≥ 40: "C" elif marks ≥ 75: "A"` give?

A C

B A

C Fail

D Nothing

Answer: A. The first true branch wins, so the order is wrong.

4. Which is the best way to test for None?

A $x = \text{None}$

B $x \text{ is None}$

C `not x`

D $x = \text{None}$

Answer: B. `is None` tests identity and avoids surprises with falsy values like 0.

5. What is a guard clause?

- A A loop that never ends
- B An early return that handles a special case so the rest of the function is not nested**
- C A try/except block
- D A decorator

Answer: B. Guard clauses keep code flat and readable.

6. What does match do?

- A Matches regular expressions only
- B Compares a value against patterns and runs the first matching case**
- C Loops over a list
- D Imports modules

Answer: B. Structural pattern matching arrived in Python 3.10.

EXERCISE Rules in code

Write `delivery_charge(order_value, city, express)`: free if order $\geq$ ₹1,500; otherwise ₹49 in metros (Delhi, Mumbai, Bengaluru, Chennai, Kolkata, Hyderabad) and ₹79 elsewhere; express adds ₹100 unless the value is $\geq$ ₹5,000. Test five cases including boundaries.

Show solution (worked answer)

Python

```
METROS = {"Delhi", "Mumbai", "Bengaluru", "Chennai", "Kolkata", "Hyderabad"}
def delivery_charge(value, city, express=False):
    fee = 0 if value  $\geq$  1500 else (49 if city in METROS else 79)
    if express and value < 5000: fee += 100
    return fee
tests = [(1500, "Pune", False), (1499, "Delhi", False), (1499, "Pune", True), (5000, "Pune", True), (100, "Mumbai", True)]
print([delivery_charge(*t) for t in tests])      # [0, 49, 179, 0, 149]
```

INTERVIEW CORNER**Questions you may be asked**

Q: What is the difference between `elif` and multiple `if` statements? `elif` branches are mutually exclusive (only the first true runs); separate `ifs` are all evaluated, so several can run.

Q: How do you write a switch-case in Python? Use `match/case` (3.10+), or a dictionary mapping values to functions or results.

Q: What are falsy values in Python? `False`, `None`, `0`, `0.0`, `""`, `[]`, `{}`, `()`, `set()`, and objects whose `__bool__` returns `False` or `__len__` returns `0`.

Q: How do you replace nested `if/else` cleanly? Guard clauses, combined conditions, lookup tables (dicts), or polymorphism.

Loops: while, for, range, break, continue & Patterns

Repeat work without repeating code. Learn while and for loops, range(), enumerate(), zip(), break, continue, the loop else clause, nested loops, accumulation, search and counting patterns, and common loop bugs; with an eye on when to replace loops with vectorised pandas/NumPy.

● Python classes 3–4

✦ Python 3.13

Beginner

🕒 110 min

By the end of this lesson you can

- Write while and for loops and choose between them
- Use range(), enumerate() and zip() for index, value and parallel iteration
- Control loops with break, continue and the loop-else clause; avoid infinite loops
- Apply accumulator, counter, search, nested-loop and pattern-printing recipes
- Recognise when a loop should be replaced by a built-in, comprehension or vectorised operation

Why loops?

A **loop** repeats a block of code. Processing 10,000 rows, trying each file in a folder, retrying a failed request, or checking each number in a list are all loops.

for vs while

for loop

- Iterates over a sequence (list, string, range, dict, file)
- Number of repeats is known from the data
- for city in cities:
- Safe: ends when the items run out

while loop

- Repeats while a condition is True
- Use when the number of repeats is unknown
- while balance > 0:
- Risk: infinite loop if the condition never becomes False

Two kinds of loop. for repeats over a known sequence; while repeats as long as a condition stays true.

The while loop

Python

```
count = 1
while count ≤ 5:
    print("Count:", count)
    count += 1          # without this line the loop would never end!
print("Done")
```

OUTPUT

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Done
```

Every while loop needs: **(1)** a starting value, **(2)** a condition, **(3)** a change inside the loop that eventually makes the condition False.

Python

```
# Compound growth: how many years until 1,00,000 doubles at 8%?
amount, years = 100000, 0
while amount < 200000:
    amount *= 1.08
    years += 1
print(f"{years} years → {amount:,.0f}")          # the 'rule of 72': 72 / 8 = 9 years
```

OUTPUT

```
10 years → 215,892
```

while True with break

Use an "infinite" loop that you leave deliberately when the input is right:

Python

```
attempts = ["abc", "-4", "12"]          # simulated user answers
i = 0
while True:
    answer = attempts[i]; i += 1
    if answer.isdigit() and int(answer) > 0:
        print("Accepted:", answer); break
    print("Invalid:", answer)
```

OUTPUT

```
Invalid: abc
Invalid: -4
```

The for loop

`for item in sequence:` takes each item in turn.

Python

```
for city in ["Pune", "Delhi", "Kochi"]:
    print(city.upper())
for ch in "SQL":
    print(ch, end=" ")
print()
```

OUTPUT

```
PUNE
DELHI
KOCHI
S Q L
```

range()

`range(stop)`, `range(start, stop)`, `range(start, stop, step)` produce integers (stop excluded). It is lazy: `range(1_000_000)` does not store a million numbers.

Python

```
print(list(range(5)), list(range(2, 8)), list(range(10, 0, -3)), list(range(0, 1)))
total = 0
for n in range(1, 101): total += n
print("Sum 1..100 =", total, "| check:", sum(range(1, 101)), "| formula:", 100 * 101 // 2)
```

OUTPUT

```
[0, 1, 2, 3, 4] [2, 3, 4, 5, 6, 7] [10, 7, 4, 1] [0]
Sum 1..100 = 5050 | check: 5050 | formula: 5050
```

enumerate() and zip()

Python

```
products = ["Pen", "Notebook", "Stapler"]
prices = [10, 45, 180]
for i, p in enumerate(products, start=1):          # index and value together
    print(i, p)
for p, price in zip(products, prices):             # walk two lists in parallel
    print(f"{p:<10}-{price:>5}")
print(dict(zip(products, prices)))                  # build a dictionary from two lists
```

OUTPUT

```
1 Pen
2 Notebook
3 Stapler
Pen          10
Notebook     45
Stapler      180
{'Pen': 10, 'Notebook': 45, 'Stapler': 180}
```

Avoid `for i in range(len(x))` when you only need the items; use `for item in x` or `enumerate` .

break, continue, else

Python

```
for n in range(1, 20):
    if n % 3 != 0: continue      # skip numbers not divisible by 3
    if n > 12: break           # leave the loop completely
    print(n, end=" ")
print()
for n in range(2, 6):
    if n == 9: break
else:                          # runs only if the loop finished WITHOUT break
    print("loop completed with no break")
```

OUTPUT

```
3 6 9 12
loop completed with no break
```

Keyword	Effect
<code>break</code>	Exit the nearest loop immediately
<code>continue</code>	Skip to the next iteration
<code>else</code> on a loop	Runs if the loop ended normally (no <code>break</code>): handy for "search and report not found"
<code>pass</code>	Do nothing (placeholder)



Loop patterns

Accumulate (sum / product / build a list)

Python

```
orders = [1200, 450, 3000, 780, 2200]
total, large = 0, []
for amount in orders:
    total += amount
    if amount >= 1000: large.append(amount)
print(total, large, total / len(orders))
```

OUTPUT

7630 [1200, 3000, 2200] 1526.0

Count and find the maximum

Python

```
sales = [("North", 120), ("South", 95), ("East", 140), ("West", 88)]
best_region, best_value = None, float("-inf")
count_above_100 = 0
for region, value in sales:
    if value > best_value: best_region, best_value = region, value
    if value > 100: count_above_100 += 1
print(best_region, best_value, count_above_100)
```

OUTPUT

East 140 2

(`max(sales, key=lambda x: x[1])` does the same in one line, Lesson 10.)

Search with early exit

Python

```
def find_first(items, target):
    for i, x in enumerate(items):
        if x == target: return i          # stop at the first match
    return -1
print(find_first(["a", "b", "c"], "c"), find_first(["a", "b"], "z"))
```

OUTPUT

2 -1

Nested loops

Python

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i * j, end="\t")
    print()
```

OUTPUT

1	2	3
2	4	6
3	6	9

Python

```
# Pattern: right triangle of stars
for row in range(1, 6): print("*" * row)
```

OUTPUT

```
*
**
***
****
*****
```

Nested loops multiply the work: $1,000 \times 1,000 = 1,000,000$ iterations. Be careful with big data.

Reading until a sentinel

Python

```
readings = iter([12, 15, 11, -1, 99])          # -1 marks the end
total = 0
for r in readings:
    if r == -1: break
    total += r
print("Total before sentinel:", total)
```

OUTPUT

Total before sentinel: 38

Data examples

Looping over rows (and why it is slow)

Python

```
import pandas as pd, time
s = pd.read_csv("superstore_sales.csv")
t0 = time.perf_counter()
loss_lines = 0
for profit in s["profit"]:
    if profit < 0: loss_lines += 1
t1 = time.perf_counter()
vec = (s["profit"] < 0).sum()
t2 = time.perf_counter()
print(f"loop: {loss_lines} loss-making lines in {1000 * (t1 - t0):.2f} ms | vectorised: {vec} in {1000 * (t2 - t1):.2f} ms")
```

OUTPUT

loop: 1682 loss-making lines in 0.84 ms | vectorised: 1682 in 0.34 ms

Same answer; the vectorised version is usually many times faster and shorter. **Rule:** in pandas/NumPy avoid Python loops over rows; use column operations, `groupby`, `np.where`. Use loops for files, APIs, orchestration, and small collections.

Summarising with a loop

Python

```
totals = {}
for region, category, sales in zip(s["region"][:2000], s["category"][:2000], s["sales"][:2000]):
    key = (region, category)
    totals[key] = totals.get(key, 0) + sales
for k in sorted(totals)[:6]: print(k, f"{totals[k]:,.0f}")
```

OUTPUT

```
('East', 'Furniture') 2,148,878
('East', 'Office Supplies') 717,588
('East', 'Technology') 10,862,196
('North', 'Furniture') 2,782,383
('North', 'Office Supplies') 645,662
('North', 'Technology') 13,766,572
```

This is a hand-made `GROUP BY`: you now see what `groupby` does behind the scenes.

Common loop bugs

Bug	Symptom	Fix
Missing update in <code>while</code>	Runs forever (press ■ to interrupt)	Change the condition variable inside the loop
Off-by-one with <code>range</code>	Misses the last item	Remember stop is excluded: <code>range(1, n + 1)</code>
Modifying a list while iterating over it	Skipped items	Iterate over a copy <code>for x in items[:]</code> or build a new list
<code>for i in range(len(a))</code> then using a wrong index	<code>IndexError</code>	Use <code>enumerate</code> / <code>zip</code>
Re-using a loop variable after the loop	Holds the last value	Avoid relying on it
Resetting the accumulator inside the loop	Wrong totals	Initialise before the loop
String <code>+=</code> in a loop	Slow for big text	<code>"".join(parts)</code>

TIP

Debug with print or a debugger

Print the loop variable and key values each iteration (`print(i, value, total)`), or use the debugger in VS Code/Jupyter (`%debug`). Test the loop on a tiny example by hand first.

Summary

KEY TAKEAWAYS

- `while` repeats while a condition holds (make sure something changes!); `for` iterates over any sequence.
- `range(start, stop, step)` generates integers, stop excluded; `enumerate` gives index and value; `zip` iterates in parallel.
- `break` exits, `continue` skips to the next item, loop `else` runs when no `break` happened.
- Patterns: accumulator, counter, max/min tracking, search with early exit, nested loops, sentinel.
- For tables of data prefer **vectorised** pandas/NumPy over row loops; use comprehensions and built-ins for small collections.

Quick quiz

6 questions · tap an answer to check it

1. What does `list(range(2, 10, 3))` give?

A [2, 5, 8, 11]

B [2, 5, 8]

C [3, 6, 9]

D [2, 10]

Answer: B. Start 2, step 3, stop 10 excluded.

2. What does `continue` do inside a loop?

A Ends the loop

B Skips the rest of this iteration and moves to the next

C Restarts the program

D Pauses the loop

Answer: B. `break` ends the loop; `continue` skips one iteration.

3. When does a loop's `else` block run?

A When the loop has an error

B When the loop finishes without hitting `break`

C Before the loop

D Every iteration

Answer: B. It is useful for "not found" logic after a search loop.

4. Why is this loop infinite? `n = 5` then `while n > 0: print(n)`

A `print` is wrong

B `n` is never changed inside the loop

C 5 is too large

D range is missing

Answer: B. The condition never becomes False.

5. What is a better way to count rows with negative profit in pandas?

A A for loop over rows

B `(df["profit"] < 0).sum()`

C A while loop

D Nested loops

Answer: B. Vectorised operations are shorter and faster.

6. Which function walks two lists at the same time?

A `range`

B `zip`

C `sorted`

D `len`

Answer: B. `zip(a, b)` pairs items by position.

EXERCISE Loop problems

(1) Print the multiplication table of 7 up to 7×10 . (2) Given a list of order values, print the first one above 5,000 and its position, or "none". (3) Count vowels in a sentence. (4) Print the first 10 Fibonacci numbers. (5) FizzBuzz for 1..30.

Show solution (worked answer)

Python

```
for i in range(1, 11): print(f"7 x {i} = {7 * i}")
vals = [900, 4200, 6100, 800]
for i, v in enumerate(vals):
    if v > 5000: print(v, "at", i); break
else: print("none")
print(sum(ch in "aeiou" for ch in "data engineering is fun".lower()))
a, b = 0, 1
for _ in range(10): print(a, end=" "); a, b = b, a + b
print()
for n in range(1, 31): print("FizzBuzz" if n % 15 == 0 else "Fizz" if n % 3 == 0 else "Buzz" if n % 5 == 0 else n, end=" ")
```

INTERVIEW CORNER Questions you may be asked

Q: FizzBuzz, reverse a string, print Fibonacci: why do interviewers ask these? They test loops, conditions, modulo, variable swapping and clear thinking, and whether you handle edge cases.

Q: for vs while? `for` iterates over a known sequence; `while` loops until a condition changes, for unknown counts such as retries or reading until a sentinel.

Q: How would you speed up a slow loop over a DataFrame? Vectorise (column operations, `np.where`, `groupby`), use built-ins or `apply` only when needed, or push the work into SQL.

Q: What does `for i in range(len(lst))` signal? Often a less Pythonic style; prefer iterating over items, `enumerate` or `zip` unless you need indexes to modify items.

Lists: Creating, Slicing, Methods, Sorting & Comprehensions

Store and process ordered collections with lists. Index and slice them, add, remove and sort items, understand mutability, aliasing and copying (shallow vs deep), use nested lists, list comprehensions, and common patterns for cleaning and summarising data.

● Python class 5

✦ Python 3.13

Beginner

🕒 120 min

By the end of this lesson you can

- Create lists and use indexing, slicing, concatenation, repetition and membership tests
- Apply list methods (append, extend, insert, remove, pop, sort, reverse, index, count) and built-ins (len, sum, min, max, sorted)
- Explain mutability, aliasing and the difference between assignment, shallow copy and deep copy
- Build lists with comprehensions, including conditions and nested loops
- Use lists for typical data tasks: filtering, ranking, de-duplicating while keeping order, flattening and unzipping

What is a list?

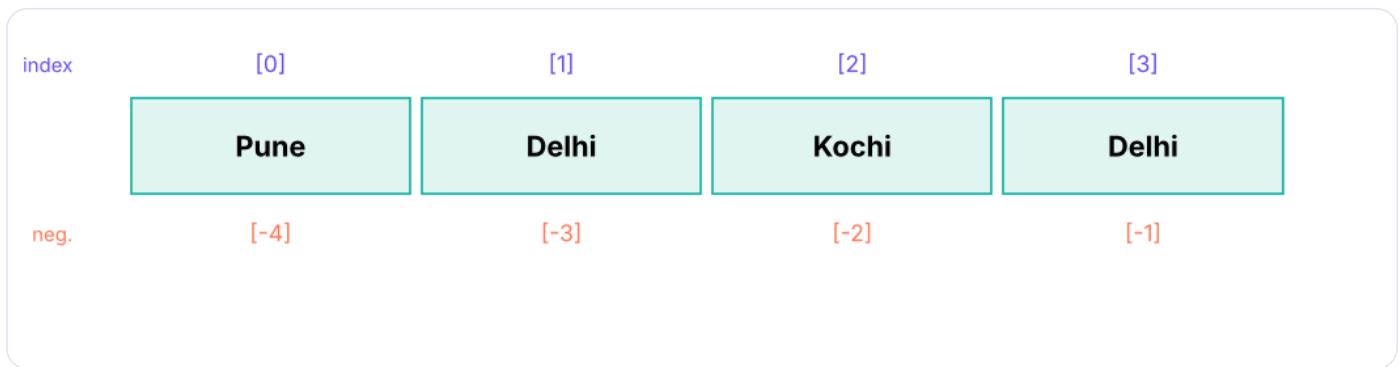
A **list** is an ordered, **mutable** collection of items written in square brackets. Items can be of any type, and may repeat.

Python

```
cities = ["Pune", "Delhi", "Kochi", "Delhi"]
mixed = [101, "Kavya", 97000.5, True, None]
empty = []
print(cities, len(cities), type(cities).__name__)
print(list("data"), list(range(5)), list((1, 2, 3)), [0] * 5)
```

OUTPUT

```
['Pune', 'Delhi', 'Kochi', 'Delhi'] 4 list
['d', 'a', 't', 'a'] [0, 1, 2, 3, 4] [1, 2, 3] [0, 0, 0, 0, 0]
```



A list is a row of numbered boxes. Items are referenced by position; the list itself can grow, shrink and change.

Indexing and slicing

Works exactly like strings (Lesson 3), but lists can also be **changed**.

Python

```
nums = [10, 20, 30, 40, 50, 60]
print(nums[0], nums[-1], nums[1:4], nums[:2], nums[-2:], nums[::2], nums[::-1])
nums[0] = 99 # change one item
nums[1:3] = [21, 31, 32] # replace a slice (the length may change)
print(nums)
print(3 in nums, 99 in nums, nums.count(40), nums.index(40))
```

OUTPUT

```
10 60 [20, 30, 40] [10, 20] [50, 60] [10, 30, 50] [60, 50, 40, 30, 20, 10]
[99, 21, 31, 32, 40, 50, 60]
False True 1 4
```

Adding and removing items

Method	What it does	Example	Returns
<code>append(x)</code>	Add one item at the end	<code>a.append(5)</code>	None
<code>extend(iterable)</code> or <code>+=</code>	Add all items from another sequence	<code>a.extend([6, 7])</code>	None
<code>insert(i, x)</code>	Insert before position i	<code>a.insert(0, "first")</code>	None
<code>remove(x)</code>	Delete the first item equal to x	<code>a.remove(5)</code>	None (error if missing)
<code>pop(i=-1)</code>	Remove and return item at i	<code>last = a.pop()</code>	the item
<code>del a[i]</code> / <code>del a[i:j]</code>	Delete by position or slice	<code>del a[0]</code>	
<code>clear()</code>	Empty the list	<code>a.clear()</code>	None

Python

```
a = [1, 2, 3]
a.append(4); a.extend([5, 6]); a.insert(0, 0); a += [7]
print(a)
print(a.pop(), a.pop(0), a)           # removes 7, then 0
a.remove(3); del a[0]; print(a)
print([1, 2] + [3], [1, 2] * 2)       # concatenation creates new lists
print(a.append(9))                     # None! methods that change a list return None
```

OUTPUT

```
[0, 1, 2, 3, 4, 5, 6, 7]
7 0 [1, 2, 3, 4, 5, 6]
[2, 4, 5, 6]
[1, 2, 3] [1, 2, 1, 2]
None
```

COMMON MISTAKES ``x = x.append(...)``

`append`, `sort`, `reverse` and `extend` change the list **in place** and return `None`. Writing `nums = nums.sort()` leaves `nums` as `None`. Use `sorted(nums)` when you want a new list.

Sorting and reversing

Python

```
scores = [72, 95, 61, 88, 95]
print(sorted(scores), sorted(scores, reverse=True), scores)   # sorted() returns a new list
scores.sort(); print(scores)                                   # .sort() changes the list in
place
words = ["banana", "Apple", "cherry", "apple"]
print(sorted(words), sorted(words, key=str.lower), sorted(words, key=len))
staff = [("Asha", 29, 61000), ("Ravi", 34, 88000), ("Meera", 41, 72000)]
print(sorted(staff, key=lambda p: p[2], reverse=True))         # sort records by the 3rd field
print(list(reversed([1, 2, 3])), min(staff, key=lambda p: p[1])[0])
```

OUTPUT

```
[61, 72, 88, 95, 95] [95, 95, 88, 72, 61] [72, 95, 61, 88, 95]
[61, 72, 88, 95, 95]
['Apple', 'apple', 'banana', 'cherry'] ['Apple', 'apple', 'banana', 'cherry'] ['Apple', 'apple',
'banana', 'cherry']
[('Ravi', 34, 88000), ('Meera', 41, 72000), ('Asha', 29, 61000)]
[3, 2, 1] Asha
```

`sorted()` and `.sort()` are **stable**: equal items keep their original order, so you can sort by several keys by sorting repeatedly from the least to the most important key, or use a tuple key: `key=lambda p: (p[1], -p[2])`.

Built-in helpers

Python

```
data = [4, 8, 15, 16, 23, 42]
print(len(data), sum(data), min(data), max(data), sum(data) / len(data))
print(any(x > 40 for x in data), all(x > 0 for x in data))
print(list(enumerate(["a", "b"], 1)), list(zip([1, 2, 3], "abc")))
a, b, *rest = data; print(a, b, rest) # unpacking with *
first, *middle, last = data; print(first, middle, last)
```

OUTPUT

```
6 108 4 42 18.0
True True
[(1, 'a'), (2, 'b')] [(1, 'a'), (2, 'b'), (3, 'c')]
4 8 [15, 16, 23, 42]
4 [8, 15, 16, 23] 42
```

Mutability, aliasing and copying

A variable holds a **reference** to a list. Assigning a list to another name does **not** copy it.

Python

```
a = [1, 2, 3]
b = a # b is another name for the SAME list
b.append(4)
print(a, b, a is b)
c = a.copy() # a real (shallow) copy; also list(a) or a[:]
c.append(99)
print(a, c, a is c)
```

OUTPUT

```
[1, 2, 3, 4] [1, 2, 3, 4] True
[1, 2, 3, 4] [1, 2, 3, 4, 99] False
```

Alias vs copy

b = a (alias)

- a   [1, 2, 3, 4]   b
- One list, two names
- Changing b changes a
- Use when you really want to share

c = a.copy() (copy)

- a   [1, 2, 3, 4] c   [1, 2, 3, 4, 99]
- Two independent lists
- Changing c leaves a alone
- Shallow: inner lists are still shared

Assignment gives two names for one list; copy() makes a second list. A shallow copy still shares inner lists.

Python

```
import copy
grid = [[1, 2], [3, 4]]
shallow = grid.copy(); deep = copy.deepcopy(grid)
grid[0][0] = 100
print(shallow, deep)           # shallow copy sees the change in the inner list; deep copy does not
```

OUTPUT

```
[[100, 2], [3, 4]] [[1, 2], [3, 4]]
```

WATCH OUT Mutable default arguments and `[[0] \* 3] \* 3`

`[[0] * 3] * 3` creates **three references to the same inner list**; changing one row changes all. Build grids with a comprehension: `[[0] * 3 for _ in range(3)]`. Never use a list as a default function argument (`def f(x, items=[]):` use `None` and create the list inside (Lesson 9).

Nested lists

Python

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[1][2], [row[0] for row in matrix], [sum(row) for row in matrix])
print([list(col) for col in zip(*matrix)])           # transpose
flat = [x for row in matrix for x in row]; print(flat) # flatten
```

OUTPUT

```
6 [1, 4, 7] [6, 15, 24]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions

A comprehension builds a list in one readable line: `[expression for item in iterable if condition]`.

Python

```
squares = [n ** 2 for n in range(1, 8)]
evens = [n for n in range(20) if n % 2 == 0]
upper = [c.upper() for c in ["pune", "delhi"]]
labels = ["High" if s >= 80 else "Low" for s in [95, 60, 82]]
pairs = [(i, j) for i in range(1, 4) for j in range(i, 4)]
print(squares, evens, upper, labels, pairs, sep="\n")
```

OUTPUT

```
[1, 4, 9, 16, 25, 36, 49]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
['PUNE', 'DELHI']
['High', 'Low', 'High']
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

Compare with the loop version:

Python

```
result = []
for n in range(20):
    if n % 2 == 0:
        result.append(n)
print(result == [n for n in range(20) if n % 2 == 0])
```

OUTPUT

```
True
```

Use comprehensions for **simple** transformations and filters; switch to a loop when the logic needs several statements. For very large data use generators (Lesson 10) or pandas.

Data tasks with lists

Python

```
import pandas as pd
emp = pd.read_csv("employees_small.csv")
salaries = emp.salary.tolist(); names = emp.name.tolist(); depts = emp.department.tolist()
top3 = sorted(zip(salaries, names), reverse=True)[:3]
print("Top 3:", [(n, s) for s, n in top3])
fin = [s for s, d in zip(salaries, depts) if d == "Finance"]
print("Finance:", len(fin), "people, average", round(sum(fin) / len(fin)))
above_avg = [n for n, s in zip(names, salaries) if s > sum(salaries) / len(salaries)]
print(len(above_avg), "people earn above the average;", above_avg[:3], "...")
```

OUTPUT

Top 3: [('Vivaan Gupta', 157000), ('Imran Reddy', 157000), ('Krishna Khan', 156000)]
Finance: 7 people, average 95714
9 people earn above the average; ['Imran Reddy', 'Harpreet Kulkarni', 'Simran Iyer'] ...

Python

```
# Remove duplicates but keep the original order (set() would lose the order)
raw = ["Pune", "Delhi", "Pune", "Kochi", "Delhi", "Goa"]
seen, unique = set(), []
for c in raw:
    if c not in seen: seen.add(c); unique.append(c)
print(unique, "| shortcut:", list(dict.fromkeys(raw)))
# Chunk a list into groups of 4
ids = list(range(1, 11)); print([ids[i:i + 4] for i in range(0, len(ids), 4)])
# Moving average of 3
x = [10, 12, 9, 14, 15, 13]; print([round(sum(x[i:i + 3]) / 3, 1) for i in range(len(x) - 2)])
```

OUTPUT

['Pune', 'Delhi', 'Kochi', 'Goa'] | shortcut: ['Pune', 'Delhi', 'Kochi', 'Goa']
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]
[10.3, 11.7, 12.7, 14.0]

Performance notes

Operation	Cost
<code>a[i]</code> , <code>a.append(x)</code> , <code>a.pop()</code>	$O(1)$
<code>x in a</code> , <code>a.index(x)</code> , <code>a.remove(x)</code> , <code>a.insert(0, x)</code>	$O(n)$
<code>a.sort()</code>	$O(n \log n)$

For many membership tests use a **set** ($O(1)$); for a queue use `collections.deque` . (More in the DSA lessons.)

Summary

KEY TAKEAWAYS

- A list is an ordered, **mutable** collection; index and slice like strings; negative indexes count from the end.
- `append` , `extend` , `insert` , `remove` , `pop` , `sort` , `reverse` change the list in place and return `None` ; `sorted()` and `reversed()` give new objects.
- `b = a` is an **alias**; use `a.copy()` (shallow) or `copy.deepcopy(a)` for independent copies; build grids with comprehensions.
- Sort records with `key=` (and tuple keys); sorting is stable.
- Comprehensions `[f(x) for x in xs if cond]` create lists concisely; unpack with `a, *rest = items` .
- Membership tests on lists are $O(n)$: use sets for lookups.

Quick quiz

6 questions · tap an answer to check it

1. What does `[1, 2, 3].append(4)` return?

A `[1, 2, 3, 4]`

B `None` (the list is changed in place)

C `4`

D A new list

Answer: B. Mutating methods return `None`.

2. After `b = a`; `b.append(1)` , what is true?

A `a` is unchanged

B `a` and `b` are the same list, so `a` also contains the new item

C `b` is a copy

D An error occurs

Answer: B. Assignment creates an alias, not a copy.

3. What does `[x * 2 for x in range(5) if x % 2 == 0]` return?

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

B `[0, 4, 8]`

C `[0, 2, 4]`

D `[2, 6]`

Answer: B. Only `x = 0, 2, 4` pass the filter and are doubled.

4. How do you sort a list of tuples by the second element, descending?

A `sorted(t)`

B `sorted(t, key=lambda p: p[1], reverse=True)`

C `t.sort(1)`

D `sort(t, 1)`

Answer: B. The key function selects the field to compare.

5. Why does `grid = [[0] * 3] * 3; grid[0][0] = 1` change every row?

A A bug in Python

B The outer multiplication repeats references to the same inner list

C Lists are immutable

D Comprehensions are required for all lists

Answer: B. Use `[[0] * 3 for _ in range(3)]`.

6. Which is the fastest way to test if an item is in a large collection many times?

A A list

B A set

C A string

D A tuple

Answer: B. Sets use hashing for $O(1)$ average lookups.

EXERCISE List drills

(1) From `[3, 8, 1, 9, 4, 8]` return the second-largest distinct value. (2) Remove duplicates from `["b", "a", "b", "c", "a"]` keeping order. (3) Flatten `[[1, 2], [3], [4, 5, 6]]`. (4) Given names and salaries lists, print "name: salary" sorted by salary descending. (5) Rotate `[1, 2, 3, 4, 5]` by 2 positions to the right.

[Show solution \(worked answer\)](#)

Python

```
print(sorted(set([3, 8, 1, 9, 4, 8]))[-2])           # 8
print(list(dict.fromkeys(["b", "a", "b", "c", "a"]))) # ['b', 'a', 'c']
print([x for sub in [[1, 2], [3], [4, 5, 6]] for x in sub]) # [1, 2, 3, 4, 5, 6]
names, sal = ["A", "B", "C"], [50, 90, 70]
for n, s in sorted(zip(names, sal), key=lambda p: -p[1]): print(f"{n}: {s}")
r = [1, 2, 3, 4, 5]; print(r[-2:] + r[:-2])         # [4, 5, 1, 2, 3]
```

INTERVIEW CORNER Questions you may be asked

Q: Difference between a list and a tuple? Lists are mutable, tuples immutable (hashable if their items are); tuples are used for fixed records and as dictionary keys, lists for collections that change.

Q: Shallow vs deep copy? A shallow copy duplicates the outer list but shares inner objects; a deep copy recursively copies everything.

Q: What is the time complexity of `x in list` vs `x in set`? $O(n)$ vs $O(1)$ average.

Q: How do you reverse a list in place? `lst.reverse()` or `lst[:] = lst[::-1]`; `lst[::-1]` creates a reversed copy.

Tuples & Sets: Immutable Records, Unpacking and Set Algebra

Use tuples for fixed records and multiple return values, master unpacking and named tuples, and use sets for uniqueness, fast membership tests and set algebra (union, intersection, difference) on real customer data.

● Python class 5

✦ Python 3.13

Beginner

🕒 90 min

By the end of this lesson you can

- Create tuples, explain immutability and hashability, and unpack tuples including starred unpacking
- Use namedtuple and dataclass-style records for readable fixed structures
- Create sets and frozensets and use add, discard, remove, update and set operators
- Apply union, intersection, difference and symmetric difference to compare groups
- Choose between list, tuple, set and dict for a given task

Tuples

A **tuple** is an ordered, **immutable** sequence written with commas (brackets are usually optional).

Python

```
point = (28.61, 77.20)
single = (5,)          # the trailing comma makes a one-item tuple
empty = ()
rec = "E101", "Kavya Iyer", "HR", 97000    # packing: no brackets needed
print(point, type(single).__name__, type((5)).__name__, rec[1], rec[-1], rec[1:3], len(rec))
print(rec.count("HR"), rec.index("HR"), (1, 2) + (3,), (0,) * 3, 5 in (1, 5, 9))
try: rec[0] = "E102"
except TypeError as e: print("Error:", e)
```

OUTPUT

```
(28.61, 77.2) tuple int Kavya Iyer 97000 ('Kavya Iyer', 'HR') 4
1 2 (1, 2, 3) (0, 0, 0) True
Error: 'tuple' object does not support item assignment
```


Unpacking

Python

```
emp_id, name, dept, salary = rec          # one name per item
a, b = 1, 2; a, b = b, a                  # swap
first, *others = (10, 20, 30, 40)         # * collects the rest into a list
*head, last = (10, 20, 30, 40)
x, (y, z) = 1, (2, 3)                     # nested unpacking
_, dept_only, *_ = rec                     # _ means "ignore"
print(emp_id, name, a, b, first, others, head, last, x, y, z, dept_only)
for city, sales in [("Pune", 120), ("Delhi", 95)]: # unpack in loops
    print(city, sales)
```

OUTPUT

```
E101 Kavya Iyer 2 1 10 [20, 30, 40] [10, 20, 30] 40 1 2 3 Kavya Iyer
Pune 120
Delhi 95
```

Why tuples?

Use	Example
Fixed records	(lat, lon) , (year, month) , a row from a database
Multiple return values	def stats(x): return min(x), max(x) → lo, hi = stats(data)
Dictionary keys and set members (hashable)	sales[("North", "Technology")]
Protection from accidental change	Constants like METROS = ("Delhi", "Mumbai")
Slightly less memory and faster than lists	

Python

```
def describe(values):
    return min(values), max(values), sum(values) / len(values)
lo, hi, avg = describe([4, 8, 15, 16, 23, 42])
print(lo, hi, round(avg, 2), describe([1, 2]))
print(hash((1, 2)), "| a tuple containing a list is not hashable:", end=" ")
try: hash((1, [2]))
except TypeError as e: print(e)
```

OUTPUT

```
4 42 18.0 (1, 2, 1.5)
-3550055125485641917 | a tuple containing a list is not hashable: unhashable type: 'list'
```

NOTE Tuples are immutable, but their contents might not be

`t = ([1, 2], 3); t[0].append(9)` works, because the tuple holds a **reference** to a mutable list. The tuple itself still points to the same list.

Named tuples and data classes

When a record has many fields, positions like `rec[3]` are hard to read. Name the fields:

Python

```
from collections import namedtuple
from dataclasses import dataclass
Employee = namedtuple("Employee", ["emp_id", "name", "dept", "salary"])
e = Employee("E101", "Kavya Iyer", "HR", 97000)
print(e, e.name, e[3], e._asdict(), e._replace(salary=100000).salary)

@dataclass(frozen=True)                                # frozen = immutable, like a tuple, but with types and methods
class Product:
    sku: str
    price: float
    def with_gst(self, rate=0.18): return round(self.price * (1 + rate), 2)
p = Product("P101", 4500.0)
print(p, p.with_gst())
```

OUTPUT

```
Employee(emp_id='E101', name='Kavya Iyer', dept='HR', salary=97000) Kavya Iyer 97000 {'emp_id':
'E101', 'name': 'Kavya Iyer', 'dept': 'HR', 'salary': 97000} 100000
Product(sku='P101', price=4500.0) 5310.0
```

Sets

A **set** is an **unordered** collection of **unique**, hashable items, written with braces. Sets support fast membership tests and mathematical set operations.

Python

```
cities = {"Pune", "Delhi", "Kochi", "Delhi"}           # duplicates vanish
print(cities, len(cities), "Pune" in cities)
empty = set()                                           # {} is an empty DICT, not a set!
print(set("mississippi"), set([3, 1, 2, 3, 1]), sorted({5, 2, 9}), type({}).__name__,
type(empty).__name__)
```

OUTPUT

```
{'Delhi', 'Pune', 'Kochi'} 3 True
{'m', 'p', 's', 'i'} {1, 2, 3} [2, 5, 9] dict set
```

Method / operator	Effect
<code>s.add(x)</code>	Add an item
<code>s.update(iterable)</code> or <code>s = other</code>	Add many items
<code>s.discard(x)</code>	Remove x if present (no error)
<code>s.remove(x)</code>	Remove x; <code>KeyError</code> if missing
<code>s.pop()</code>	Remove and return an arbitrary item
<code>s.clear()</code>	Empty the set
<code>x in s</code>	Membership, O(1) on average
<code>len(s)</code> , <code>min(s)</code> , <code>max(s)</code> , <code>sum(s)</code> , <code>sorted(s)</code>	Usual helpers (sorted returns a list)

Python

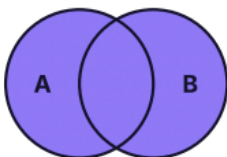
```
s = {1, 2, 3}
s.add(4); s.update([5, 6]); s.discard(10); s.remove(1)
print(s, s.pop() in {2, 3, 4, 5, 6})
```

OUTPUT

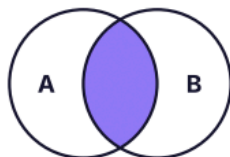
```
{3, 4, 5, 6} True
```

Set algebra

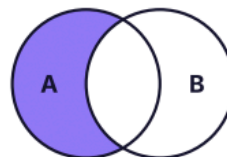
A | B union



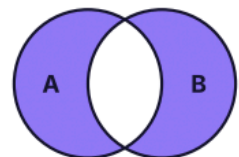
A & B intersection



A - B difference



A ^ B symmetric diff.



The four set operations as Venn diagrams (A and B are sets of customers).

Python

```
a = {"Asha", "Ravi", "Meera", "Kabir"}
b = {"Meera", "Kabir", "Zoya", "Neha"}
print("union      :", sorted(a | b), "  also a.union(b)")
print("intersection:", sorted(a & b), "  also a.intersection(b)")
print("A - B       :", sorted(a - b), "  also a.difference(b)")
print("B - A       :", sorted(b - a))
print("A ^ B       :", sorted(a ^ b), "  (in exactly one)")
print(a ≤ {"Asha", "Ravi", "Meera", "Kabir", "X"}, {"Asha"} < a, a.isdisjoint({"Zoya"}))  # subset,
proper subset, disjoint
```

OUTPUT

```

union      : ['Asha', 'Kabin', 'Meera', 'Neha', 'Ravi', 'Zoya']    also a.union(b)
intersection: ['Kabin', 'Meera']    also a.intersection(b)
A - B      : ['Asha', 'Ravi']    also a.difference(b)
B - A      : ['Neha', 'Zoya']
A ^ B      : ['Asha', 'Neha', 'Ravi', 'Zoya']    (in exactly one)
True True True

```

These are the same operations as SQL's `UNION`, `INTERSECT` and `EXCEPT` (SQL Lesson 9).

Frozen sets

`frozenset` is an immutable set: it can be a dictionary key or a member of another set.

Python

```

fs = frozenset({"North", "South"})
print(fs, {fs: "regions"}, fs | {"East"})

```

OUTPUT

```

frozenset({'North', 'South'}) {frozenset({'North', 'South'}): 'regions'} frozenset({'North', 'South', 'East'})

```

Real data: customer movement with sets

The Superstore data lets us rebuild the retention analysis from SQL Lesson 9 with sets:

Python

```

import pandas as pd
s = pd.read_csv("superstore_sales.csv", parse_dates=["order_date"])
cust = {y: set(s.loc[s.order_date.dt.year == y, "customer_id"]) for y in (2023, 2024, 2025)}
print({y: len(c) for y, c in cust.items()})
retained = cust[2024] & cust[2025]
lost = cust[2024] - cust[2025]
new_in_2025 = cust[2025] - (cust[2023] | cust[2024])
print(f"Retained 2024→2025: {len(retained)} | lost: {len(lost)} | brand new in 2025: {len(new_in_2025)}")
print(f"Retention rate: {len(retained) / len(cust[2024]):.1%} (matches the SQL result: 467 of 590)")
print("Bought in all 3 years:", len(cust[2023] & cust[2024] & cust[2025]), "| bought in exactly one year:", len((cust[2023] ^ cust[2024] ^ cust[2025]) - (cust[2023] & cust[2024] & cust[2025])))

```

OUTPUT

```

{2023: 540, 2024: 590, 2025: 623}
Retained 2024→2025: 467 | lost: 123 | brand new in 2025: 52
Retention rate: 79.2% (matches the SQL result: 467 of 590)
Bought in all 3 years: 311 | bought in exactly one year: 128

```

Same answers as the SQL queries, in three lines. Sets are also the fastest way to **de-duplicate** and to **check membership** in big collections:

Python

```
import time, random
big_list = list(range(200_000)); big_set = set(big_list); probes = [random.randrange(400_000) for _ in range(300)]
t0 = time.perf_counter(); [p in big_list for p in probes]; t1 = time.perf_counter(); [p in big_set for p in probes]; t2 = time.perf_counter()
print(f"300 lookups: list {1000 * (t1 - t0):.1f} ms | set {1000 * (t2 - t1):.3f} ms")
```

OUTPUT

300 lookups: list 329.5 ms | set 0.244 ms

Choosing a collection

Need	Use
Ordered, changeable sequence	<code>list</code>
Ordered, fixed record or dict key	<code>tuple</code>
Unique items, fast "is it in there?", set algebra	<code>set</code>
Key → value lookup	<code>dict</code> (next lesson)
Unique and ordered	<code>dict.fromkeys(items)</code> or a list + set
FIFO queue	<code>collections.deque</code>
Counting	<code>collections.Counter</code>

Summary

KEY TAKEAWAYS

- **Tuples** are ordered and immutable: use them for fixed records, multiple return values and dictionary keys; unpack with `a, b = t` and `first, *rest = t`.
- A one-item tuple needs a comma: `(5,)`. Use **namedtuple** or a frozen **dataclass** for readable records.
- **Sets** hold unique, hashable items with $O(1)$ membership; `{}` is an empty dict, so use `set()`.
- Set algebra: `|` union, `&` intersection, `-` difference, `^` symmetric difference; subset tests with `≤`.
- Sets turn retention/overlap analyses into one-liners and are far faster than lists for membership tests.

Quick quiz

6 questions · tap an answer to check it

1. What is `type((5))` ?

A tuple

B int

C list

D set

Answer: B. Brackets alone do not make a tuple; `(5,)` does.

2. What does `{1, 2, 3} & {2, 3, 4}` return?

A `{1, 2, 3, 4}`

B `{2, 3}`

C `{1, 4}`

D `{1}`

Answer: B. `&` is the intersection.

3. Which can be used as a dictionary key?

A A list

B A tuple of strings

C A set

D A dict

Answer: B. Keys must be hashable; lists, sets and dicts are not.

4. How do you create an empty set?

A `{}`

B `set()`

C `[]`

D `()`

Answer: B. `{}` creates an empty dictionary.

5. What does `first, *rest = (1, 2, 3, 4)` assign to `rest` ?

A `(2, 3, 4)`

B `[2, 3, 4]`

C `4`

D `2`

Answer: B. Starred unpacking collects the remaining items into a list.

6. Why are set lookups faster than list lookups?

A Sets are sorted

B Sets use hashing, giving $O(1)$ average membership tests

C Sets store fewer items

D They are not faster

Answer: B. A list scans item by item ($O(n)$).

EXERCISE Set problems

(1) Find the common cities between {"Pune", "Delhi", "Goa"} and {"Goa", "Kochi", "Pune"} . (2) Customers who bought in 2023 and 2024 but not in 2025 (use the Superstore sets). (3) Write min_max_mean(values) returning a tuple, and unpack it. (4) Check whether two words are anagrams using sorted and also using collections.Counter .

Show solution (worked answer)

Python

```
print({"Pune", "Delhi", "Goa"} & {"Goa", "Kochi", "Pune"})
print(len((cust[2023] & cust[2024]) - cust[2025]))
def min_max_mean(v): return min(v), max(v), sum(v) / len(v)
lo, hi, m = min_max_mean([3, 9, 6])
from collections import Counter
print(sorted("listen") == sorted("silent"), Counter("listen") == Counter("silent"))
```

INTERVIEW CORNER Questions you may be asked

Q: Tuple vs list: when would you choose each? Tuple for fixed, heterogeneous records, hashable keys and returning several values; list for homogeneous collections that change.

Q: How do you remove duplicates from a list while preserving order? list(dict.fromkeys(items)) ; list(set(items)) does not preserve order.

Q: What is the time complexity of set membership and why? Average $O(1)$ because sets are hash tables (worst case $O(n)$ with many collisions).

Q: How do you find items in A but not in B? A - B for sets; for lists use a set for B: [x for x in A if x not in set_b] .

Dictionaries: Key-Value Data, Counting and Grouping

Store and look up data by key. Create and update dictionaries, use get, setdefault, items, Counter and defaultdict, nest dictionaries, and convert between dictionaries, JSON-like records and DataFrames using the Superstore data.

● Python class 6

✦ Python 3.13

Beginner

🕒 100 min

By the end of this lesson you can

- Create dictionaries and read, add, update and delete keys safely
- Loop over keys, values and items and merge dictionaries
- Count and group with get, setdefault, Counter and defaultdict
- Work with nested dictionaries and lists of dictionaries (JSON-like records)
- Convert between dictionaries and pandas DataFrames

What is a dictionary?

A **dictionary** (`dict`) maps **keys** to **values**: look up by name instead of by position. It keeps insertion order (Python 3.7+), and lookups are $O(1)$ on average.

`emp = {'name': 'Kavya Iyer', 'dept': 'HR', 'salary': 97000}`

'name' → 'Kavya Iyer'

key 'name' is hashed to find its value

'dept' → 'HR'

key 'dept' maps to a string

'salary' → 97000

values can be any type

A dictionary is a set of key → value pairs. The key is hashed, so the lookup does not scan the data.

Python

```
emp = {"name": "Kavya Iyer", "dept": "HR", "salary": 97000}
print(emp["name"], emp.get("dept"), emp.get("city"), emp.get("city", "unknown"), len(emp), "salary" in emp)
try: emp["city"]
except KeyError as e: print("KeyError:", e)
```

OUTPUT

```
Kavya Iyer HR None unknown 3 True
KeyError: 'city'
```

`d[key]` raises `KeyError` when missing; `d.get(key, default)` never does.

Changing a dictionary

Python

```
emp["city"] = "Pune"                # add
emp["salary"] += 3000               # update
emp.update({"dept": "Finance", "grade": "M2"})
removed = emp.pop("grade")          # remove and return
del emp["city"]
print(emp, removed)
print(emp.setdefault("level", "L3"), emp.setdefault("dept", "X"), emp)  # set only if missing
```

OUTPUT

```
{'name': 'Kavya Iyer', 'dept': 'Finance', 'salary': 100000} M2
L3 Finance {'name': 'Kavya Iyer', 'dept': 'Finance', 'salary': 100000, 'level': 'L3'}
```

Method	Effect
<code>d[k] = v</code>	Add or overwrite
<code>d.get(k, default)</code>	Safe read
<code>d.setdefault(k, default)</code>	Read; insert default if missing
<code>d.update(other)</code> or <code>d = other</code>	Merge (right side wins)
<code>d.pop(k[, default])</code>	Remove and return
<code>d.popitem()</code>	Remove the last pair
<code>del d[k]</code>	Delete (KeyError if missing)
<code>d.keys()</code> , <code>d.values()</code> , <code>d.items()</code>	Views
<code>d.copy()</code>	Shallow copy
<code>dict.fromkeys(keys, value)</code>	Create with the same value

Looping

Python

```
prices = {"Pen": 10, "Notebook": 45, "Stapler": 180}
for k in prices: print(k, end=" ")
print()
for k, v in prices.items(): print(f"{k:<10}{v:>5}")
print(list(prices.values()), sum(prices.values()), max(prices, key=prices.get), sorted(prices.items(),
key=lambda kv: -kv[1]))
merged = prices | {"Pen": 12, "Folder": 30} # Python 3.9+
print(merged)
print({v: k for k, v in prices.items()}) # invert (values must be unique and
hashable)
```

OUTPUT

Pen Notebook Stapler

Pen 10

Notebook 45

Stapler 180

[10, 45, 180] 235 Stapler [('Stapler', 180), ('Notebook', 45), ('Pen', 10)]

{'Pen': 12, 'Notebook': 45, 'Stapler': 180, 'Folder': 30}

{10: 'Pen', 45: 'Notebook', 180: 'Stapler'}

Counting and grouping

The most common dictionary jobs are **count** and **group**:

Python

```
from collections import Counter, defaultdict
words = "the cat and the hat and the bat".split()
manual = {}
for w in words: manual[w] = manual.get(w, 0) + 1 # classic pattern
print(manual)
c = Counter(words)
print(c, c.most_common(2), c["the"], c["dog"])
groups = defaultdict(list) # missing key starts as an empty list
for w in words: groups[len(w)].append(w)
print(dict(groups))
```

OUTPUT

{'the': 3, 'cat': 1, 'and': 2, 'hat': 1, 'bat': 1}

Counter({'the': 3, 'and': 2, 'cat': 1, 'hat': 1, 'bat': 1}) [('the', 3), ('and', 2)] 3 0

{3: ['the', 'cat', 'and', 'the', 'hat', 'and', 'the', 'bat']}

Python

```
import pandas as pd
s = pd.read_csv("superstore_sales.csv")
by_region = Counter(s["region"]) # count rows
revenue = defaultdict(float)
for r, amount in zip(s["region"], s["sales"]): revenue[r] += amount
print(by_region.most_common())
print({k: round(v / 1e6, 1) for k, v in sorted(revenue.items(), key=lambda kv: -kv[1])}, "(sales in millions)")
print(s.groupby("region")["sales"].sum().div(1e6).round(1).sort_values(ascending=False).to_dict()) # same in pandas
```

OUTPUT

```
[('South', 1656), ('West', 1614), ('North', 1530), ('East', 1523)]
{'South': 54.0, 'West': 52.0, 'North': 49.0, 'East': 46.5} (sales in millions)
{'South': 54.0, 'West': 52.0, 'North': 49.0, 'East': 46.5}
```

Nested dictionaries and JSON-like records

Real data (API responses, configs, documents) nests dictionaries and lists. Python's `json` module reads and writes exactly this structure.

Python

```
import json
order = {"order_id": "0-1001", "customer": {"id": "C7", "name": "Asha", "city": "Pune"},
        "items": [{"sku": "P1", "qty": 2, "price": 450.0}, {"sku": "P2", "qty": 1, "price": 1200.0}]}
total = sum(i["qty"] * i["price"] for i in order["items"])
print(order["customer"]["city"], len(order["items"]), total, order.get("shipping", {}).get("city", "n/a"))
text = json.dumps(order, indent=2)
print(text[:90].replace("\n", " "), "...", json.loads(text) == order)
```

OUTPUT

```
Pune 2 2100.0 n/a
{  "order_id": "0-1001",  "customer": {      "id": "C7",      "name": "Asha",      "city": ...
True
```

A **list of dictionaries** is the standard row format, and it converts straight to a DataFrame:

Python

```
rows = [{"sku": "P1", "qty": 2, "price": 450.0}, {"sku": "P2", "qty": 1, "price": 1200.0}, {"sku": "P3", "qty": 5, "price": 80.0}]
df = pd.DataFrame(rows)
df["value"] = df.qty * df.price
print(df)
print(df.set_index("sku")["value"].to_dict(), df.to_dict("records")[0])
```

OUTPUT

	sku	qty	price	value
0	P1	2	450.0	900.0
1	P2	1	1200.0	1200.0
2	P3	5	80.0	400.0

```
{'P1': 900.0, 'P2': 1200.0, 'P3': 400.0} {'sku': 'P1', 'qty': 2, 'price': 450.0, 'value': 900.0}
```

Lookup tables and dispatch

Python

```
tier = {"North": "A", "South": "A", "East": "B", "West": "C"} # mapping/lookup
print(s["region"].map(tier).value_counts().to_dict())
ops = {"add": lambda a, b: a + b, "mul": lambda a, b: a * b} # dictionary of functions
print(ops["add"](3, 4), ops["mul"](3, 4))
```

OUTPUT

```
{'A': 3186, 'C': 1614, 'B': 1523}
7 12
```

NOTE Keys must be hashable

Strings, numbers and tuples work as keys; lists, sets and dicts do not. Use a tuple `("North", 2025)` for composite keys. Copying: `d.copy()` is shallow; use `copy.deepcopy` for nested data (Lesson 6).

Common mistakes

Mistake	Fix
<code>d["missing"]</code> raises <code>KeyError</code>	<code>d.get(k, default)</code> or <code>if k in d</code>
Changing a dictionary's size while looping over it	Loop over <code>list(d)</code> or build a new dict
<code>{}</code> for an empty set	<code>set()</code> ; <code>{}</code> is an empty dict
Duplicate keys in a literal	The last one silently wins
Assuming a shallow copy protects nested data	<code>copy.deepcopy</code>

Summary

KEY TAKEAWAYS

- A dict maps hashable keys to values with $O(1)$ average lookup and keeps insertion order.
- Use `get` for safe reads, `setdefault` / `defaultdict` for grouping, `Counter` for counting, `d | other` to merge.
- Loop with `items()` ; build with dict comprehensions `{k: f(v) for k, v in ...}` .
- Nested dict/list structures are JSON; a list of dicts converts to a DataFrame with `pd.DataFrame(rows)` .
- Dictionaries are lookup tables: `series.map(dict)` is a fast way to recode a column.

Quick quiz

6 questions · tap an answer to check it

1. What does `d.get("x", 0)` do when "x" is missing?

A Raises `KeyError`

B Returns 0

C Adds the key

D Returns `None`

Answer: B. `get` never raises; it returns the default.

2. Which is a valid dictionary key?

A `[1, 2]`

B `(1, 2)`

C `{1, 2}`

D `{"a": 1}`

Answer: B. Keys must be hashable (immutable).

3. Which tool counts occurrences most directly?

A `defaultdict(list)`

B `collections.Counter`

C `set`

D `zip`

Answer: B. `Counter(items).most_common()` .

4. What does `{"a": 1} | {"a": 2, "b": 3}` give?

A `{"a": 1, "b": 3}`

B `{"a": 2, "b": 3}`

C Error

D `{"a": 3}`

Answer: B. The right-hand dictionary wins on conflicts.

5. How do you loop over keys and values together?

A `for k in d.values()`

B `for k, v in d.items()`

C `for k, v in d`

D `for v in d.keys()`

Answer: B. `items()` yields (key, value) pairs.

6. What does `pd.DataFrame(list_of_dicts)` produce?

A A Series

B A table with one row per dictionary and columns from the keys

C A list

D An error

Answer: B. This is the standard way to turn JSON-like records into a table.

EXERCISE Dictionary problems

(1) Count the frequency of each character in `"banana"`. (2) Group Superstore product `category` into a dictionary of sets of `sub_category`. (3) Invert `{"a": 1, "b": 2}`. (4) Given a list of orders, compute the total per customer using `defaultdict`. (5) Merge two dictionaries where the second overrides the first.

[Show solution \(worked answer\)](#)

Python

```
from collections import Counter, defaultdict
print(Counter("banana"))
g = defaultdict(set)
for c, sc in zip(s["category"], s["sub_category"]): g[c].add(sc)
print({k: sorted(v) for k, v in g.items()})
print({v: k for k, v in {"a": 1, "b": 2}.items()})
orders = [("C1", 100), ("C2", 50), ("C1", 25)]
t = defaultdict(int)
for cid, amt in orders: t[cid] += amt
print(dict(t), {"x": 1} | {"x": 9, "y": 2})
```

INTERVIEW CORNER Questions you may be asked

Q: How does a dictionary achieve $O(1)$ lookup? It hashes the key to find a slot in a hash table; collisions are resolved by probing, so the average cost is constant.

Q: `d[k]` vs `d.get(k)` ? `d[k]` raises `KeyError` for a missing key; `get` returns `None` (or a default).

Q: Are dictionaries ordered? Yes, they preserve insertion order from Python 3.7 onwards.

Q: Count word frequencies in a text. `Counter(text.lower().split()).most_common(10)`.

Functions: Reusable Code, Arguments, Scope and Docstrings

Package logic into reusable functions. Learn `def`, parameters and arguments (positional, keyword, default, `*args`, `**kwargs`), return values, scope, docstrings and type hints, plus testing small functions on real data.

● Python class 7

✦ Python 3.13

Beginner

🕒 100 min

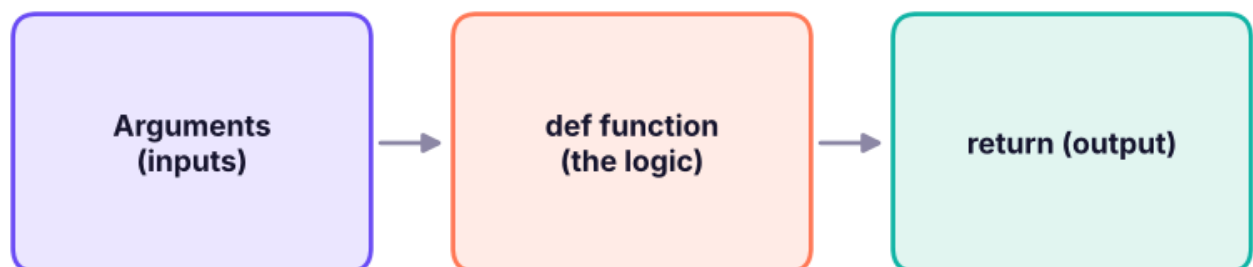
By the end of this lesson you can

- Define and call functions with positional, keyword and default arguments
- Use `*args` and `**kwargs` and return multiple values
- Explain local vs global scope and avoid the mutable-default trap
- Write docstrings and type hints that make functions self-explaining
- Build small, testable functions for data cleaning and metrics

Why functions?

A **function** is a named, reusable block of code. It removes copy-paste, makes code testable, and lets you give a name to an idea (`clean_phone` , `margin_pct`).

Anatomy of a function



A function takes inputs (arguments), runs its body, and gives back an output (return value).

Python

```
def greet(name):  
    return f"Hello, {name}!"  
print(greet("Asha"), greet("Ravi"))
```

OUTPUT

Hello, Asha! Hello, Ravi!

Parameters and arguments

Python

```
def margin_pct(profit, sales, decimals=1):
    """Profit as a percentage of sales."""
    return round(100 * profit / sales, decimals)

print(margin_pct(25, 200))           # positional
print(margin_pct(sales=200, profit=25)) # keyword: order does not matter
print(margin_pct(25, 200, decimals=3)) # default overridden
```

OUTPUT

12.5
12.5
12.5

Kind	Example	Notes
Positional	<code>f(1, 2)</code>	Matched by order
Keyword	<code>f(b=2, a=1)</code>	Matched by name; clearer
Default	<code>def f(a, b=10)</code>	Optional argument; defaults go last
<code>*args</code>	<code>def f(*args)</code>	Extra positional arguments as a tuple
<code>**kwargs</code>	<code>def f(**kwargs)</code>	Extra keyword arguments as a dict
Keyword-only	<code>def f(a, *, b)</code>	<code>b</code> must be named

Python

```
def total(*nums): return sum(nums)
def describe(**info): return ", ".join(f"{k}={v}" for k, v in info.items())
def report(title, *values, sep=" | ", **opts): return title + ": " + sep.join(map(str, values)) + f"{opts}"

print(total(1, 2, 3), total(), describe(city="Pune", year=2025))
print(report("Sales", 10, 20, 30, sep=",", currency="INR"))
nums = [4, 5, 6]; opts = {"decimals": 2}
print(total(*nums), margin_pct(7, 30, **opts)) # unpack a list / dict into arguments
```

OUTPUT

6 0 city=Pune, year=2025
Sales: 10,20,30 {'currency': 'INR'}
15 23.33

Return values

Python

```
def stats(values):
    n = len(values)
    mean = sum(values) / n
    var = sum((v - mean) ** 2 for v in values) / (n - 1)
    return min(values), max(values), round(mean, 2), round(var ** 0.5, 2)    # a tuple
lo, hi, mean, sd = stats([12, 15, 11, 18, 14])
print(lo, hi, mean, sd)

def say(): print("no return")
print(say())                                # a function with no return gives None
```

OUTPUT

```
11 18 14.0 2.74
no return
None
```

TIP Return, do not print

A function that **prints** can only be read by humans. One that **returns** can be tested, combined and reused. Print at the very end, outside the function.

Scope: local vs global

Python

```
rate = 0.18                                # global
def price_with_gst(price):
    gst = price * rate                      # reads the global
    return price + gst                    # gst is local: gone after the call
print(price_with_gst(1000))
try: print(gst)
except NameError as e: print("NameError:", e)

counter = 0
def bump():
    global counter                        # needed to *rebind* a global (rarely a good idea)
    counter += 1
bump(); bump(); print(counter)
```

OUTPUT

```
1180.0
NameError: name 'gst' is not defined
2
```

Rule of thumb (LEGB): Python looks in **L**ocal, then **E**nclosing, then **G**lobal, then **B**uilt-in scope. Prefer passing values in and returning values out.

The mutable-default trap

Python

```
def add_item(x, bucket=[]):          # BAD: one list shared by every call
    bucket.append(x); return bucket
print(add_item(1), add_item(2))

def add_item_ok(x, bucket=None):     # GOOD
    if bucket is None: bucket = []
    bucket.append(x); return bucket
print(add_item_ok(1), add_item_ok(2))
```

OUTPUT

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

Lists, dicts and sets as default values are created **once**, when `def` runs. Use `None` and create the object inside.

Docstrings and type hints

Python

```
def clean_phone(raw: str, country_code: str = "+91") → str:
    """Return a phone number as '+91 XXXXX XXXXX', or '' if it is not 10 digits.

    >>> clean_phone("98765-43210")
    '+91 98765 43210'
    """
    digits = "".join(ch for ch in raw if ch.isdigit())[-10:]
    if len(digits) != 10: return ""
    return f"{country_code} {digits[:5]} {digits[5:]}"

print(clean_phone("98765-43210"), repr(clean_phone("12345")), clean_phone("+91 (98765) 43210"))
help(clean_phone)
```

OUTPUT

```
+91 98765 43210 '' +91 98765 43210
Help on function clean_phone in module __main__:

clean_phone(raw: str, country_code: str = '+91') → str
    Return a phone number as '+91 XXXXX XXXXX', or '' if it is not 10 digits.

    >>> clean_phone("98765-43210")
    '+91 98765 43210'
```

Type hints do not change how the code runs, but editors and tools such as `mypy` use them, and they document intent. Write the docstring first line as a short summary; tools like `help()` and IDE hovers display it.

Functions are objects

Python

```
def square(x): return x * x
fn = square                                # assign, store in a list, pass as an argument
def apply_twice(f, x): return f(f(x))
print(fn(4), apply_twice(square, 3), [f.__name__ for f in (square, len, str)])

def make_multiplier(n):                    # a function that returns a function (a closure)
    def times(x): return x * n
    return times
double, triple = make_multiplier(2), make_multiplier(3)
print(double(7), triple(7))
```

OUTPUT

```
16 81 ['square', 'len', 'str']
14 21
```

Functions on real data

Small, pure functions are easy to test and to reuse across `apply`, comprehensions and pipelines.

Python

```
import pandas as pd
s = pd.read_csv("superstore_sales.csv", parse_dates=["order_date"])

def discount_band(d: float) → str:
    if d == 0: return "none"
    if d ≤ 0.10: return "low"
    if d ≤ 0.20: return "medium"
    return "high"

def summarise(df: pd.DataFrame, by: str) → pd.DataFrame:
    """Sales, profit and margin % per group."""
    g = df.groupby(by).agg(sales=("sales", "sum"), profit=("profit", "sum"), lines=("sales", "size"))
    g["margin_pct"] = (100 * g.profit / g.sales).round(1)
    return g.round(0).astype({"sales": int, "profit": int})

s["band"] = s["discount"].map(discount_band)
print(summarise(s, "band").sort_values("margin_pct", ascending=False))
print(summarise(s, "region").sort_values("sales", ascending=False))
```

OUTPUT

	sales	profit	lines	margin_pct
band				
none	123936290	21718963	3480	18.0

low	21864636	1286023	726	6.0
medium	35864461	-588882	1217	-2.0
high	19748182	-4063940	900	-21.0
	sales	profit	lines	margin_pct
region				
South	53952768	4702012	1656	9.0
West	51958520	4719833	1614	9.0
North	49018161	4555830	1530	9.0
East	46484119	4374490	1523	9.0

The same `summarise` works for any column. That reuse is the point: write once, call with `by="category"`, `by="segment"`, and so on.

Testing a function

Python

```
assert margin_pct(25, 200) == 12.5
assert clean_phone("12345") == ""
assert discount_band(0.15) == "medium"
print("all checks passed")
```

OUTPUT

all checks passed

`assert` is the quickest test. Larger projects use `pytest` (Lesson 14).

Recursion in one paragraph

A function can call itself; it needs a **base case** that stops the calls.

Python

```
def factorial(n):
    return 1 if n <= 1 else n * factorial(n - 1)
def fib(n, memo={}):
    # deliberate cache; see the lesson on functools.lru_cache
    if n < 2: return n
    if n not in memo: memo[n] = fib(n - 1) + fib(n - 2)
    return memo[n]
print(factorial(6), [fib(i) for i in range(12)], fib(60))
```

OUTPUT

720 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89] 1548008755920

Python limits recursion depth (about 1,000 calls), so prefer loops for deep or large problems.

Common mistakes

Mistake	Fix
Forgetting <code>return</code> (function gives <code>None</code>)	Check the value you expect back
Mutable default arguments	Use <code>None</code> and create inside
Calling <code>f</code> instead of <code>f()</code>	<code>f</code> is the function object; <code>f()</code> runs it
Modifying a global inside a function without <code>global</code>	<code>UnboundLocalError</code> ; pass and return values instead
Function does too many things	One job per function; compose small ones
Positional order confusion	Use keyword arguments for clarity

Summary

KEY TAKEAWAYS

- `def name(params): ... return value` defines a function; no `return` means it returns `None` .
- Arguments can be positional, keyword, default, `*args` (tuple) and `**kwargs` (dict); put defaults last.
- Variables created inside a function are local; read globals freely, but prefer parameters and return values.
- Never use a mutable object as a default; use `None` .
- Docstrings and type hints document intent; `assert` gives quick tests.
- Functions are objects: you can pass them around, store them and return them (closures).

Quick quiz

6 questions · tap an answer to check it

1. What does a function without a `return` statement return?

- A** 0
- B** None
- C** An empty string
- D** An error

Answer: B. Python returns `None` implicitly.

2. What is wrong with `def f(x, items=[]):` ?

- A Nothing
- B The default list is created once and shared across calls
- C Lists cannot be arguments
- D It must be a tuple

Answer: B. Use `items=None` and create the list inside.

3. What does `*args` collect?

- A Keyword arguments in a dict
- B Extra positional arguments in a tuple
- C Only the first argument
- D Nothing

Answer: B. `**kwargs` collects the keyword ones.

4. Which scope does Python search first?

- A Global
- B Built-in
- C Local
- D Enclosing only

Answer: C. LEGB: Local, Enclosing, Global, Built-in.

5. How do you call `f(a, b)` with a list `pair = [1, 2]` ?

A `f(pair)`

B `f(*pair)`

C `f(**pair)`

D `f[pair]`

Answer: B. `*` unpacks the list into positional arguments.

6. Why return values instead of printing them?

A Printing is slower

B Returned values can be tested, reused and combined

C Print is not allowed in functions

D Python requires it

Answer: B. Printing is for the final display only.

EXERCISE Function problems

(1) Write `is_prime(n)`. (2) Write `percent_change(old, new)` returning a rounded percentage and guard against `old == 0`. (3) Write `top_n(df, col, n=5)` that returns the top-n Superstore rows by a column. (4) Write a function that accepts any number of numbers and returns their average, or `None` if none are given. (5) Add `assert` tests for each.

Show solution (worked answer)

Python

```
def is_prime(n):
    return n > 1 and all(n % d for d in range(2, int(n ** 0.5) + 1))
def percent_change(old, new): return None if old == 0 else round(100 * (new - old) / old, 1)
def top_n(df, col, n=5): return df.nlargest(n, col)
def average(*nums): return sum(nums) / len(nums) if nums else None
assert [x for x in range(20) if is_prime(x)] == [2, 3, 5, 7, 11, 13, 17, 19]
assert percent_change(0, 5) is None and percent_change(200, 250) == 25.0
assert average() is None and average(2, 4) == 3
print(top_n(s, "sales", 3)[["order_id", "sales"]])
```

INTERVIEW CORNER Questions you may be asked

Q: Difference between an argument and a parameter? Parameters are the names in the `def` ; arguments are the values you pass when calling.

Q: What are `*args` and `**kwargs` ? They let a function accept any number of extra positional and keyword arguments; the first arrives as a tuple, the second as a dict.

Q: What is a closure? A function that remembers variables from the scope in which it was created, even after that scope has finished.

Q: Why is a mutable default argument dangerous? It is evaluated once at definition time, so state leaks between calls.

Comprehensions, Lambda, map/filter and Generators

Write compact, readable Python. Learn list, set and dict comprehensions, lambda with sorted/min/max, map and filter, any and all, generators and itertools, and when plain loops or pandas are the better choice.

● Python class 8

✦ Python 3.13

Beginner

🕒 100 min

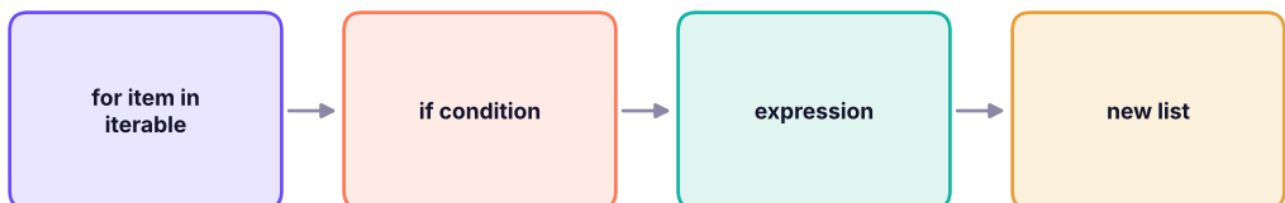
By the end of this lesson you can

- Write list, set and dictionary comprehensions with conditions and nesting
- Use lambda functions with sorted, min, max, map and filter
- Use any, all, zip, enumerate and sum with generator expressions
- Explain lazy evaluation and write generator functions with yield
- Use itertools and functools for common patterns and know when not to be clever

List comprehensions

A **comprehension** builds a collection from another in one expression: `[expression for item in iterable if condition]`.

[expression for item in iterable if condition]



A comprehension reads like the sentence "give me X for each item, if the condition holds".

Python

```
nums = [1, 2, 3, 4, 5, 6]
squares = [n * n for n in nums]
evens = [n for n in nums if n % 2 == 0]
labels = ["even" if n % 2 == 0 else "odd" for n in nums]           # if/else goes BEFORE the for
loop_version = []
for n in nums:
    if n % 2 == 0: loop_version.append(n * n)
print(squares, evens, labels, [n * n for n in nums if n % 2 == 0] == loop_version)
```

OUTPUT

```
[1, 4, 9, 16, 25, 36] [2, 4, 6] ['odd', 'even', 'odd', 'even', 'odd', 'even'] True
```

Python

```
words = [" Pune ", "delhi", " KOCHI", "goa "]
print([w.strip().title() for w in words])
print([(x, y) for x in (1, 2) for y in ("a", "b")])              # two loops
matrix = [[1, 2, 3], [4, 5, 6]]
print([v for row in matrix for v in row], [list(col) for col in zip(*matrix)])  # flatten, transpose
```

OUTPUT

```
['Pune', 'Delhi', 'Kochi', 'Goa']
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
[1, 2, 3, 4, 5, 6] [[1, 4], [2, 5], [3, 6]]
```

Set and dictionary comprehensions

Python

```
print({len(w) for w in ["tea", "cup", "mug", "plate"]})         # set
prices = {"Pen": 10, "Notebook": 45, "Stapler": 180}
print({k: round(v * 1.18, 2) for k, v in prices.items()})       # dict
print({k: v for k, v in prices.items() if v > 20}, {c: ord(c) for c in "abc"})
```

OUTPUT

```
{3, 5}
{'Pen': 11.8, 'Notebook': 53.1, 'Stapler': 212.4}
{'Notebook': 45, 'Stapler': 180} {'a': 97, 'b': 98, 'c': 99}
```

TIP Keep it readable

If a comprehension needs more than one condition or nested level, or does not fit comfortably on one or two lines, use a normal loop or a helper function. Clever is not the goal; clear is.

Lambda functions

A **lambda** is a small anonymous function: `lambda args: expression` .

Python

```
sq = lambda x: x * x                                # fine to learn with, but def is better for named functions
print(sq(5), (lambda a, b: a + b)(2, 3))
people = [("Asha", 29, 62000), ("Ravi", 35, 88000), ("Meera", 31, 75000)]
print(sorted(people, key=lambda p: p[2], reverse=True))    # sort by salary
print(min(people, key=lambda p: p[1]), max(people, key=lambda p: p[2])[0])
print(sorted(["banana", "kiwi", "apple", "fig"], key=lambda w: (len(w), w)))    # sort by length, then alphabet
```

OUTPUT

```
25 5
[('Ravi', 35, 88000), ('Meera', 31, 75000), ('Asha', 29, 62000)]
('Asha', 29, 62000) Ravi
['fig', 'kiwi', 'apple', 'banana']
```

Lambdas shine as the `key=` for `sorted` , `min` , `max` , and in `df.apply` .

map, filter, any, all

Python

```
nums = [3, 8, 12, 5, 20]
print(list(map(lambda x: x * 2, nums)), list(map(str, nums)), list(filter(lambda x: x > 6, nums)))
print([x * 2 for x in nums], [x for x in nums if x > 6])    # comprehensions say the same thing
print(any(n > 15 for n in nums), all(n > 0 for n in nums), sum(1 for n in nums if n % 2 == 0))
print(list(zip(["a", "b", "c"], [1, 2, 3])), dict(zip("abc", range(3))), list(enumerate("xyz", 1)))
```

OUTPUT

```
[6, 16, 24, 10, 40] ['3', '8', '12', '5', '20'] [8, 12, 20]
[6, 16, 24, 10, 40] [8, 12, 20]
True True 3
[('a', 1), ('b', 2), ('c', 3)] {'a': 0, 'b': 1, 'c': 2} [(1, 'x'), (2, 'y'), (3, 'z')]
```

`map` and `filter` return lazy iterators; wrap with `list(...)` . Most Python programmers prefer comprehensions, but you will see both.

Generators: lazy sequences

A **generator expression** uses parentheses and produces items one at a time, without building the whole list in memory.

Python

```
import sys
big_list = [n * n for n in range(1_000_000)]
big_gen = (n * n for n in range(1_000_000))
print(sys.getsizeof(big_list) // 1000, "KB list vs", sys.getsizeof(big_gen), "bytes generator")
print(sum(n * n for n in range(1_000_000)), next(iter(big_gen)), next(big_gen))
```

OUTPUT

```
8448 KB list vs 200 bytes generator
333332833333500000 0 1
```

A **generator function** uses `yield`. It pauses at each `yield` and resumes on the next request:

Python

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
print(list(countdown(5)))
def read_in_batches(items, size):
    for i in range(0, len(items), size):
        yield items[i:i + size]
for batch in read_in_batches(list(range(10)), 4): print(batch)
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
import itertools
print(list(itertools.islice(fib(), 12)))
```

OUTPUT

```
[5, 4, 3, 2, 1]
[0, 1, 2, 3]
[4, 5, 6, 7]
[8, 9]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Generators are how Python reads large files line by line, streams API pages and chains pipeline steps with constant memory. Once consumed, a generator is empty:

Python

```
g = (x for x in range(3))
print(list(g), list(g))
```

OUTPUT

```
[0, 1, 2] []
```

itertools and functools highlights

Python

```
import itertools, functools, operator
from collections import Counter
print(list(itertools.chain([1, 2], [3], [4, 5])), list(itertools.combinations("ABC", 2)),
list(itertools.product("AB", [0, 1])))
print(list(itertools.accumulate([10, 20, 5, 15])), functools.reduce(operator.mul, [1, 2, 3, 4, 5]))
for key, grp in itertools.groupby(sorted(["ant", "bee", "ape", "bat"]), key=lambda w: w[0]): print(key,
list(grp))
@functools.lru_cache(maxsize=None)                                # memoise a function: remember results of earlier
calls
def slow_square(n): return n * n
print([slow_square(n) for n in (4, 4, 5, 4)], slow_square.cache_info())
```

OUTPUT

```
[1, 2, 3, 4, 5] [('A', 'B'), ('A', 'C'), ('B', 'C')] [('A', 0), ('A', 1), ('B', 0), ('B', 1)]
[10, 30, 35, 50] 120
a ['ant', 'ape']
b ['bat', 'bee']
[16, 16, 25, 16] CacheInfo(hits=2, misses=2, maxsize=None, currsize=2)
```

Real data examples

Python

```
import pandas as pd
s = pd.read_csv("superstore_sales.csv", parse_dates=["order_date"])
regions = sorted(s["region"].unique())
print({r: round(float(s.loc[s.region == r, "sales"].sum()) / 1e6, 1) for r in regions})
# dict comprehension
top_customers = [c for c, n in s["customer_id"].value_counts().items() if n >= 18][:5]
print(top_customers, len([c for c, n in s["customer_id"].value_counts().items() if n >= 18]))
rows = s.head(4)[["order_id", "sales", "profit"]].to_dict("records")
print([{"r": r, "margin": round(100 * r["profit"] / r["sales"], 1)} for r in rows]) #
enrich records
print(sorted(s["category"].unique(), key=lambda c: -s.loc[s.category == c, "sales"].sum())) #
sort by a computed key
```

OUTPUT

```
{'East': 46.5, 'North': 49.0, 'South': 54.0, 'West': 52.0}
['C0719', 'C0149', 'C0126', 'C0384', 'C0257'] 21
[{'order_id': 'IN-2023-00001', 'sales': 36000.0, 'profit': 3472.55, 'margin': 9.6}, {'order_id':
'IN-2023-00001', 'sales': 3850.0, 'profit': 1169.54, 'margin': 30.4}, {'order_id': 'IN-2023-
00001', 'sales': 62080.0, 'profit': 6044.47, 'margin': 9.7}, {'order_id': 'IN-2023-00002',
'sales': 5216.0, 'profit': 450.16, 'margin': 8.6}]
['Technology', 'Furniture', 'Office Supplies']
```

Which tool when?

Situation	Prefer
Transform or filter a small list	List comprehension
Sort by a rule	<code>sorted(..., key=lambda ...)</code>
Huge or streaming data	Generator (expression or function)
Table columns	Vectorised pandas (<code>df["a"] * 2</code> , <code>np.where</code>), not comprehensions
Multi-step logic with side effects	Plain <code>for</code> loop
Reuse the same transformation	Named <code>def</code> function

Summary

KEY TAKEAWAYS

- `[expr for x in it if cond]` builds a list; `{...}` builds sets and dicts; the conditional expression (`a if c else b`) comes before `for` .
- `lambda` is an anonymous one-expression function, mostly used as `key=` and in `apply` .
- `map` / `filter` are lazy; comprehensions are usually clearer. `any` , `all` , `sum` work well with generator expressions.
- Generators (`( ... )` or `yield`) produce values lazily, use almost no memory and can be infinite; they can be consumed only once.
- `itertools` and `functools` (`chain` , `combinations` , `groupby` , `lru_cache` , `reduce`) cover many patterns; keep code readable and vectorise table work.

Quick quiz

6 questions · tap an answer to check it

1. What does `[x * 2 for x in range(4) if x != 2]` give?

- A [0, 2, 4, 6]
- B [0, 2, 6]
- C [0, 4, 6]
- D [2, 6]

Answer: B. Item 2 is filtered out before doubling.

2. Where does an `if ... else` go in a comprehension?

- A** After the for
- B** Before the for, as part of the expression
- C** It is not allowed
- D** At the end

Answer: B. `[a if cond else b for x in it]` .

3. What is the main advantage of a generator expression over a list comprehension?

- A** It is faster to type
- B** It is lazy and uses very little memory
- C** It supports indexing
- D** It can be reused

Answer: B. Values are produced one at a time.

4. What does `sorted(people, key=lambda p: p[2])` do?

- A** Sorts by the first item
- B** Sorts by the third item of each tuple
- C** Reverses the list
- D** Raises an error

Answer: B. `key` supplies the sort value.

5. What happens when you iterate a generator twice?

- A** It restarts
- B** The second pass is empty because it was consumed
- C** It raises an error
- D** It doubles the values

Answer: B. Create a new generator or use a list.

6. Which is usually best for adding a column to a DataFrame?

- A** A for loop over rows
- B** A vectorised column operation
- C** A generator
- D** A lambda inside a nested loop

Answer: B. Vectorised code is shorter and much faster.

EXERCISE Comprehension problems

(1) List the squares of odd numbers from 1 to 20. (2) Build a dictionary from words to their lengths for words longer than 3 letters. (3) Flatten `[[1,2],[3],[4,5,6]]` . (4) Sort `["pune","Delhi","goa","Kochi"]` case-insensitively. (5) Write a generator that yields only the primes below 50. (6) From the Superstore data, list sub-categories with total profit below zero.

Show solution (worked answer)

Python

```
print([n * n for n in range(1, 21) if n % 2])
print({w: len(w) for w in ["tea", "plate", "mug", "bottle"] if len(w) > 3})
print([v for part in [[1, 2], [3], [4, 5, 6]] for v in part])
print(sorted(["pune", "Delhi", "goa", "Kochi"], key=str.lower))
def primes(limit):
    for n in range(2, limit):
        if all(n % d for d in range(2, int(n ** 0.5) + 1)): yield n
print(list(primes(50)))
loss = s.groupby("sub_category")["profit"].sum()
print(list(loss[loss < 0].index))
```

INTERVIEW CORNER Questions you may be asked

Q: List comprehension vs generator expression? Both iterate with the same syntax; the list builds everything now in memory, the generator yields lazily and is single-use.

Q: When should you avoid a comprehension? When logic needs several statements, side effects, or deep nesting; readability matters more than brevity.

Q: What does `yield` do? It turns a function into a generator: the function pauses, hands back a value, and resumes where it left off on the next call.

Q: How do you flatten a list of lists? `[x for sub in lists for x in sub]` or `itertools.chain.from_iterable(lists)` .

Object-Oriented Python: Classes, Objects, Inheritance and Dataclasses

Model real things as objects. Learn classes, **init**, attributes and methods, encapsulation, inheritance, polymorphism, special (dunder) methods, class and static methods, properties and dataclasses, with a small data-pipeline example.

● Python class 9

✦ Python 3.13

Intermediate

🕒 110 min

By the end of this lesson you can

- Define classes with attributes and methods and create objects
- Use inheritance, `super()` and polymorphism to share and customise behaviour
- Implement **str**, **repr**, **eq** and **len** to make objects behave like built-ins
- Use `@property`, `@classmethod` and `@staticmethod` appropriately
- Choose between a plain class, a dataclass and a dictionary

Why classes?

A **class** bundles **data** (attributes) and **behaviour** (methods) into one unit. An **object** is one instance of a class. Libraries you already use are classes: a pandas `DataFrame` has data (rows) and methods (`.groupby()`).

Class and objects

class Account

Blueprint: owner, balance,
deposit(), withdraw()

acc1 = Account('Asha', 500)

Object with its own owner
and balance

acc2 = Account('Ravi', 900)

Another object, independent
of acc1

A class is the blueprint; objects are the things built from it, each with its own data.

Python

```
class Account:
    bank = "SkillBank"  # class attribute: shared by all objects

    def __init__(self, owner, balance=0):  # constructor: runs when you create an object
        self.owner = owner  # instance attributes: belong to this object
        self.balance = balance

    def deposit(self, amount):
        if amount <= 0: raise ValueError("amount must be positive")
        self.balance += amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance: raise ValueError("insufficient funds")
        self.balance -= amount
        return self.balance

a, b = Account("Asha", 500), Account("Ravi", 900)
a.deposit(250); b.withdraw(100)
print(a.owner, a.balance, b.owner, b.balance, Account.bank, a.bank)
try: a.withdraw(5000)
except ValueError as e: print("Error:", e)
```

OUTPUT

```
Asha 750 Ravi 800 SkillBank SkillBank
Error: insufficient funds
```

`self` is the object the method was called on. Python passes it automatically: `a.deposit(250)` is `Account.deposit(a, 250)`.

Special (dunder) methods

Methods with double underscores customise how objects behave with `print`, `=`, `len`, `+` and more.

Python

```
class Account:
    def __init__(self, owner, balance=0): self.owner, self.balance = owner, balance
    def __repr__(self): return f"Account({self.owner!r}, {self.balance})"  # for developers
    def __str__(self): return f"{self.owner}: ₹{self.balance:,.2f}"  # for print()
    def __eq__(self, other): return (self.owner, self.balance) == (other.owner, other.balance)
    def __lt__(self, other): return self.balance < other.balance  # enables sorting
    def __add__(self, other): return Account(self.owner + "&" + other.owner, self.balance +
other.balance)

x, y = Account("Asha", 5000), Account("Ravi", 9000)
print(x, "|", repr(x), "|", x = Account("Asha", 5000), "|", sorted([y, x]), "|", x + y)
```

OUTPUT

Asha: ₹5,000 | Account('Asha', 5000) | True | [Account('Asha', 5000), Account('Ravi', 9000)] |
Asha&Ravi: ₹14,000

Method	Used by
<code>__init__</code>	Creating the object
<code>__repr__</code> , <code>__str__</code>	<code>repr()</code> , <code>print()</code>
<code>__eq__</code> , <code>__lt__</code>	<code>=</code> , <code><</code> , sorting
<code>__len__</code> , <code>__getitem__</code>	<code>len(obj)</code> , <code>obj[i]</code>
<code>__iter__</code> , <code>__next__</code>	<code>for</code> loops
<code>__enter__</code> , <code>__exit__</code>	<code>with</code> blocks
<code>__call__</code>	Calling the object like a function

Encapsulation and properties

Python has no strict private members; a leading underscore (`_balance`) means "internal, please do not touch". A **property** exposes an attribute while keeping control over it.

Python

```
class Temperature:
    def __init__(self, celsius): self._c = celsius
    @property
    def celsius(self): return self._c
    @celsius.setter
    def celsius(self, v):
        if v < -273.15: raise ValueError("below absolute zero")
        self._c = v
    @property
    def fahrenheit(self): return self._c * 9 / 5 + 32          # computed, read-only
t = Temperature(25)
t.celsius = 30
print(t.celsius, t.fahrenheit)
try: t.celsius = -500
except ValueError as e: print("Error:", e)
```

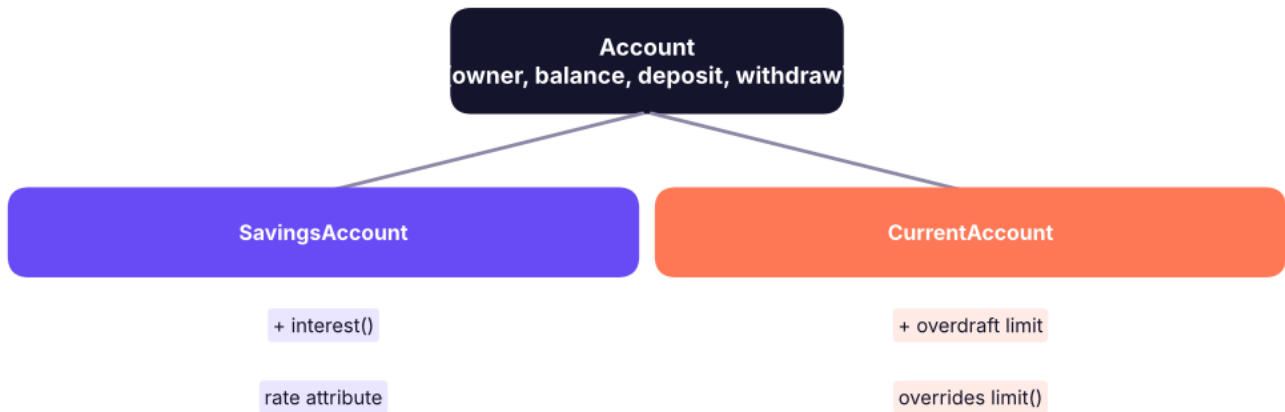
OUTPUT

30 86.0
Error: below absolute zero

Inheritance and polymorphism

A **child** class reuses a **parent** and adds or overrides behaviour. `super()` calls the parent's version.

Inheritance



Savings and Current accounts inherit everything from Account and add their own rules.

Python

```
class Account:
    def __init__(self, owner, balance=0): self.owner, self.balance = owner, balance
    def withdraw(self, amount):
        if amount > self.limit(): raise ValueError("limit exceeded")
        self.balance -= amount
    def limit(self): return self.balance
    def __str__(self): return f"{type(self).__name__}({self.owner}, {self.balance})"

class Savings(Account):
    def __init__(self, owner, balance=0, rate=0.04):
        super().__init__(owner, balance) # reuse the parent's setup
        self.rate = rate
    def interest(self): return round(self.balance * self.rate, 2)

class Current(Account):
    def __init__(self, owner, balance=0, overdraft=10000):
        super().__init__(owner, balance); self.overdraft = overdraft
    def limit(self): return self.balance + self.overdraft # override

accounts = [Savings("Asha", 50000), Current("Ravi", 2000)]
for acc in accounts: # polymorphism: same call, different
    acc.withdraw(5000) # behaviour
    print(acc, "| limit now", acc.limit())
print(accounts[0].interest(), isinstance(accounts[0], Account), issubclass(Current, Account),
[c.__name__ for c in Current.__mro__])
```

OUTPUT

```
Savings(Asha, 45000) | limit now 45000
Current(Ravi, -3000) | limit now 7000
1800.0 True True ['Current', 'Account', 'object']
```

Polymorphism means different classes respond to the same method name in their own way, so code like the loop above works with any account type.

NOTE Composition often beats inheritance

"Is-a" relationships (a Savings **is an** Account) suit inheritance. "Has-a" relationships (a Pipeline **has** steps) suit composition: store other objects as attributes. Prefer composition when in doubt.

Class and static methods

Python

```
class Order:
    GST = 0.18
    def __init__(self, amount): self.amount = amount
    @classmethod
    def from_string(cls, text):
        # alternative constructor: "₹1,200"
        return cls(float(text.replace("₹", "").replace(",", "")))
    @staticmethod
    def is_valid(amount): return amount > 0 # no self/cls: just a function in the class
    namespace
    def total(self): return round(self.amount * (1 + self.GST), 2)
o = Order.from_string("₹1,200")
print(o.amount, o.total(), Order.is_valid(-5))
```

OUTPUT

1200.0 1416.0 False

Dataclasses: classes for data

For classes that mainly store data, `@dataclass` generates `__init__`, `__repr__` and `__eq__` for you.

Python

```
from dataclasses import dataclass, field, asdict

@dataclass(order=True)
class Employee:
    name: str
    dept: str
    salary: float = 0.0
    skills: list = field(default_factory=list) # mutable default done safely
    def raise_salary(self, pct): self.salary = round(self.salary * (1 + pct / 100), 2)

staff = [Employee("Kavya", "HR", 97000, ["Excel"]), Employee("Rohan", "IT", 120000), Employee("Asha",
"HR", 88000)]
staff[0].raise_salary(10)
print(staff[0], "|", staff[0] = Employee("Kavya", "HR", 106700.0, ["Excel"]), "|", max(staff,
key=lambda e: e.salary).name)
print(asdict(staff[1]), sorted(staff)[0].name)
import pandas as pd
print(pd.DataFrame([asdict(e) for e in staff]).groupby("dept")["salary"].mean())
```


OUTPUT

```
Employee(name='Kavya', dept='HR', salary=106700.0, skills=['Excel']) | True | Rohan
{'name': 'Rohan', 'dept': 'IT', 'salary': 120000, 'skills': []} Asha
dept
HR      97350.0
IT      120000.0
Name: salary, dtype: float64
```

A small data-pipeline class

Python

```
import pandas as pd

class CleaningPipeline:
    """Holds a list of cleaning steps and applies them in order."""
    def __init__(self): self.steps = []
    def add(self, name, func): self.steps.append((name, func)); return self    # return self allows
    chaining
    def run(self, df):
        for name, func in self.steps:
            before = len(df); df = func(df)
            print(f" {name:<18} rows {before} → {len(df)}")
        return df

raw = pd.DataFrame({"city": [" Pune", "pune ", "Delhi", None, "Delhi"], "sales": [100, 100, 250, 80,
-5]})
pipe = (CleaningPipeline()
        .add("drop missing city", lambda d: d.dropna(subset=["city"]))
        .add("tidy text", lambda d: d.assign(city=d.city.str.strip().str.title()))
        .add("remove negatives", lambda d: d[d.sales ≥ 0])
        .add("drop duplicates", lambda d: d.drop_duplicates()))
print(pipe.run(raw))
```

OUTPUT

```
drop missing city  rows 5 → 4
tidy text          rows 4 → 4
remove negatives   rows 4 → 3
drop duplicates    rows 3 → 2
   city  sales
0  Pune    100
2  Delhi    250
```

This is the idea behind scikit-learn pipelines (ML track): objects with `fit` / `transform` methods that you chain.

When to use what

Need	Use
A few related values	Tuple / dict
Record with named fields, no logic	<code>@dataclass</code> or <code>namedtuple</code>
Data plus behaviour, state that changes	Class
Share behaviour across related types	Inheritance (or composition)
One function	Just a function

Common mistakes

Mistake	Fix
Forgetting <code>self</code> in the method signature	<code>def method(self, ...)</code>
Forgetting <code>self.</code> when reading an attribute	<code>self.balance</code> , not <code>balance</code>
Mutable class attribute shared by all objects (<code>items = []</code>)	Create it in <code>__init__</code>
Overusing inheritance	Prefer composition and small functions
Missing <code>super().__init__()</code> in the child	Call it to initialise parent state
Comparing objects without <code>__eq__</code>	Compares identity, not content; use a dataclass

Summary

KEY TAKEAWAYS

- A class is a blueprint; `__init__` sets up each object; `self` refers to the current object.
- Dunder methods (`__str__` , `__repr__` , `__eq__` , `__lt__` , `__len__`) make objects behave like built-ins.
- Inheritance reuses and overrides parent behaviour with `super()` ; polymorphism lets one call work across types.
- `@property` guards attributes; `@classmethod` builds alternative constructors; `@staticmethod` groups helpers.
- `@dataclass` removes boilerplate for data holders. Prefer composition over deep inheritance.

Quick quiz

6 questions · tap an answer to check it

1. What does `self` refer to?

- A The class
- B The specific object the method is called on
- C The module
- D The parent class

Answer: B. Python passes the instance automatically.

2. Which method runs when you create an object?

- A `__str__`
- B `__init__`
- C `__new__` only
- D `__call__`

Answer: B. `__init__` initialises the new instance.

3. What does `super().__init__(owner, balance)` do in a child class?

- A Deletes the parent
- B Runs the parent's constructor to set up inherited attributes
- C Creates a new class
- D Nothing

Answer: B. It reuses the parent's initialisation.

4. Which decorator gives a read-only computed attribute?

A `@staticmethod`

B `@property`

C `@classmethod`

D `@cache`

Answer: B. Access without parentheses.

5. What does `@dataclass` generate automatically?

A Only `__str__`

B `__init__` , `__repr__` and `__eq__`

C A database table

D Nothing

Answer: B. It removes boilerplate for data holders.

6. What is polymorphism?

A Having many parents

B Different classes responding to the same method name in their own way

C Copying a class

D Hiding attributes

Answer: B. The loop `for acc in accounts: acc.withdraw()` works for any account type.

EXERCISE OOP problems

(1) Create a `Rectangle` class with `area()` and `perimeter()`, and a `Square` subclass. (2) Add `__str__` and `__eq__` to a `Point(x, y)` class. (3) Build a `Stack` class (push, pop, peek, `__len__`). (4) Make a dataclass `Product` with price and `with_gst()`, then sort a list of products by price. (5) Write a `Counter`-like class that counts calls using a class attribute.

Show solution (worked answer)

Python

```
class Rectangle:
    def __init__(self, w, h): self.w, self.h = w, h
    def area(self): return self.w * self.h
    def perimeter(self): return 2 * (self.w + self.h)
class Square(Rectangle):
    def __init__(self, side): super().__init__(side, side)
print(Rectangle(3, 4).area(), Square(5).perimeter())
class Point:
    def __init__(self, x, y): self.x, self.y = x, y
    def __str__(self): return f"({self.x}, {self.y})"
    def __eq__(self, o): return (self.x, self.y) == (o.x, o.y)
print(Point(1, 2), Point(1, 2) == Point(1, 2))
class Stack:
    def __init__(self): self._d = []
    def push(self, x): self._d.append(x)
    def pop(self): return self._d.pop()
    def peek(self): return self._d[-1]
    def __len__(self): return len(self._d)
st = Stack(); st.push(1); st.push(2); print(st.pop(), len(st))
from dataclasses import dataclass
@dataclass(order=True)
class Product:
    price: float
    def with_gst(self): return round(self.price * 1.18, 2)
print(sorted([Product(500), Product(120)], Product(100).with_gst())
```

INTERVIEW CORNER Questions you may be asked

Q: Class attribute vs instance attribute? A class attribute is shared by all instances (defined in the class body); an instance attribute belongs to one object (set in `__init__`).

Q: What is the difference between `__str__` and `__repr__`? `__str__` is the readable form for users (`print`); `__repr__` is the unambiguous form for developers, ideally something that could recreate the object.

Q: Explain the four OOP pillars. Encapsulation (bundle and protect data), inheritance (reuse), polymorphism (same interface, different behaviour), abstraction (expose what, hide how).

Q: Inheritance or composition? Use inheritance for a true is-a relationship; use composition (objects holding other objects) for flexibility and lower coupling.

Files, Exceptions, JSON and CSV

Read and write text, CSV and JSON files safely with pathlib and with-blocks, handle errors with try/except/else/finally, raise your own exceptions, and log what happened, using a small "load a messy file" project.

● Python class 10

✦ Python 3.13

Intermediate

🕒 110 min

By the end of this lesson you can

- Open, read and write text files with with-blocks and pathlib
- Read and write CSV with the csv module and JSON with the json module
- Handle errors with try, except, else and finally, and catch specific exceptions
- Raise exceptions and define custom exception classes
- Use logging instead of print for programs that run unattended

Files and paths

Use `pathlib.Path` for paths (works on Windows, macOS and Linux) and a `with` block to open files: the file is closed automatically, even if an error happens.

Python

```
from pathlib import Path
folder = Path("demo_files"); folder.mkdir(exist_ok=True)
notes = folder / "notes.txt" # / joins path parts
with open(notes, "w", encoding="utf-8") as f: # "w" overwrites, "a" appends, "r" reads
    (default)
    f.write("Line one\n")
    f.writelines(["Line two\n", "Line three\n"])
with open(notes, "a", encoding="utf-8") as f: print("Line four", file=f)
print(notes.read_text(encoding="utf-8"))
print(notes.exists(), notes.name, notes.stem, notes.suffix, notes.stat().st_size, "bytes")
```

OUTPUT

```
Line one
Line two
Line three
Line four
```

```
True notes.txt notes .txt 39 bytes
```

Python

```
with open(notes, encoding="utf-8") as f:
    for i, line in enumerate(f, 1):
        print(i, line.rstrip("\n"))
with open(notes, encoding="utf-8") as f: lines = f.read().splitlines()
print(lines, sorted(p.name for p in folder.glob("*.txt")))
```

iterate line by line: constant memory for huge files

OUTPUT

```
1 Line one
2 Line two
3 Line three
4 Line four
['Line one', 'Line two', 'Line three', 'Line four'] ['notes.txt']
```

Mode	Meaning
"r"	Read (error if missing)
"w"	Write (creates or overwrites)
"a"	Append to the end
"x"	Create; error if it exists
"b" suffix	Binary ("rb" , "wb") for images, Excel, Parquet

TIP Always give the encoding

`encoding="utf-8"` avoids garbled ₹, é or Hindi text and avoids differences between Windows and Linux defaults.

CSV files

The standard `csv` module handles quoting and commas inside values. (For analysis you will usually use `pandas.read_csv` , Lesson 16.)

Python

```
import csv
rows = [{"id": 1, "city": "Pune", "note": "fast, friendly"}, {"id": 2, "city": "Delhi", "note": 'said "ok"'}]
with open(folder / "people.csv", "w", newline="", encoding="utf-8") as f:
    w = csv.DictWriter(f, fieldnames=["id", "city", "note"])
    w.writeheader(); w.writerows(rows)
print((folder / "people.csv").read_text(encoding="utf-8"))
with open(folder / "people.csv", newline="", encoding="utf-8") as f:
    for r in csv.DictReader(f): print(r)
```

every value is a string

OUTPUT

```
id,city,note
1,Pune,"fast, friendly"
2,Delhi,"said ""ok""""

{'id': '1', 'city': 'Pune', 'note': 'fast, friendly'}
{'id': '2', 'city': 'Delhi', 'note': 'said "ok"'}
```

Python

```
# Stream a real 6,000-line CSV without pandas: total sales per region
from collections import defaultdict
totals = defaultdict(float)
with open("superstore_sales.csv", newline="", encoding="utf-8") as f:
    for row in csv.DictReader(f): totals[row["region"]] += float(row["sales"])
print({k: round(v / 1e6, 1) for k, v in sorted(totals.items())})
```

OUTPUT

```
{'East': 46.5, 'North': 49.0, 'South': 54.0, 'West': 52.0}
```

JSON files

JSON maps to Python like this: object ↔ dict, array ↔ list, string ↔ str, number ↔ int/float, true/false ↔ True/False, null ↔ None.

Python

```
import json
config = {"source": "superstore_sales.csv", "regions": ["North", "South"], "min_sales": 1000, "debug":
False, "owner": None}
with open(folder / "config.json", "w", encoding="utf-8") as f: json.dump(config, f, indent=2)
with open(folder / "config.json", encoding="utf-8") as f: loaded = json.load(f)
print(loaded == config, loaded["regions"], json.dumps({"price": 4500.5, "ok": True, "x": None}))
print(json.dumps({"city": "Pune", "x": 1}, ensure_ascii=False))
```

OUTPUT

```
True ['North', 'South'] {"price": 4500.5, "ok": true, "x": null}
{"city": "Pune", "x": 1}
```

`json.dump` / `json.load` work with files; `json.dumps` / `json.loads` work with strings (the "s" is for string).
Dates and sets are not JSON types: convert them first (`d.isoformat()`, `list(s)`).

Exceptions

An **exception** is Python's way of reporting an error. Unhandled, it stops the program and prints a traceback. Handled, your program can recover.

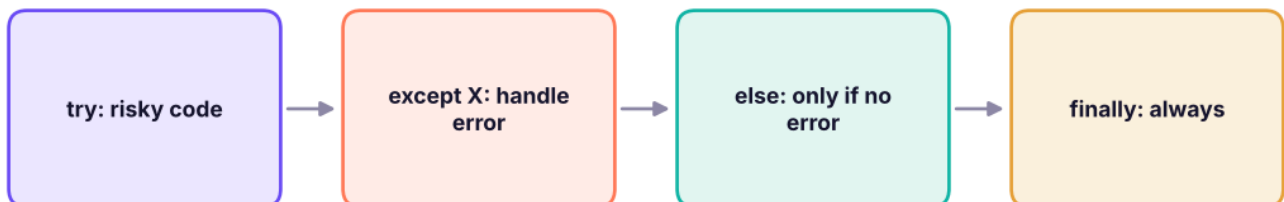
Python

```
def safe_ratio(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print(" cannot divide by zero")
        return None
    except TypeError as e:
        print(" wrong types:", e)
        return None
    else:
        # runs only when no exception happened
        return round(result, 2)
    finally:
        # always runs: clean-up
        print(" checked", a, b)
print(safe_ratio(10, 4)); print(safe_ratio(1, 0)); print(safe_ratio("10", 2))
```

OUTPUT

```
checked 10 4
2.5
cannot divide by zero
checked 1 0
None
wrong types: unsupported operand type(s) for /: 'str' and 'int'
checked 10 2
None
```

Exception handling flow



try / except / else / finally: which blocks run.

Common built-in exceptions

Exception	Typical cause
ValueError	Right type, wrong value: <code>int("abc")</code>
TypeError	Wrong type: <code>"a" + 1</code>
KeyError	Missing dict key
IndexError	List index out of range
FileNotFoundError	Missing file
ZeroDivisionError	Divide by zero
AttributeError	Object has no such attribute or method
ImportError / ModuleNotFoundError	Package not installed

Python

```
tests = [lambda: int("abc"), lambda: "a" + 1, lambda: {"x"}, lambda: [1][5], lambda:
open("missing.csv"), lambda: None.upper()]
for t in tests:
    try: t()
    except Exception as e: print(f"{type(e).__name__}<18} {e}")
```

OUTPUT

ValueError	invalid literal for int() with base 10: 'abc'
TypeError	can only concatenate str (not "int") to str
KeyError	'x'
IndexError	list index out of range
FileNotFoundError	[Errno 2] No such file or directory: 'missing.csv'
AttributeError	'NoneType' object has no attribute 'upper'

WATCH OUT

Do not swallow every error

`except: pass` or `except Exception: pass` hides bugs. Catch the **specific** exceptions you expect, and let unexpected ones surface (or log them and re-raise).

Raising your own exceptions

Python

```
class DataQualityError(Exception):
    """Raised when an input file fails validation."""

def validate(df):
    missing = {"order_id", "sales"} - set(df.columns)
    if missing: raise DataQualityError(f"missing columns: {sorted(missing)}")
    if (df["sales"] < 0).any(): raise ValueError("negative sales found")
    return True

import pandas as pd
for d in (pd.DataFrame({"order_id": [1], "sales": [5]}), pd.DataFrame({"order_id": [1]}),
pd.DataFrame({"order_id": [1], "sales": [-2]})):
    try: print("valid:", validate(d))
    except (DataQualityError, ValueError) as e: print(type(e).__name__, "→", e)
```

OUTPUT

```
valid: True
DataQualityError → missing columns: ['sales']
ValueError → negative sales found
```

Use `raise ... from e` to wrap a low-level error in a clearer one while keeping the original cause.

Logging

`print` is for interactive work. Programs that run on a schedule (ETL jobs, APIs) should use `logging`, which adds timestamps and levels and can write to files.

Python

```
import logging, io
stream = io.StringIO()
log = logging.getLogger("etl"); log.handlers.clear(); log.setLevel(logging.INFO)
h = logging.StreamHandler(stream); h.setFormatter(logging.Formatter("%(levelname)s | %(name)s | %(message)s")); log.addHandler(h)
log.debug("not shown: below INFO"); log.info("loaded 6,323 rows"); log.warning("12 rows had blank city")
try: 1 / 0
except ZeroDivisionError: log.exception("step failed") # logs the traceback too
print(stream.getvalue().splitlines()[:3])
```

OUTPUT

```
['INFO | etl | loaded 6,323 rows', 'WARNING | etl | 12 rows had blank city', 'ERROR | etl | step failed']
```

Levels from low to high: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`.

Project: load a messy file safely

Python

```
messy = """order_id,amount,city
1,1200,Pune
2,abc,Delhi
3,,Kochi
4,950,
5,-40,Goa
"""

(folder / "messy.csv").write_text(messy, encoding="utf-8")

def load_orders(path):
    good, bad = [], []
    with open(path, newline="", encoding="utf-8") as f:
        for lineno, row in enumerate(csv.DictReader(f), start=2):
            try:
                amount = float(row["amount"])
                if amount < 0: raise ValueError("negative amount")
                if not row["city"]: raise ValueError("blank city")
                good.append({"order_id": int(row["order_id"]), "amount": amount, "city": row["city"]})
            except ValueError as e:
                bad.append((lineno, str(e)))
    return good, bad

good, bad = load_orders(folder / "messy.csv")
print("good:", good); print("rejected:", bad)
with open(folder / "rejected.json", "w", encoding="utf-8") as f: json.dump(bad, f)
import shutil; shutil.rmtree(folder) # tidy up the demo folder
```

OUTPUT

```
good: [{'order_id': 1, 'amount': 1200.0, 'city': 'Pune'}]
rejected: [(3, "could not convert string to float: 'abc'"), (4, "could not convert string to
float: ''"), (5, 'blank city'), (6, 'negative amount')]
```

Real pipelines do exactly this: keep the good rows, quarantine and report the bad ones, never crash on one bad line.

Summary

KEY TAKEAWAYS

- Use `pathlib.Path` and `with open(path, mode, encoding="utf-8")` ; iterate a file line by line for big files.
- `csv.DictReader` / `DictWriter` handle quoting (values are strings); `json.load` / `dump` convert JSON ↔ Python.
- `try` / `except SpecificError` / `else` / `finally` ; never use a bare `except: pass` .
- `raise` your own exceptions (subclass `Exception`) to signal bad data clearly.
- Use `logging` with levels for unattended programs.

Quick quiz

6 questions · tap an answer to check it

1. Why use `with open(...)` as `f`: ?

- A It is faster
- B The file is closed automatically, even if an error occurs
- C It reads the whole file
- D It is required for CSV

Answer: B. The context manager handles clean-up.

2. What does mode `"w"` do to an existing file?

- A Appends
- B Overwrites it
- C Raises an error
- D Reads it

Answer: B. Use `"a"` to append and `"x"` to refuse overwriting.

3. When does the `else` block of a `try` statement run?

- A Always
- B Only if no exception was raised in `try`
- C Only after an exception
- D Never

Answer: B. `finally` is the block that always runs.

4. What does `json.loads` take?

A A file

B A string

C A dictionary

D A path

Answer: B. `load` takes a file object; `loads` takes a string.

5. What exception does `int("abc")` raise?

A `TypeError`

B `ValueError`

C `KeyError`

D `SyntaxError`

Answer: B. The type (`str`) is acceptable, the value is not.

6. Why prefer `logging` over `print` for scheduled jobs?

A It is shorter

B It adds levels and timestamps and can write to files and monitoring tools

C `print` is deprecated

D `logging` is faster

Answer: B. You can filter by level and keep a history.

EXERCISE File and error problems

(1) Write a function that counts lines, words and characters in a text file. (2) Read `superstore_sales.csv` with `csv.DictReader` and count rows per `category`. (3) Save that result as JSON and load it back. (4) Write `to_int(text, default=0)` that returns the default on `ValueError`. (5) Raise a custom `EmptyFileError` when a file has no data rows.

Show solution (worked answer)

Python

```
def wc(path):
    text = Path(path).read_text(encoding="utf-8")
    return len(text.splitlines()), len(text.split()), len(text)
from collections import Counter
with open("superstore_sales.csv", newline="", encoding="utf-8") as f:
    counts = Counter(r["category"] for r in csv.DictReader(f))
Path("cat.json").write_text(json.dumps(counts), encoding="utf-8")
print(json.loads(Path("cat.json").read_text(encoding="utf-8")))
def to_int(text, default=0):
    try: return int(text)
    except ValueError: return default
class EmptyFileError(Exception): pass
def check(path):
    with open(path, encoding="utf-8") as f:
        if len(f.read().splitlines()) < 2: raise EmptyFileError(path)
```

INTERVIEW CORNER Questions you may be asked

Q: How do you read a 10 GB file in Python? Iterate it line by line (`for line in f`), or read in chunks (`pd.read_csv(..., chunksize=...)`), so memory stays constant.

Q: Difference between `except Exception` and a bare `except` ? A bare `except` also catches `KeyboardInterrupt` and `SystemExit` ; catch specific exceptions where possible.

Q: What is `finally` used for? Clean-up that must always happen, such as closing connections or releasing locks.

Q: How do you handle bad rows in an ETL job? Validate each row, keep good rows, write rejects with the reason to a quarantine file or table, log counts, and alert if the reject rate is too high.

Modules, Packages, Virtual Environments and Git

Organise code into modules and packages, install libraries with pip or uv inside virtual environments, pin dependencies, use the standard library, and version-control your projects with Git and GitHub.

● Python class 10

✦ Python 3.13 · pip 25 · uv · Git 2.5x

Intermediate

🕒 90 min

By the end of this lesson you can

- Import modules in the different ways and explain if `name == "main"`
- Create a package with `init.py` and a sensible project layout
- Create virtual environments and install and pin packages with pip or uv
- Use key standard-library modules (os, pathlib, datetime, random, math, statistics, collections)
- Use the everyday Git workflow: init, add, commit, branch, push, pull request

Modules

A **module** is any `.py` file. Importing it gives you its functions, classes and variables.

Python

```
import math # whole module, use with prefix
from statistics import mean, median # specific names
import datetime as dt # alias
from collections import Counter as C
print(math.sqrt(144), math.pi, mean([3, 5, 10]), median([3, 5, 10]), dt.date(2025, 4, 1).strftime("%d %b %Y"), C("aab"))
print(math.ceil(4.2), math.floor(4.8), math.log10(1000), round(math.exp(1), 4), math.comb(5, 2))
```

OUTPUT

```
12.0 3.141592653589793 6 5 01 Apr 2025 Counter({'a': 2, 'b': 1})
5 4 3.0 2.7183 10
```

Style	When
<code>import pandas as pd</code>	Standard aliases (<code>np</code> , <code>pd</code> , <code>plt</code> , <code>sns</code>)
<code>from pathlib import Path</code>	A few names used often
<code>from module import *</code>	Avoid: hides where names come from

Writing your own module

Python

```
from pathlib import Path
Path("metrics.py").write_text('"""Reusable business metrics."""'
GST = 0.18

def margin_pct(profit, sales):
    return round(100 * profit / sales, 1) if sales else None

def with_gst(amount):
    return round(amount * (1 + GST), 2)

if __name__ == "__main__":
    # runs only when executed directly, not when imported
    print("self-test:", margin_pct(25, 200))
    '''', encoding="utf-8")
import importlib, sys
if "" not in sys.path: sys.path.insert(0, "") # notebooks and scripts already search the current
folder
import metrics
importlib.reload(metrics)
print(metrics.margin_pct(30, 120), metrics.with_gst(1000), metrics.__doc__)
import subprocess, sys
print(subprocess.run([sys.executable, "metrics.py"], capture_output=True, text=True).stdout.strip())
Path("metrics.py").unlink()
```

OUTPUT

```
25.0 1180.0 Reusable business metrics.
self-test: 12.5
```

`__name__` is `"__main__"` when the file is run as a script and the module name when imported. The guard lets one file be both a library and a script.

Where does Python look?

The current folder, then the folders in `sys.path` (including `site-packages`, where pip installs libraries).

`ModuleNotFoundError` usually means: wrong environment, package not installed, or a typo. Never name your own file `pandas.py` or `random.py`: it will shadow the real library.

Packages and project layout

A **package** is a folder of modules (with an `__init__.py`). A clean data project:

Text

```
churn-project/
├── pyproject.toml          # project metadata and dependencies
├── README.md
├── .gitignore             # venv, data, secrets, __pycache__
├── data/                  # raw/ and processed/ (usually not committed)
├── notebooks/            # exploration
├── src/churn/
│   ├── __init__.py
│   ├── load.py           # read and validate data
│   ├── features.py       # feature engineering
│   └── train.py          # model training
└── tests/
    └── test_features.py
```

Import with `from churn.features import add_tenure_band`. Notebooks for exploring; `.py` modules for anything reused or scheduled.

The standard library tour

Python

```
import os, random, statistics, datetime as dt, itertools, textwrap, uuid, time
random.seed(42)
print(random.randint(1, 6), random.choice(["N", "S", "E"]), random.sample(range(100), 3),
round(random.random(), 3))
print(statistics.stdev([2, 4, 4, 4, 5, 5, 7, 9]), statistics.mode([1, 2, 2, 3]))
today = dt.date(2025, 10, 27); due = today + dt.timedelta(days=45)
print(due, due.strftime("%A"), (dt.date(2026, 1, 1) - today).days, "days to new year")
print(os.cpu_count() > 0, os.getenv("HOME") is not None, textwrap.shorten("A very long product
description here", 20), len(str(uuid.uuid4())))
```

OUTPUT

```
6 N [3, 94, 35] 0.245
2.138089935299395 2
2025-12-11 Thursday 66 days to new year
True True A very long [...] 36
```

Module	Use
<code>pathlib</code> , <code>os</code> , <code>shutil</code> , <code>glob</code>	Files and folders
<code>datetime</code> , <code>zoneinfo</code> , <code>time</code>	Dates, time zones, timing
<code>math</code> , <code>statistics</code> , <code>random</code> , <code>decimal</code>	Numbers (<code>decimal</code> for money)
<code>collections</code> , <code>itertools</code> , <code>functools</code>	Data structures and iteration helpers
<code>csv</code> , <code>json</code> , <code>sqlite3</code> , <code>pickle</code>	Data formats and storage
<code>re</code>	Regular expressions
<code>logging</code> , <code>argparse</code> , <code>subprocess</code>	Scripts and tools
<code>urllib</code> , <code>http</code>	Web (most people use <code>requests</code> / <code>httpx</code>)
<code>concurrent.futures</code> , <code>asyncio</code>	Parallel and asynchronous work

pip, virtual environments and uv

Each project should have its **own virtual environment** so its library versions do not clash with other projects.

Text

```
# Create and activate an environment (built-in venv)
python -m venv .venv
source .venv/bin/activate          # macOS/Linux
.venv\Scripts\activate            # Windows

# Install, list, pin, reproduce
pip install pandas scikit-learn matplotlib
pip list
pip freeze > requirements.txt
pip install -r requirements.txt

# uv: a much faster modern alternative (by Astral)
uv init churn-project && cd churn-project
uv add pandas scikit-learn
uv run python src/churn/train.py
```

Virtual environments



The project isolation model: each project gets its own environment and its own library versions.

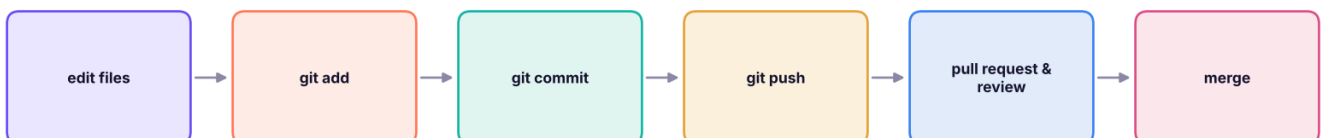
TIP Conda, Anaconda, Colab

Anaconda/Miniconda bundle Python with data libraries and the `conda` environment manager; popular in data science courses. **Google Colab** and **Kaggle Notebooks** run Python in the browser with libraries pre-installed: ideal for practice. Pick one tool per project and stick to it.

Git and GitHub essentials

Git tracks changes to files; **GitHub** hosts repositories online for backup, collaboration and your portfolio.

Everyday Git workflow



The everyday Git cycle.

Text

```
git init                # start tracking a folder
git status              # what changed?
git add analysis.py     # stage a file (git add . = everything)
git commit -m "Add churn EDA" # save a snapshot with a message
git log --oneline       # history
git switch -c feature/rfm # new branch
git push -u origin feature/rfm # upload the branch to GitHub
git pull                # download others' changes
git clone https://github.com/user/repo.git
git diff                # unstaged changes
git restore file.py     # discard local changes to a file
```

A `.gitignore` keeps junk and secrets out:

Text

```
.venv/
__pycache__/
.ipynb_checkpoints/
data/raw/
*.env
.DS_Store
```

WATCH OUT Never commit secrets

API keys, passwords and tokens belong in environment variables or a `.env` file listed in `.gitignore`. Read them with `os.getenv("OPENAI_API_KEY")`. If a key is ever pushed, revoke it immediately: deleting the commit is not enough.

Summary

KEY TAKEAWAYS

- A module is a `.py` file; a package is a folder of modules. Use standard aliases and avoid `import *`.
- `if __name__ == "__main__":` separates script code from importable code.
- One virtual environment per project; pin versions in `requirements.txt` or `pyproject.toml` (pip, uv or conda).
- The standard library covers files, dates, maths, data formats, logging and more.
- Git: add → commit → push → pull request → merge; use `.gitignore` and never commit secrets.

Quick quiz

6 questions · tap an answer to check it

1. What does `if __name__ == "__main__": do?`

- A Imports main
- B Runs the block only when the file is executed directly
- C Defines the main function
- D Speeds up the program

Answer: B. When imported, `__name__` is the module name.

2. Why use a virtual environment?

- A It makes Python faster
- B It isolates each project's library versions
- C It is required by Git
- D It encrypts the code

Answer: B. Different projects may need different versions.

3. How do you record the exact installed versions for others?

- A `pip list`
- B `pip freeze > requirements.txt`
- C `git add .`
- D `pip update`

Answer: B. Others recreate with `pip install -r requirements.txt`.

4. Why should you not name your file `pandas.py` ?

- A It is too long
- B It shadows the real pandas library on import
- C Git rejects it
- D It is reserved

Answer: B. Python finds your file first.

5. Which Git command uploads your commits to GitHub?

- A `git commit`
- B `git push`
- C `git add`
- D `git log`

Answer: B. Commit saves locally; push uploads.

6. Where should API keys go?

- A In the code
- B In environment variables or an ignored `.env` file
- C In the README
- D In a commit message

Answer: B. Never commit secrets.

EXERCISE Setup checklist

(1) Create a project folder with a virtual environment and install pandas and matplotlib. (2) Write a module `cleaning.py` with two functions and import it from a notebook. (3) Create a GitHub repository, add a `.gitignore` and a README, and push. (4) Create a branch, change the README and open a pull request to yourself.

Show solution (worked answer)

Follow the commands in this lesson. Check with `python -c "import pandas; print(pandas.__version__)"` and `git log --oneline`. A good README states the question, the data source, how to run the code and the key findings with one chart.

INTERVIEW CORNER Questions you may be asked

Q: Difference between a module, a package and a library? A module is one file, a package is a folder of modules, and a library is a collection of packages distributed together (e.g. on PyPI).

Q: How do you make a Python project reproducible? Virtual environment, pinned dependencies, a README with run instructions, fixed random seeds, version control, and data lineage.

Q: git merge vs git rebase? Merge combines histories with a merge commit; rebase replays your commits on top of another branch for a linear history. Never rebase shared branches.

Debugging, Testing, Regular Expressions and Code Quality

Find and fix bugs systematically, write automated tests with pytest, clean text with regular expressions, and keep code readable with PEP 8, type hints, linters and formatters.

● Python class 10

✦ Python 3.13 · pytest 9 · ruff

Intermediate

⌚ 100 min

By the end of this lesson you can

- Read a traceback and debug with print, breakpoints and pdb
- Write and run unit tests with assert and pytest, including parametrised tests
- Use regular expressions to search, extract, validate and replace text
- Apply PEP 8 naming and layout, type hints, and automatic tools (ruff, black, mypy)
- Use AI assistants to debug and review code responsibly

Reading a traceback

Read a traceback **from the bottom up**: the last line names the error, the lines above show where it happened.

Python

```
def average(values):
    return sum(values) / len(values)
def report(data):
    return f"Average: {average(data['scores']):.1f}"
print(report({"scores": []}))
```

OUTPUT

```
ZeroDivisionError: division by zero
```

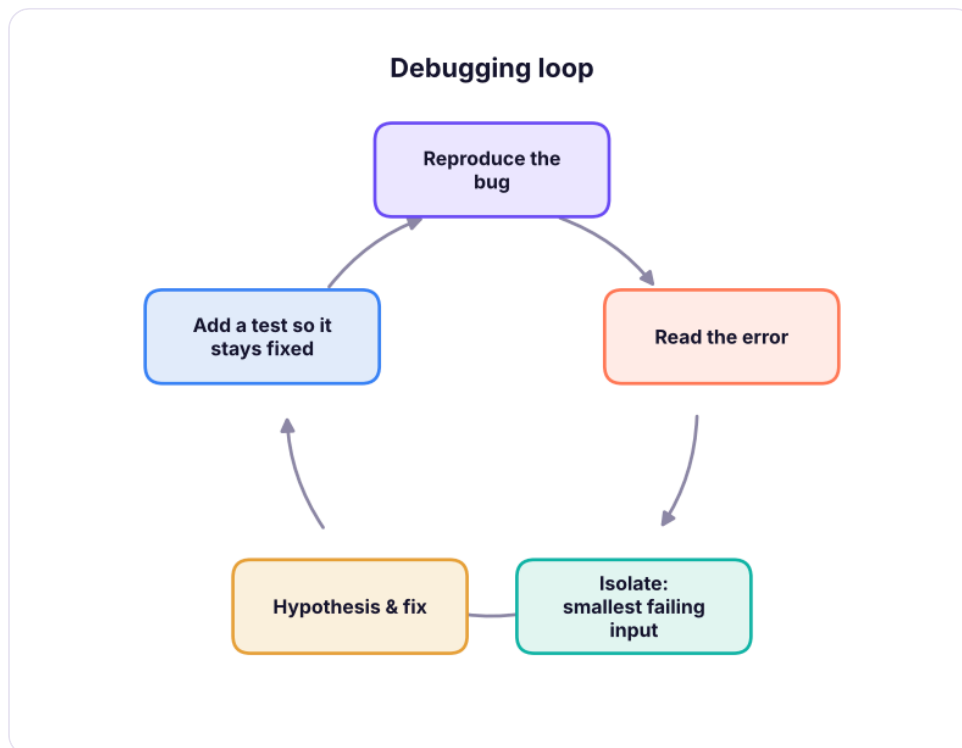
The last line says `ZeroDivisionError`; the frame above points at `average`, called from `report`. The fix is a guard for empty input:

Python

```
def average(values):
    return sum(values) / len(values) if values else None
print(average([], average([4, 6])))
```

OUTPUT

A debugging routine



A systematic debugging loop beats random changes.

Technique	How
Print / f-string debugging	<code>print(f"{i=} {total=}")</code> shows names and values
<code>breakpoint()</code>	Pauses and opens the <code>pdb</code> debugger at that line
IDE debugger	VS Code / PyCharm: breakpoints, step over/into, watch variables
Jupyter <code>%debug</code>	Post-mortem debugging after an error in a cell
Rubber-duck	Explain the code line by line to someone (or an AI)
Bisect	Comment out halves of the code to find the failing part

Python

```
total = 0
for i, v in enumerate([5, 10, 15]):
    total += v
    print(f"{i=} {v=} {total=}")  # the = specifier prints the name and the value
```

OUTPUT

```
i=0 v=5 total=5
i=1 v=10 total=15
i=2 v=15 total=30
```

pdb commands: `n` (next line), `s` (step into), `c` (continue), `p var` (print), `l` (list code), `q` (quit).

Testing

A **test** is code that checks other code. Tests catch regressions when you change things later.

Python

```
def clean_city(raw):
    return " ".join(raw.split()).title() if raw else None

assert clean_city(" new delhi ") == "New Delhi"
assert clean_city("") is None
print("asserts passed")
```

OUTPUT

asserts passed

pytest

`pytest` finds files named `test_*.py` and runs functions named `test_*`. Plain `assert` statements give detailed failure messages.

Python

```
from pathlib import Path
import subprocess, sys
Path("tests_demo").mkdir(exist_ok=True)
Path("tests_demo/cleaning.py").write_text('''
def clean_city(raw):
    return " ".join(raw.split()).title() if raw else None

def to_inr(value):
    return f"₹{value:,.0f}"
''', encoding="utf-8")
Path("tests_demo/test_cleaning.py").write_text('''
import pytest
from cleaning import clean_city, to_inr

@pytest.mark.parametrize("raw, expected", [("  pune", "Pune"), ("NEW  DELHI", "New Delhi"), ("", None),
(None, None)])
def test_clean_city(raw, expected):
    assert clean_city(raw) == expected

def test_to_inr():
    assert to_inr(1234567) == "₹1,234,567"

def test_bad_input():
    with pytest.raises(ValueError):          # a str cannot take the ',' number format
        to_inr("abc")

def test_deliberately_failing():
    assert to_inr(1000) == "₹1000"          # wrong expectation, to show a failure report
''', encoding="utf-8")
r = subprocess.run([sys.executable, "-m", "pytest", "-q", "--no-header", "-p", "no:cacheprovider",
"tests_demo"], capture_output=True, text=True)
lines = r.stdout.strip().splitlines()
print("\n".join([l for l in lines if "assert" in l or "passed" in l or "failed" in l][-4:]))
import shutil; shutil.rmtree("tests_demo")
```

OUTPUT

```
>      assert to_inr(1000) == "₹1000"          # wrong expectation, to show a failure report
E      AssertionError: assert '₹1,000' == '₹1000'
1 failed, 6 passed in 0.02s
```

The report says which test failed and shows both values: `'₹1,000' == '₹1000'`. The function is right, so the test expectation is fixed.

TIP What to test in data work

Cleaning functions, metric calculations, edge cases (empty, missing, negative, duplicates), schema checks on loaded data (columns, types, ranges), and that a pipeline's row counts reconcile. Tools such as **pandera** and **Great Expectations** validate DataFrames.

Regular expressions

A **regular expression** (regex) is a pattern for text. Python's `re` module searches, extracts, validates and replaces.

Pattern	Matches
<code>\d / \D</code>	A digit / a non - digit
<code>\w / \s</code>	Word character (letter, digit, _) / whitespace
<code>.</code>	Any character except newline
<code>[A-Z] , [^0-9]</code>	Character set / negated set
<code>*</code> , <code>+</code> , <code>?</code>	0 or more, 1 or more, 0 or 1
<code>{3}</code> , <code>{2,4}</code>	Exactly 3, between 2 and 4
<code>^</code> , <code>\$</code>	Start, end of string
<code>()</code>	Capture group; <code>(?P<name>)</code> named group
<code>\ </code>	Or

Python

```
import re
text = "Call 98765-43210 or +91 91234 56789; email asha.k@gmail.com, ravi@skillnavora.in. Order #A-1029 on 2025-04-07."
print(re.findall(r"\d{5}[- ]?\d{5}", text)) # phone numbers
print(re.findall(r"[\w.+-]+@[ \w-]+(?:\. [ \w-]+)+", text)) # emails (no trailing full stop)
m = re.search(r"(?P<y>\d{4})-(?P<m>\d{2})-(?P<d>\d{2})", text)
print(m.group(), m["y"], m.groupdict())
print(re.sub(r"\d", "X", "PAN ABCDE1234F"), re.split(r"[;,\s*]", "a, b;c ,d"))
print(bool(re.fullmatch(r"[A-Z]{5}\d{4}[A-Z]", "ABCDE1234F")), bool(re.fullmatch(r"[A-Z]{5}\d{4}[A-Z]", "ABCD1234F"))) # PAN format
```

OUTPUT

```
['98765-43210', '91234 56789']
['asha.k@gmail.com', 'ravi@skillnavora.in']
2025-04-07 2025 {'y': '2025', 'm': '04', 'd': '07'}
PAN ABCDEXXXXF ['a', 'b', 'c ', 'd']
True False
```

Always use **raw strings** `r"..."` for patterns, so `\d` is not treated as an escape. Test patterns at [regex101.com](#).

Regex on a DataFrame

Python

```
import pandas as pd
c = pd.DataFrame({"name": ["Asha", "Ravi", "Meera", "Kabir", "Zoya"],
                  "email": ["asha@gmail.com", "ravi@company", "MEERA@Yahoo.co.in", "Kabir @gmail.com",
                             None],
                  "phone": ["98765 43210", "+91-91234-56789", "12345", "9123456789", "0987654321"]})
c["email_ok"] = c["email"].str.fullmatch(r"[\w.+-]+@[ \w-]+(?:\.[\w-]+)+", na=False)
c["domain"] = c["email"].str.lower().str.extract(r"@([\w.-]+)$")[0]
digits = c["phone"].str.replace(r"\D", "", regex=True).str[-10:]
c["phone_ok"] = digits.str.fullmatch(r"[6-9]\d{9}")
print(c[["name", "email_ok", "domain", "phone_ok"]])
```

OUTPUT

	name	email_ok	domain	phone_ok
0	Asha	True	gmail.com	True
1	Ravi	False	company	True
2	Meera	True	yahoo.co.in	False
3	Kabir	False	gmail.com	True
4	Zoya	False	NaN	False

pandas string methods (`.str.contains` , `.str.extract` , `.str.replace` , `.str.fullmatch`) accept regex and run on the whole column.

Code quality

PEP 8 is Python's style guide. The essentials:

Rule	Example
<code>snake_case</code> for variables and functions	<code>total_sales</code> , <code>clean_city()</code>
<code>PascalCase</code> for classes	<code>CleaningPipeline</code>
<code>UPPER_CASE</code> for constants	<code>GST_RATE = 0.18</code>
4-space indentation, lines $\leq$ 79–100 chars	
Spaces around operators, after commas	<code>x = a + b</code> , <code>f(a, b)</code>
Imports at the top, grouped std / third-party / local	
Meaningful names over comments	<code>days_since_last_order</code> , not <code>d</code>

Let tools do the work:

Text

```
pip install ruff mypy
ruff check .          # lint: unused imports, undefined names, bad patterns
ruff format .         # auto-format (black-compatible)
mypy src/             # check type hints
pre-commit install    # run them automatically before each commit
```

NOTE Using AI to debug

Paste the **full traceback and the smallest code that reproduces it** into an AI assistant (Claude, ChatGPT, Copilot, Gemini). Ask it to explain the cause before proposing a fix, then run the fix and add a test. Never paste secrets or confidential data, and check that suggested library functions really exist.

Summary

KEY TAKEAWAYS

- Read tracebacks bottom-up; reproduce, isolate, hypothesise, fix, then add a test.
- Debug with `print(f"{x=}")`, `breakpoint()`, an IDE debugger or `%debug`.
- `pytest` runs `test_*` functions with plain `assert`; use `parametrize` for many cases and `pytest.raises` for errors.
- Regex: `re.search`, `findall`, `fullmatch`, `sub`, `split`; raw strings; `pandas .str` methods accept regex.
- Follow PEP 8 and automate it with `ruff`, a formatter and `mypy`.

Quick quiz

6 questions · tap an answer to check it

1. Where do you start reading a Python traceback?

- A The first line
- B The last line, which names the error**
- C The middle
- D The import lines

Answer: B. Then read upward to find where it happened.

2. What does `print(f"{total=}")` print when total is 5?

A 5

B `total=5`

C `{total=5}`

D An error

Answer: B. The `=` specifier is a debugging shortcut.

3. Which file name will pytest discover by default?

A `cleaning_tests.py`

B `test_cleaning.py`

C `tests.txt`

D `cleaning.py`

Answer: B. Files `test_*.py` or `*_test.py`, functions `test_*`.

4. What does `\d{3}` match?

A The letter d three times

B Exactly three digits

C Up to three digits

D Any three characters

Answer: B. `\d` is a digit; `{3}` repeats it exactly three times.

5. Why use `r"..."` for regex patterns?

- A It runs faster
- B Backslashes are kept literally instead of being treated as escapes**
- C It makes the pattern case-insensitive
- D It is required by pandas

Answer: B. Raw strings avoid double escaping.

6. Which naming follows PEP 8 for a function?

- A CalculateMargin
- B calculate_margin**
- C CALCULATE_MARGIN
- D calculateMargin

Answer: B. snake_case for functions and variables.

EXERCISE Practice

(1) Write a regex that extracts all hashtags from a tweet. (2) Validate Indian PIN codes (6 digits, not starting with 0). (3) Write three pytest tests for a `percent_change` function, including the zero case. (4) Run `ruff check` on one of your notebooks exported as `.py` and fix the warnings.

Show solution (worked answer)

Python

```
import re
print(re.findall(r"#\w+", "Loving #Python and #DataScience!"))
pin = re.compile(r"[1-9]\d{5}")
print(bool(pin.fullmatch("411001")), bool(pin.fullmatch("011001")))
# test_metrics.py
# def test_up(): assert percent_change(100, 120) == 20
# def test_down(): assert percent_change(100, 80) == -20
# def test_zero(): assert percent_change(0, 5) is None
```

INTERVIEW CORNER Questions you may be asked

Q: How do you debug a function that gives wrong output only for some inputs? Reproduce with the smallest failing input, inspect intermediate values (prints or debugger), form a hypothesis, fix, and add the case as a regression test.

Q: Unit test vs integration test? A unit test checks one small function in isolation; an integration test checks that components work together (e.g. load → transform → write).

Q: Write a regex for a 10-digit Indian mobile number. `^[6-9]\d{9}$` (optionally allowing a `+91` prefix: `^(?:\+91[- ])?[6-9]\d{9}$`).

NumPy: Fast Arrays, Vectorisation and Broadcasting

Learn the array library under pandas, scikit-learn and PyTorch. Create arrays, understand shape and dtype, index and slice in 1-D and 2-D, filter with boolean masks, compute with vectorised operations, aggregations along axes and broadcasting, generate random data and do basic linear algebra.

● Python class 11

✦ NumPy 2.x

Intermediate

🕒 120 min

By the end of this lesson you can

- Create arrays and describe their shape, ndim, dtype and size
- Index, slice and filter 1-D and 2-D arrays with boolean masks and fancy indexing
- Use vectorised arithmetic, universal functions and aggregations along an axis
- Apply broadcasting rules, reshape, stack and split arrays
- Generate reproducible random data and perform matrix operations

Why NumPy?

A Python list holds pointers to separate objects; a NumPy **array** stores numbers of one type in one contiguous block of memory and runs operations in compiled C. The result: code that is shorter **and** usually many times faster (the gap grows with heavier maths).

Python

```
import numpy as np, time
n = 1_000_000
py = list(range(n)); arr = np.arange(n)
t0 = time.perf_counter(); sq_list = [x * x for x in py]; t1 = time.perf_counter(); sq_arr = arr * arr;
t2 = time.perf_counter()
print(f"list comprehension {1000*(t1-t0):.1f} ms | numpy {1000*(t2-t1):.1f} ms | same result:
{sq_list[-1] == sq_arr[-1]}")
print(np.__version__)
```

OUTPUT

```
list comprehension 44.7 ms | numpy 34.8 ms | same result: True
2.5.3
```

Creating arrays

Python

```
a = np.array([3, 1, 4, 1, 5])
m = np.array([[1, 2, 3], [4, 5, 6]])
print(a, a.dtype, a.shape, a.ndim, a.size)
print(m, m.shape, m.ndim, m.dtype)
print(np.zeros(3), np.ones((2, 2)), np.full(3, 7), np.eye(2))
print(np.arange(0, 10, 2.5), np.linspace(0, 1, 5), np.array([1, 2.5]).dtype, np.array([1, 2],
dtype=np.float32).dtype)
```

OUTPUT

```
[3 1 4 1 5] int64 (5,) 1 5
[[1 2 3]
 [4 5 6]] (2, 3) 2 int64
[0. 0. 0.] [[1. 1.]
 [1. 1.]] [7 7 7] [[1. 0.]
 [0. 1.]]
[0. 2.5 5. 7.5] [0. 0.25 0.5 0.75 1. ] float64 float32
```

Array dimensions

1-D shape (5,)

A vector: [3 1 4 1 5]

2-D shape (2, 3)

A matrix: 2 rows × 3 columns. axis 0 = rows, axis 1 = columns

3-D shape (2, 2, 3)

A stack of matrices, e.g. an image batch or months × regions × products

Shapes of 1-D, 2-D and 3-D arrays. axis 0 runs down the rows, axis 1 across the columns.

All items share one **dtype** (`int64` , `float64` , `bool` , `<U5` strings...). Mixing ints and floats upcasts to float.

Indexing and slicing

Python

```
a = np.array([10, 20, 30, 40, 50, 60])
print(a[0], a[-1], a[1:4], a[::2], a[::-1])
m = np.arange(1, 13).reshape(3, 4)
print(m)
print(m[0, 2], m[1], m[:, 1], m[1:, 2:], m[-1, -1]) # [row, column]
```

OUTPUT

```
10 60 [20 30 40] [10 30 50] [60 50 40 30 20 10]
[[ 1  2  3  4]
 [ 5  6  7  8]]
```

```
[ 9 10 11 12]]
3 [5 6 7 8] [ 2  6 10] [[ 7  8]
[11 12]] 12
```

WATCH OUT Slices are views

`b = a[1:4]` does **not** copy; changing `b` changes `a`. Use `a[1:4].copy()` when you need an independent array. (Python lists behave the other way: slicing copies.)

Boolean masks and fancy indexing

Python

```
sales = np.array([120, 45, 300, 80, 210, 15])
mask = sales > 100
print(mask, sales[mask], sales[(sales > 50) & (sales < 250)], sales[~mask]) # & | ~ with brackets
print(sales[[0, 2, 4]], np.where(sales > 100, "high", "low"), np.where(sales > 100)[0])
sales[sales < 50] = 0 # conditional assignment
print(sales, np.count_nonzero(sales), (sales > 100).mean())
```

OUTPUT

```
[ True False  True False  True False] [120 300 210] [120  80 210] [45 80 15]
[120 300 210] ['high' 'low' 'high' 'low' 'high' 'low'] [0 2 4]
[120  0 300  80 210  0] 4 0.5
```

Vectorised maths and aggregations

Python

```
price = np.array([100.0, 250.0, 80.0]); qty = np.array([3, 1, 10])
print(price * qty, (price * qty).sum(), price * 1.18, np.round(np.sqrt(price), 2),
np.log(price).round(2))
m = np.array([[5, 2, 9], [1, 7, 3]])
print(m.sum(), m.sum(axis=0), m.sum(axis=1), m.max(axis=1), m.argmax(axis=1), m.mean(axis=0))
x = np.array([2, 4, 4, 4, 5, 5, 7, 9])
print(x.mean(), np.median(x), x.std(), x.std(ddof=1).round(3), np.percentile(x, [25, 75]),
np.cumsum(x), np.diff(x))
```

OUTPUT

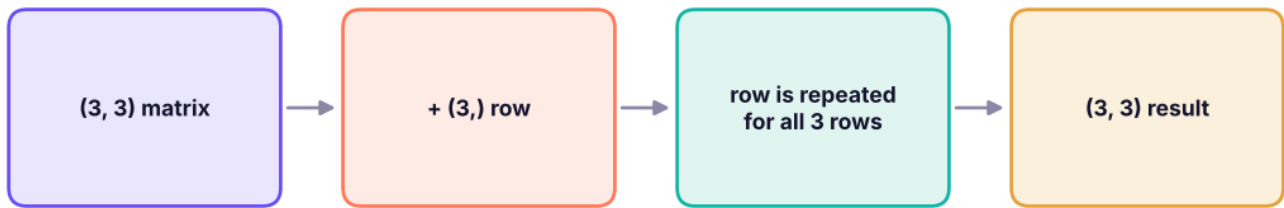
```
[300. 250. 800.] 1350.0 [118. 295. 94.4] [10. 15.81 8.94] [4.61 5.52 4.38]
27 [ 6  9 12] [16 11] [9 7] [2 1] [3. 4.5 6. ]
5.0 4.5 2.0 2.138 [4. 5.5] [ 2  6 10 14 19 24 31 40] [2 0 0 1 0 2 2]
```

`axis=0` collapses rows (one result per column); `axis=1` collapses columns (one result per row). `ddof=1` gives the **sample** standard deviation that pandas uses by default.

Broadcasting

When shapes differ, NumPy **stretches** the smaller array along dimensions of size 1, without copying data.

Broadcasting a row across a matrix



Broadcasting: a (3, 3) matrix plus a (3,) row is applied to every row.

Python

```
m = np.arange(1, 10).reshape(3, 3)
print(m + np.array([10, 20, 30]))           # add to each row
print(m * np.array([[1], [10], [100]]))    # multiply each row by its own factor (column vector)
scores = np.array([[70, 80, 90], [60, 75, 85], [88, 92, 79]], dtype=float)
z = (scores - scores.mean(axis=0)) / scores.std(axis=0) # standardise each column in one line
print(z.round(2))
```

OUTPUT

```
[[11 22 33]
 [14 25 36]
 [17 28 39]]
[[ 1  2  3]
 [40 50 60]
 [700 800 900]]
[[-0.23 -0.33  1.19]
 [-1.09 -1.03  0.07]
 [ 1.32  1.36 -1.26]]
```

Rule: compare shapes from the right; dimensions are compatible if equal or one of them is 1.

Reshaping, stacking, sorting

Python

```
a = np.arange(12)
print(a.reshape(3, 4).shape, a.reshape(2, -1).shape, a.reshape(3, 4).T.shape, a.reshape(3, 4).ravel()[:5])
x, y = np.array([1, 2]), np.array([3, 4])
print(np.vstack([x, y]), np.hstack([x, y]), np.column_stack([x, y]), np.concatenate([x, y]))
v = np.array([30, 10, 20, 10])
print(np.sort(v), np.argsort(v), np.unique(v, return_counts=True), v[np.argsort(-v)][:-2])
```

OUTPUT

```
(3, 4) (2, 6) (4, 3) [0 1 2 3 4]
[[1 2]
 [3 4]] [1 2 3 4] [[1 3]
 [2 4]] [1 2 3 4]
[10 10 20 30] [1 3 2 0] (array([10, 20, 30]), array([2, 1, 1])) [30 20]
```

Random numbers

Use the modern generator API with a seed so results are reproducible.

Python

```
rng = np.random.default_rng(42)
print(rng.integers(1, 7, size=10))           # dice
print(rng.normal(loc=50, scale=10, size=5).round(1)) # normal distribution
print(rng.choice(["A", "B"], size=8, p=[0.7, 0.3]), rng.uniform(0, 1, 3).round(3))
sample = rng.normal(50, 10, 100_000)
print(round(sample.mean(), 2), round(sample.std(), 2), round(((sample > 40) & (sample < 60)).mean(),
3)) # about 68% within 1 sd
```

OUTPUT

```
[1 5 4 3 3 6 1 5 2 1]
[37.  51.3 46.8 49.8 41.5]
['A' 'B' 'A' 'B' 'A' 'A' 'A' 'A'] [0.828 0.632 0.758]
49.96 10.04 0.682
```

Linear algebra basics

Machine-learning models are built from these operations.

Python

```
A = np.array([[2, 1], [1, 3]]); b = np.array([5, 10])
print(A @ np.array([1, 2]), A.T, np.linalg.det(A).round(2), np.linalg.inv(A).round(3))
print(np.linalg.solve(A, b)) # solves 2x + y = 5, x + 3y = 10
w = np.array([0.5, -0.2]); X = np.array([[1.0, 2.0], [3.0, 1.0], [2.0, 2.0]])
print(X @ w) # predictions of a linear model y = Xw
```

OUTPUT

```
[4 7] [[2 1]
 [1 3]] 5.0 [[ 0.6 -0.2]
 [-0.2  0.4]]
[1.  3.]
[0.1 1.3 0.6]
```

Real data: house prices

Python

```
import pandas as pd
h = pd.read_csv("house_prices.csv")
area, price = h["area_sqft"].to_numpy(), h["price_lakh"].to_numpy()
print("missing values:", np.isnan(area).sum(), "area,", np.isnan(price).sum(), "price")      # real data
has gaps
ok = ~np.isnan(area) & ~np.isnan(price)
area, price = area[ok], price[ok]                                                         # keep
complete rows
per_sqft = price * 1e5 / area
print(f"{len(price)} houses | median price ₹{np.median(price):.1f} lakh | median ₹
{np.median(per_sqft):.0f}/sq ft")
print("price percentiles 10/50/90:", np.percentile(price, [10, 50, 90]).round(1))
big = area > 1500
print(f"{big.mean():.0%} are over 1,500 sq ft; their median price is ₹{np.median(price[big]):.1f} lakh
vs ₹{np.median(price[~big]):.1f} lakh")
r = np.corrcoef(area, price)[0, 1]
slope, intercept = np.polyfit(area, price, 1)
print(f"correlation area-price r = {r:.2f}; fitted line: price ≈ {intercept:.1f} + {slope*100:.2f} lakh
per 100 sq ft")
```

OUTPUT

```
missing values: 25 area, 0 price
1175 houses | median price ₹78.7 lakh | median ₹6,258/sq ft
price percentiles 10/50/90: [ 40.3  78.7 119.9]
27% are over 1,500 sq ft; their median price is ₹110.0 lakh vs ₹68.1 lakh
correlation area-price r = 0.79; fitted line: price ≈ -7.5 + 7.05 lakh per 100 sq ft
```

`np.polyfit` fits a straight line by least squares, which is the same idea as linear regression (ML track).

Summary

KEY TAKEAWAYS

- Arrays hold one dtype in contiguous memory; operations are vectorised and much faster than Python loops.
- Know `shape`, `ndim`, `dtype`, `size`; index with `[row, col]`; slices are views (use `.copy()`).
- Boolean masks (`a[a > 0]`, `&`, `|`, `~`) and `np.where` filter and recode.
- Aggregate with `axis=0` (per column) or `axis=1` (per row); `ddof=1` for sample standard deviation.
- Broadcasting stretches size-1 dimensions; reshape with `-1` to infer a dimension.
- Use `np.random.default_rng(seed)` and `@` for matrix multiplication.

Quick quiz

6 questions · tap an answer to check it

1. What is `np.arange(12).reshape(3, -1).shape` ?

A (3, 3)

B (3, 4)

C (4, 3)

D Error

Answer: B. -1 lets NumPy infer the missing dimension: $12 / 3 = 4$.

2. What does `m.sum(axis=0)` return for a 2×3 matrix?

A One number

B Three numbers: the sum of each column

C Two numbers: the sum of each row

D A 2×3 matrix

Answer: B. `axis=0` collapses the rows.

3. Which expression keeps values between 10 and 20?

A `a[a > 10 and a < 20]`

B `a[(a > 10) & (a < 20)]`

C `a[10:20]`

D `a.between(10, 20)`

Answer: B. Use `&` with brackets for element-wise conditions.

4. After `b = a[0:3]; b[0] = 99` , what happens to `a[0]` ?

- A Unchanged
- B It becomes 99 because a slice is a view
- C Error
- D It becomes 0

Answer: B. Use `.copy()` to avoid this.

5. Can a (3, 4) array be added to a (4,) array?

- A Yes, the (4,) array is broadcast across the rows
- B No, shapes must match exactly
- C Only with a loop
- D Only if both are float

Answer: A. Trailing dimensions match (4 and 4).

6. How do you make random results reproducible?

- A Use `np.random.random()`
- B Create a generator with a seed: `np.random.default_rng(42)`
- C Run the cell twice
- D Use floats

Answer: B. The same seed gives the same numbers.

EXERCISE NumPy practice

(1) Create a 5×5 array of 1..25 and select the diagonal and the last column. (2) Replace all negative values in an array with 0. (3) Standardise the `area_sqft` column (z-scores) and count values beyond ± 2 . (4) Simulate 10,000 rolls of two dice and estimate $P(\text{sum} = 7)$. (5) Solve $3x + 2y = 12$, $x - y = 1$.

Show solution (worked answer)

Python

```
g = np.arange(1, 26).reshape(5, 5); print(np.diag(g), g[:, -1])
v = np.array([3, -1, 4, -5]); v[v < 0] = 0; print(v)
z = (area - area.mean()) / area.std(); print((np.abs(z) > 2).sum())
rng = np.random.default_rng(0); rolls = rng.integers(1, 7, (10_000, 2)).sum(axis=1); print((rolls
= 7).mean()) # about 1/6
print(np.linalg.solve([[3, 2], [1, -1]], [12, 1])) # x = 2.8, y = 1.8
```

INTERVIEW CORNER Questions you may be asked

Q: Why is NumPy faster than Python lists? Contiguous memory of one dtype, operations implemented in C over the whole array (vectorisation), and no per-element Python overhead.

Q: What is broadcasting? NumPy's rule for operating on arrays of different shapes by virtually repeating dimensions of size 1, aligning shapes from the right.

Q: View vs copy? Basic slices return views that share memory; fancy/boolean indexing and `.copy()` return copies.

Q: Difference between `np.std(x)` and `pandas.Series.std()`? NumPy defaults to population std (`ddof=0`); pandas defaults to sample std (`ddof=1`).

pandas I: Series, DataFrames, Reading Data, Selecting and Filtering

Meet the most-used data library. Create Series and DataFrames, read CSV and Excel files, inspect data with head, info and describe, select columns and rows with loc and iloc, filter with conditions and query, sort, add columns and save results, using the Superstore sales data.

● Python class 12

✦ pandas 2.3 / 3.0

Intermediate

🕒 120 min

By the end of this lesson you can

- Create Series and DataFrames and explain the index
- Read CSV and Excel files with the right options and write results back out
- Inspect a dataset with shape, dtypes, head, info, describe and value_counts
- Select with [], loc and iloc and filter rows with boolean conditions, isin, between and query
- Sort, rank, add and rename columns, and avoid the SettingWithCopy pitfall

Series and DataFrame

pandas gives Python spreadsheet-like tables. A **Series** is one labelled column; a **DataFrame** is a table of Series sharing one **index** (row labels).

Anatomy of a DataFrame

Index

Row labels: 0, 1, 2 ... or
order_id, dates

Columns

Each column is a Series with
one dtype

Values

The underlying NumPy / Arrow
arrays

A DataFrame is a set of Series (columns) that share an index (row labels).

Python

```
import pandas as pd, numpy as np
pd.set_option("display.max_columns", 20); pd.set_option("display.width", 160) # show wide tables in full
s = pd.Series([120, 95, 140], index=["North", "South", "East"], name="sales")
print(s, s["South"], s.max(), s.idxmax(), sep="\n")
df = pd.DataFrame({"city": ["Pune", "Delhi", "Kochi"], "sales": [120, 95, 140], "target": [100, 110, 130]})
df["hit"] = df["sales"] >= df["target"]
print(df)
print(df.shape, list(df.columns), df.index, df.dtypes.to_dict(), pd.__version__)
```

OUTPUT

```
North    120
South     95
East     140
Name: sales, dtype: int64
95
140
East
   city  sales  target  hit
0  Pune    120     100  True
1  Delhi     95     110 False
2  Kochi    140     130  True
(3, 4) ['city', 'sales', 'target', 'hit'] RangeIndex(start=0, stop=3, step=1) {'city':
dtype('O'), 'sales': dtype('int64'), 'target': dtype('int64'), 'hit': dtype('bool')} 2.3.3
```

Reading data

Python

```
s = pd.read_csv("superstore_sales.csv", parse_dates=["order_date", "ship_date"])
print(s.shape)
print(s.head(3).T)
```

OUTPUT

```
(6323, 19)
```

	0	1	2
row_id	1	2	3
order_id	IN-2023-00001	IN-2023-00001	IN-2023-00001
order_date	2023-01-01 00:00:00	2023-01-01 00:00:00	2023-01-01 00:00:00
ship_date	2023-01-05 00:00:00	2023-01-05 00:00:00	2023-01-05 00:00:00
ship_mode	Second Class	Second Class	Second Class
customer_id	C0765	C0765	C0765
customer_name	Ishaan Gupta	Ishaan Gupta	Ishaan Gupta
segment	Consumer	Consumer	Consumer
city	Bengaluru	Bengaluru	Bengaluru
state	Karnataka	Karnataka	Karnataka
region	South	South	South

product_id	P0059	P0050	P0011
category	Technology	Office Supplies	Furniture
sub_category	Phones	Appliances	Tables
product_name	Nova Phone 223	Orbit Appliance 798	Orbit Table 248
sales	36000.0	3850.0	62080.0
quantity	2	1	4
discount	0.0	0.0	0.0
profit	3472.55	1169.54	6044.47

Option	Use
<code>sep=";"</code>	Other delimiters (<code>"\t"</code> for TSV)
<code>usecols=[...]</code>	Read only some columns (faster, less memory)
<code>dtype={"pin": str}</code>	Keep leading zeros and IDs as text
<code>parse_dates=[...]</code>	Convert date columns
<code>na_values=["NA", "-"]</code>	Extra markers for missing
<code>nrows=1000 , chunksize=100_000</code>	Sample or stream big files
<code>encoding="latin-1"</code>	Fix decode errors in older files

Other readers: `pd.read_excel("file.xlsx", sheet_name="Sales")` (needs `openpyxl`), `pd.read_json` , `pd.read_parquet` , `pd.read_sql` (Lesson 23), `pd.read_html` , `pd.read_clipboard` .

Inspecting data

Run these five on every new dataset:

Python

```
print(s.info(memory_usage="deep"))
```

OUTPUT

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6323 entries, 0 to 6322
Data columns (total 19 columns):
#   Column      Non-Null Count  Dtype
---  -
0   row_id      6323 non-null  int64
1   order_id    6323 non-null  object
2   order_date  6323 non-null  datetime64[ns]
3   ship_date   6323 non-null  datetime64[ns]
4   ship_mode   6323 non-null  object
5   customer_id 6323 non-null  object
6   customer_name 6323 non-null  object
7   segment     6323 non-null  object
8   city        6323 non-null  object
9   state       6323 non-null  object
```

```

10 region          6323 non-null object
11 product_id      6323 non-null object
12 category        6323 non-null object
13 sub_category    6323 non-null object
14 product_name    6323 non-null object
15 sales           6323 non-null float64
16 quantity        6323 non-null int64
17 discount        6323 non-null float64
18 profit          6323 non-null float64
dtypes: datetime64[ns](2), float64(3), int64(2), object(12)
memory usage: 4.6 MB
None

```

Python

```

print(s.describe(include="number").round(1).T)
print(s["region"].value_counts(), s["category"].value_counts(normalize=True).round(3), sep="\n")
print(s["customer_id"].nunique(), "customers |", s["order_date"].min().date(), "to",
s["order_date"].max().date())
print(s.isna().sum().sum(), "missing cells |", s.duplicated().sum(), "duplicate rows")

```

OUTPUT

	count	mean	std	min	25%	50%	75%	max
row_id	6323.0	3162.0	1825.4	1.0	1581.5	3162.0	4742.5	6323.0
sales	6323.0	31854.1	60414.2	42.0	1564.0	8430.0	32592.0	619290.0
quantity	6323.0	2.6	1.6	1.0	1.0	2.0	3.0	7.0
discount	6323.0	0.1	0.1	0.0	0.0	0.0	0.2	0.4
profit	6323.0	2902.4	11002.0	-75873.8	-17.9	266.3	2270.1	149424.9

region

South	1656
West	1614
North	1530
East	1523

Name: count, dtype: int64

category

Office Supplies	0.398
Furniture	0.303
Technology	0.299

Name: proportion, dtype: float64

785 customers | 2023-01-01 to 2025-12-31

0 missing cells | 0 duplicate rows

Selecting columns and rows

Python

```
print(s["sales"].head(3).tolist())
print(s[["order_id", "region", "sales"]].head(3))
print(s.loc[0, "customer_name"], s.iloc[0, 6])
print(s.loc[0:2, "region": "product_id"])
print(s.iloc[0:2, 10:12])
```

one column → Series
list of columns → DataFrame
loc = labels, iloc = positions
loc slices INCLUDE the end label
iloc slices EXCLUDE the end, like Python

OUTPUT

[36000.0, 3850.0, 62080.0]

order_id region sales

0 IN-2023-00001 South 36000.0

1 IN-2023-00001 South 3850.0

2 IN-2023-00001 South 62080.0

Ishaan Gupta Ishaan Gupta

region product_id

0 South P0059

1 South P0050

2 South P0011

region product_id

0 South P0059

1 South P0050

Tool	Selects by	Example
<code>df["col"]</code> / <code>df[["a", "b"]]</code>	Column names	<code>s["sales"]</code>
<code>df.loc[rows, cols]</code>	Labels and boolean masks	<code>s.loc[s.profit < 0, ["order_id", "profit"]]</code>
<code>df.iloc[rows, cols]</code>	Integer positions	<code>s.iloc[:5, :3]</code>
<code>df.at[r, c]</code> / <code>df.iat[i, j]</code>	One value, fast	<code>s.at[0, "sales"]</code>

Filtering rows

Python

```
loss = s[s["profit"] < 0]
print(len(loss), "loss-making lines")
tech_south = s[(s["category"] == "Technology") & (s["region"] == "South")]
print(len(tech_south), round(tech_south["sales"].sum() / 1e6, 2), "M")
print(s[s["city"].isin(["Pune", "Kochi"])]["city"].value_counts().to_dict())
print(len(s[s["discount"].between(0.1, 0.2)]), len(s[s["product_name"].str.contains("Chair")]))
print(s.query("region = 'West' and sales > 50000 and discount = 0")[["order_id", "sales"]].head(3))
big = 50000
print(len(s.query("sales > @big")))
```

& | ~ need brackets
@ refers to a Python variable

OUTPUT


```
1682 loss-making lines
495 41.89 M
{'Pune': 529, 'Kochi': 417}
1943 493

      order_id    sales
7   IN-2023-00003  69280.0
22  IN-2023-00013  72000.0
98  IN-2023-00051  64450.0
1126
```

Sorting and ranking

Python

```
top = s.sort_values("sales", ascending=False)[["order_id", "product_name", "sales"]].head(5)
print(top)
print(s.sort_values(["region", "profit"], ascending=[True, False]).groupby("region").head(1)[["region",
"product_name", "profit"]])
print(s.nlargest(3, "profit")[["product_name", "profit"]], s.nsmallest(2, "profit")[["product_name",
"profit"]], sep="\n")
```

OUTPUT

```
      order_id    product_name    sales
3669 IN-2024-01857  Orbit Copier 746  619290.0
2087 IN-2024-01040  Vertex Copier 262  551530.0
1142 IN-2023-00564  Vertex Copier 262  551530.0
5696 IN-2025-02880  Orbit Copier 746  530820.0
1653 IN-2023-00809  Orbit Copier 746  526396.5

      region    product_name    profit
4219   East  Orbit Copier 746  135564.21
3669  North  Orbit Copier 746  149424.89
3893  South   Apex Copier 347  107757.96
481   West  Zenith Copier 719   95485.58

      product_name    profit
3669  Orbit Copier 746  149424.89
4219  Orbit Copier 746  135564.21
3893   Apex Copier 347  107757.96

      product_name    profit
5979  Orbit Copier 746 -75873.84
639   Pioneer Copier 366 -67392.75
```

Adding, changing and removing columns

Python

```
e = pd.read_csv("employees_small.csv", parse_dates=["join_date"])
e["bonus"] = e["salary"] * 0.1 # vectorised arithmetic
e["band"] = np.where(e["salary"] >= 100000, "Senior", "Standard") # IF
e["grade"] = pd.cut(e["salary"], bins=[0, 70000, 100000, np.inf], labels=["C", "B", "A"]) # bins
e["tenure_yrs"] = ((pd.Timestamp("2025-10-27") - e["join_date"]).dt.days / 365.25).round(1)
e = e.assign(name_upper=e["name"].str.upper()).rename(columns={"rating": "perf"}).drop(columns=["name_upper"])
e["salary_rank"] = e["salary"].rank(ascending=False, method="dense").astype(int)
print(e.head(6))
```

OUTPUT

	emp_id	name	department	city	join_date	salary	perf	bonus	band
	grade	tenure_yrs	salary_rank						
0	E101	Kavya Iyer	HR	Mumbai	2016-02-28	97000	4	9700.0	Standard
B		9.7	8						
1	E102	Imran Reddy	Finance	Delhi	2019-12-14	157000	3	15700.0	Senior
A		5.9	1						
2	E103	Harpreet Kulkarni	Finance	Mumbai	2024-10-05	131000	4	13100.0	Senior
A		1.1	4						
3	E104	Diya Iyer	HR	Bengaluru	2024-01-02	90000	4	9000.0	Standard
B		1.8	9						
4	E105	Simran Iyer	HR	Bengaluru	2017-08-03	139000	4	13900.0	Senior
A		8.2	3						
5	E106	Saanvi Rao	HR	Delhi	2025-02-18	83000	4	8300.0	Standard
B		0.7	10						

WATCH OUT SettingWithCopyWarning

`sub = df[df.x > 0]; sub["y"] = 1` may warn because `sub` might be a view. Use `df.loc[df.x > 0, "y"] = 1` to change the original, or `sub = df[df.x > 0].copy()` for an independent table. pandas 3 makes copy-on-write the default, which removes the ambiguity.

Saving results

Python

```
out = s.groupby("region", as_index=False)["sales"].sum()
out.to_csv("region_sales.csv", index=False)
print(open("region_sales.csv").read())
import os; os.remove("region_sales.csv")
```

OUTPUT

```
region,sales
East,46484119.0
North,49018161.0
```

South,53952768.5
West,51958520.5

Also: `to_excel("out.xlsx", sheet_name="Summary", index=False)` , `to_parquet` (fast, typed, compressed), `to_json(orient="records")` , `to_sql` .

pandas vs Excel vs SQL

Task	Excel	SQL	pandas
Filter	AutoFilter	<code>WHERE</code>	<code>df[df.x > 5]</code> / <code>query</code>
New column	Formula	<code>SELECT a*b AS c</code>	<code>df["c"] = df.a * df.b</code>
Sort	Sort	<code>ORDER BY</code>	<code>sort_values</code>
Lookup	XLOOKUP	<code>JOIN</code>	<code>merge</code> (Lesson 18)
Pivot	PivotTable	<code>GROUP BY</code>	<code>groupby</code> , <code>pivot_table</code> (Lesson 18)

Summary

KEY TAKEAWAYS

- A DataFrame is a table of Series sharing an index; read files with `read_csv` / `read_excel` and the right options.
- Inspect first: `shape` , `info()` , `describe()` , `value_counts()` , `isna().sum()` , `duplicated().sum()` .
- Select with `[]` (columns), `loc` (labels, end-inclusive) and `iloc` (positions, end-exclusive).
- Filter with boolean masks using `&` , `|` , `~` and brackets, plus `isin` , `between` , `str.contains` , `query` .
- Add columns with vectorised maths, `np.where` , `pd.cut` ; use `.loc` or `.copy()` to avoid SettingWithCopy issues.

Quick quiz

6 questions · tap an answer to check it

1. What does `s[["region", "sales"]]` return?

A A Series

B A DataFrame with two columns

C A list

D An error

Answer: B. Double brackets pass a list of columns.

2. Which is end-inclusive?

A `df.loc[0:2]`

B `df.iloc[0:2]`

C Both

D Neither

Answer: A. `loc` uses labels and includes the end label.

3. Which filter is correct?

A `df[df.a > 1 and df.b < 5]`

B `df[(df.a > 1) & (df.b < 5)]`

C `df[df.a > 1 & df.b < 5]`

D `df.where(a > 1, b < 5)`

Answer: B. Use `&` with brackets around each condition.

4. How do you keep leading zeros in a PIN or ID column when reading?

A `parse_dates`

B `dtype={"pin": str}`

C `usecols`

D `index_col`

Answer: B. Otherwise pandas reads it as a number.

5. Which command shows column types and non-null counts?

A `describe()`

B `info()`

C `head()`

D `shape`

Answer: B. `describe()` gives statistics.

6. How do you reference a Python variable inside `query` ?

A `{var}`

B `@var`

C `$var`

D `var()`

Answer: B. `df.query("sales > @big")` .

EXERCISE pandas practice (Superstore)

(1) How many orders (unique `order_id`) are there? (2) Show the 5 most profitable sub-categories by total profit. (3) Filter Consumer-segment orders in 2025 with discount above 20%. (4) Add `margin_pct` and list the 3 lowest-margin product names among lines with sales above 20,000. (5) Save the Technology rows to `tech.csv`.

Show solution (worked answer)

Python

```
print(s["order_id"].nunique())
print(s.groupby("sub_category")["profit"].sum().nlargest(5))
f = s[(s.segment == "Consumer") & (s.order_date.dt.year == 2025) & (s.discount > 0.2)];
print(len(f))
s["margin_pct"] = 100 * s.profit / s.sales
print(s[s.sales > 20000].nsmallest(3, "margin_pct")[["product_name", "margin_pct"]])
s[s.category == "Technology"].to_csv("tech.csv", index=False)
```

INTERVIEW CORNER Questions you may be asked

Q: loc vs iloc? `loc` selects by label (and boolean masks), end-inclusive; `iloc` selects by integer position, end-exclusive.

Q: How do you find and handle duplicates? `df.duplicated(subset=[...]).sum()` to count, `drop_duplicates(subset=..., keep="first")` to remove, after deciding which columns define a duplicate.

Q: How do you read a file that does not fit in memory? `usecols` and `dtype` to shrink it, `chunksize` to stream it, or use DuckDB/Polars/Parquet.

pandas II: Data Cleaning: Missing Values, Types, Duplicates, Text and Outliers

Turn messy data into analysis-ready data. Detect and treat missing values, fix data types, remove duplicates, standardise text, parse dates, handle outliers with IQR and caps, and document every step in a reusable cleaning function.

● Python class 13

✦ pandas 2.3 / 3.0

Intermediate

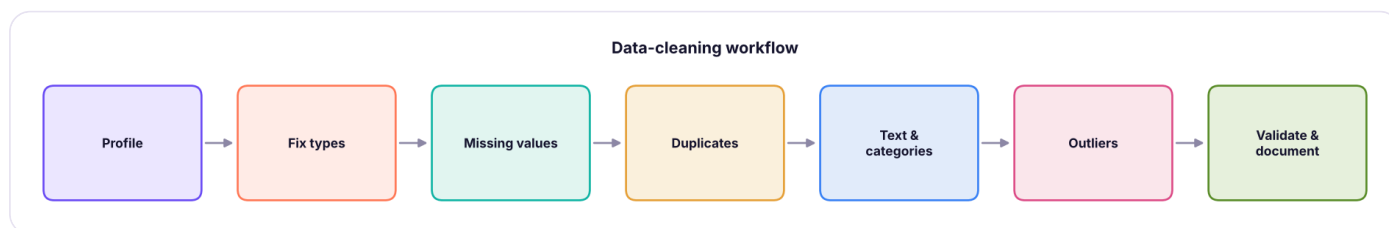
🕒 120 min

By the end of this lesson you can

- Profile a dataset for missing values, wrong types, duplicates and inconsistent categories
- Choose and apply missing-value strategies (drop, fill with statistic, group-wise fill, flag)
- Convert types safely with `to_numeric`, `to_datetime` and `astype`, including categories
- Standardise text columns and map inconsistent labels
- Detect outliers with the IQR rule and z-scores and decide how to treat them

Why cleaning matters

Analysts often say 60–80% of project time goes into preparing data. Wrong types, gaps, duplicates and typos silently break totals, joins and models: **garbage in, garbage out**.



A repeatable cleaning workflow.

We will clean a deliberately messy extract:

Python

```
import pandas as pd, numpy as np
pd.set_option("display.max_columns", 20); pd.set_option("display.width", 160)
raw = pd.DataFrame({
    "order_id": ["A1", "A2", "A3", "A3", "A4", "A5", "A6", "A7", "A8"],
    "city":      ["pune", "Pune ", "DELHI", "DELHI", "New Delhi", "kochi", None, "Bengaluru",
"Bangalore"],
    "amount":    ["1,200", "950", "₹3,400", "₹3,400", "abc", "780", "15000", "2,050", None],
    "order_date":["2025-01-05", "05/01/2025", "2025-01-07", "2025-01-07", "2025-02-30", "2025-01-09",
"2025-01-10", "2025-01-11", "2025-01-12"],
    "qty":       [2, 1, 3, 3, None, 1, 50, 2, 1],
})
print(raw)
print(raw.dtypes.to_dict())
```

OUTPUT

	order_id	city	amount	order_date	qty
0	A1	pune	1,200	2025-01-05	2.0
1	A2	Pune	950	05/01/2025	1.0
2	A3	DELHI	₹3,400	2025-01-07	3.0
3	A3	DELHI	₹3,400	2025-01-07	3.0
4	A4	New Delhi	abc	2025-02-30	NaN
5	A5	kochi	780	2025-01-09	1.0
6	A6	NaN	15000	2025-01-10	50.0
7	A7	Bengaluru	2,050	2025-01-11	2.0
8	A8	Bangalore	NaN	2025-01-12	1.0

```
{'order_id': <StringDtype(na_value=nan)>, 'city': <StringDtype(na_value=nan)>, 'amount':
<StringDtype(na_value=nan)>, 'order_date': <StringDtype(na_value=nan)>, 'qty': dtype('float64')}
```

Step 1: Profile

Python

```
def profile(df):
    return pd.DataFrame({"dtype": df.dtypes.astype(str), "missing": df.isna().sum(), "missing_%": (100
* df.isna().mean()).round(1),
                        "unique": df.nunique(), "example": df.iloc[0]})
print(profile(raw))
print("duplicate rows:", raw.duplicated().sum())
```

OUTPUT

	dtype	missing	missing_%	unique	example
order_id	str	0	0.0	8	A1
city	str	1	11.1	7	pune
amount	str	1	11.1	7	1,200
order_date	str	0	0.0	8	2025-01-05
qty	float64	1	11.1	4	2.0

```
duplicate rows: 1
```


Step 2: Fix types

Python

```
df = raw.drop_duplicates().copy() # exact duplicates first
df["amount"] = pd.to_numeric(df["amount"].str.replace(r"[₹,]", "", regex=True), errors="coerce") #
# bad values → NaN
df["order_date"] = pd.to_datetime(df["order_date"], format="%Y-%m-%d", errors="coerce") #
# strict format: others → NaT
df["qty"] = df["qty"].astype("Int64") # nullable integer keeps
# NaN
print(df[["order_id", "amount", "order_date", "qty"]])
print(df.dtypes.to_dict())
```

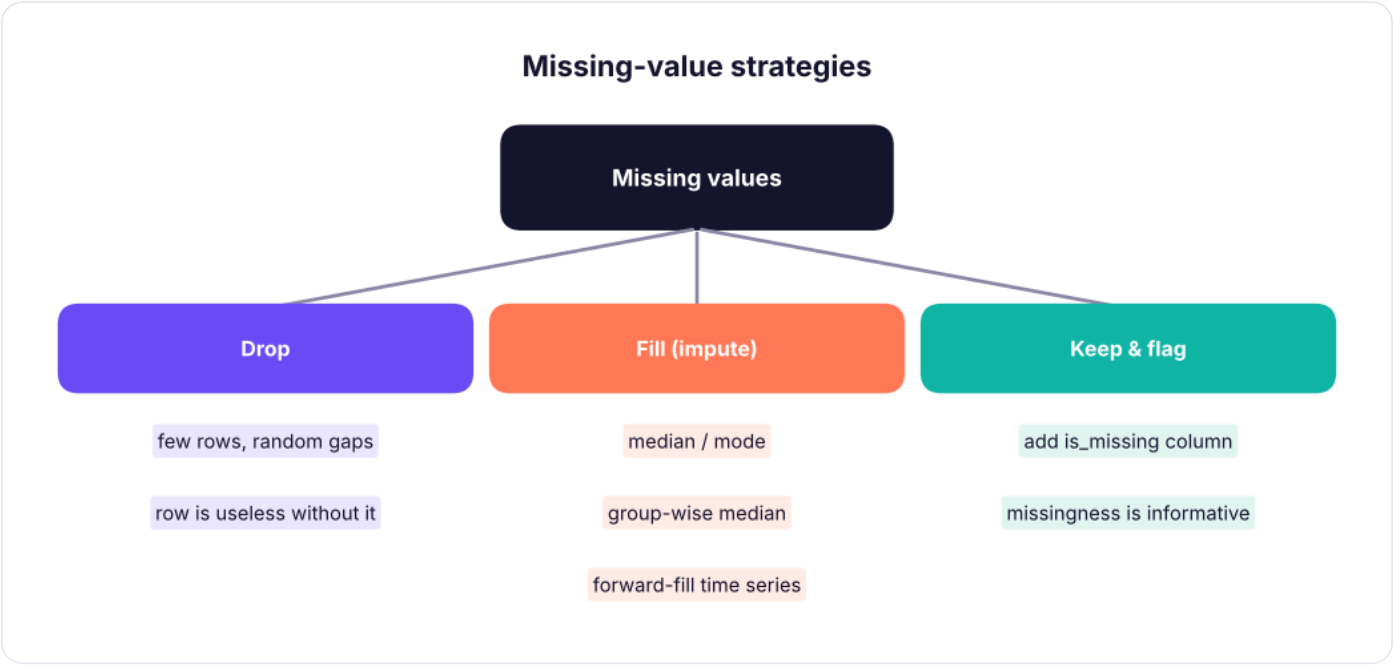
OUTPUT

```
order_id  amount  order_date  qty
0         A1   1200.0 2025-01-05    2
1         A2    950.0         NaT    1
2         A3   3400.0 2025-01-07    3
4         A4      NaN         NaT  <NA>
5         A5    780.0 2025-01-09    1
6         A6  15000.0 2025-01-10   50
7         A7   2050.0 2025-01-11    2
8         A8      NaN 2025-01-12    1
{'order_id': <StringDtype(na_value=nan)>, 'city': <StringDtype(na_value=nan)>, 'amount':
dtype('float64'), 'order_date': dtype('<M8[us]>'), 'qty': Int64Dtype()}
```

`errors="coerce"` turns unparseable values into `NaN` / `NaT` so you can count and inspect them instead of crashing. Always check how many values were coerced.

Conversion	Use
<code>pd.to_numeric(s, errors="coerce")</code>	Text with numbers
<code>pd.to_datetime(s, format=..., dayfirst=True)</code>	Dates (give the format when known)
<code>s.astype("category")</code>	Repeated labels: less memory, ordered categories
<code>s.astype("Int64")</code> , <code>"boolean"</code> , <code>"string"</code>	Nullable types that allow missing values

Step 3: Missing values



Choosing a strategy for missing values.

Python

```
print(df.isna().sum().to_dict())
df["amount_missing"] = df["amount"].isna() # flag before filling
df["amount"] = df["amount"].fillna(df["amount"].median())
df["qty"] = df["qty"].fillna(1)
df = df.dropna(subset=["order_date"]) # a row without a valid date
# is unusable here
print(df[["order_id", "amount", "amount_missing", "qty", "order_date"]])
```

OUTPUT

{'order_id': 0, 'city': 1, 'amount': 2, 'order_date': 2, 'qty': 1}

	order_id	amount	amount_missing	qty	order_date
0	A1	1200.0	False	2	2025-01-05
2	A3	3400.0	False	3	2025-01-07
5	A5	780.0	False	1	2025-01-09
6	A6	15000.0	False	50	2025-01-10
7	A7	2050.0	False	2	2025-01-11
8	A8	1625.0	True	1	2025-01-12

Real data: `house_prices.csv` has gaps in `area_sqft` . A **group-wise** fill uses similar houses (same number of bedrooms) instead of one global value:

Python

```
h = pd.read_csv("house_prices.csv")
print("missing area:", h["area_sqft"].isna().sum())
by_bed = h.groupby("bedrooms")["area_sqft"].median().round(0)
print(by_bed.to_dict())
h["area_filled"] = h["area_sqft"].fillna(h.groupby("bedrooms")["area_sqft"].transform("median"))
print("after group-wise fill:", h["area_filled"].isna().sum(), "| global median would give",
h["area_sqft"].median())
```

OUTPUT

```
missing area: 25
{1: 760.0, 2: 1115.0, 3: 1430.0, 4: 1715.0, 5: 1955.0}
after group-wise fill: 0 | global median would give 1240.0
```

NOTE Missing-data types

MCAR (completely at random) → dropping is safe. **MAR** (depends on other columns) → impute using those columns. **MNAR** (depends on the missing value itself, e.g. high earners hide income) → flag it; imputation can bias results. Never fill target variables for modelling.

Step 4: Duplicates

Python

```
orders = pd.DataFrame({"order_id": ["B1", "B2", "B2", "B3"], "status": ["new", "new", "shipped",
"new"], "updated": pd.to_datetime(["2025-01-01", "2025-01-01", "2025-01-03", "2025-01-02"])})
print(orders.duplicated().sum(), orders.duplicated(subset="order_id").sum())
latest = orders.sort_values("updated").drop_duplicates(subset="order_id", keep="last") # keep the
latest version per key
print(latest)
```

OUTPUT

```
0 1
  order_id  status  updated
0      B1      new 2025-01-01
3      B3      new 2025-01-02
2      B2  shipped 2025-01-03
```

Decide which columns define "the same record"; exact duplicates and key duplicates are different problems.

Step 5: Text and categories

Python

```
df["city"] = df["city"].str.strip().str.title()
fix = {"New Delhi": "Delhi", "Bangalore": "Bengaluru"}
df["city"] = df["city"].replace(fix).fillna("Unknown")
print(df["city"].value_counts().to_dict())
print(pd.Series([" Data Science ", "MACHINE learning"]).str.split().str.join(" ").str.capitalize().tolist())
```

OUTPUT

```
{'Bengaluru': 2, 'Pune': 1, 'Delhi': 1, 'Kochi': 1, 'Unknown': 1}
['Data science', 'Machine learning']
```

Useful `.str` methods: `strip`, `lower/upper/title`, `replace`, `contains`, `startswith`, `extract`, `split`, `len`, `zfill`, `slice`. For fuzzy spellings, libraries such as **rapidfuzz** score string similarity.

Step 6: Outliers

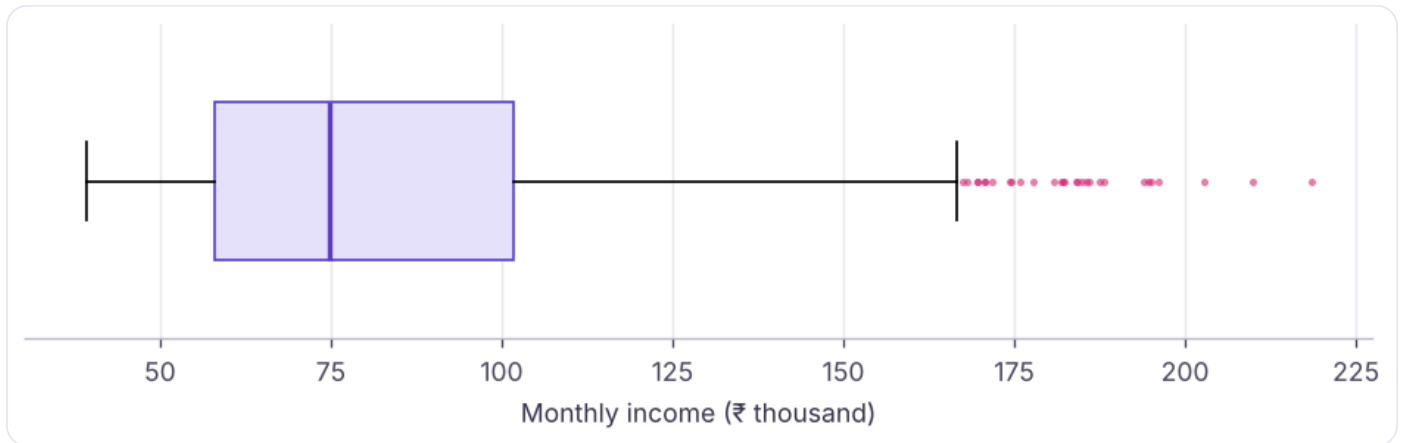
An **outlier** is far from most values. It may be an error (qty = 50 typed instead of 5) or real (one bulk order). Detect first, then decide.

Python

```
hr = pd.read_csv("hr_employees.csv")
inc = hr["monthly_income"]
q1, q3 = inc.quantile([0.25, 0.75]); iqr = q3 - q1
lo, hi = q1 - 1.5 * iqr, q3 + 1.5 * iqr
print(f"Q1 {q1:,.0f} | Q3 {q3:,.0f} | IQR fences {lo:,.0f} to {hi:,.0f}")
out = hr[(inc < lo) | (inc > hi)]
print(len(out), "IQR outliers;", out["job_title"].value_counts().head(3).to_dict())
z = (inc - inc.mean()) / inc.std()
print((z.abs() > 3).sum(), "rows with |z| > 3")
capped = inc.clip(lower=inc.quantile(0.01), upper=inc.quantile(0.99)) # winsorise at 1st/99th percentile
print(f"max before {inc.max():,.0f} after capping {capped.max():,.0f}")
```

OUTPUT

```
Q1 58,062 | Q3 101,833 | IQR fences -7,594 to 167,489
29 IQR outliers; {'Sales Manager': 6, 'Finance Manager': 5, 'Operations Manager': 5}
19 rows with |z| > 3
max before 218,417 after capping 182,086
```



Monthly income: the box plot marks values beyond the IQR fences as points.

Senior roles earn more, so high incomes here are **real**, not errors. Options: keep them, analyse by job level, use medians, log-transform, or cap for models that are sensitive to extremes. Delete only values that are clearly impossible (negative age, qty 50 in a 1–5 item shop).

A reusable cleaning function

Python

```
def clean_orders(raw: pd.DataFrame) → pd.DataFrame:
    """Return cleaned orders and print a short audit log."""
    df = raw.drop_duplicates().copy(); log = {"rows_in": len(raw), "dups_removed": len(raw) - len(df)}
    df["amount"] = pd.to_numeric(df["amount"].str.replace(r"₹,", "", regex=True), errors="coerce")
    df["order_date"] = pd.to_datetime(df["order_date"], format="%Y-%m-%d", errors="coerce")
    log["bad_dates_dropped"] = int(df["order_date"].isna().sum()); df = df.dropna(subset=
["order_date"])
    log["amount_imputed"] = int(df["amount"].isna().sum()); df["amount"] =
df["amount"].fillna(df["amount"].median())
    df["city"] = df["city"].str.strip().str.title().replace({"New Delhi": "Delhi", "Bangalore":
"Bengaluru"}).fillna("Unknown")
    df["qty"] = df["qty"].fillna(1).clip(upper=10).astype(int); log["rows_out"] = len(df)
    print(log); return df
clean = clean_orders(raw)
assert clean["amount"].notna().all() and clean["order_id"].is_unique
print(clean)
```

OUTPUT

```
{'rows_in': 9, 'dups_removed': 1, 'bad_dates_dropped': 2, 'amount_imputed': 1, 'rows_out': 6}
```

	order_id	city	amount	order_date	qty
0	A1	Pune	1200.0	2025-01-05	2
2	A3	Delhi	3400.0	2025-01-07	3
5	A5	Kochi	780.0	2025-01-09	1
6	A6	Unknown	15000.0	2025-01-10	10
7	A7	Bengaluru	2050.0	2025-01-11	2
8	A8	Bengaluru	2050.0	2025-01-12	1

Summary

KEY TAKEAWAYS

- Profile first: dtypes, missing counts, uniques, duplicates, odd categories.
- Convert with `to_numeric / to_datetime(errors="coerce")` and count what was coerced; use nullable and category dtypes.
- Missing values: drop, fill (median, mode, group-wise `transform`), forward-fill, or flag; think about why they are missing.
- Define duplicates by key; `sort_values + drop_duplicates(keep="last")` keeps the latest record.
- Standardise text with `.str` and mapping dictionaries; detect outliers with IQR or z-scores and treat them deliberately.
- Wrap cleaning in a function with an audit log and assertions so it is repeatable.

Quick quiz

6 questions · tap an answer to check it

1. What does `pd.to_numeric(s, errors="coerce")` do with "abc"?

A Raises an error

B Converts it to NaN

C Keeps it as text

D Converts it to 0

Answer: B. You can then count and inspect the NaNs.

2. Which fill keeps group differences?

A `fillna(df.col.mean())`

B `fillna(df.groupby("g")["col"].transform("median"))`

C `dropna()`

D `fillna(0)`

Answer: B. `transform` returns a value per row from its group.

3. What are the IQR outlier fences?

A mean $\pm$ 2 sd

B $Q1 - 1.5 \times IQR$ and $Q3 + 1.5 \times IQR$

C min and max

D median $\pm$ IQR

Answer: B. The IQR is $Q3 - Q1$.

4. How do you keep the latest record per order_id?

A `drop_duplicates()`

B Sort by update time, then `drop_duplicates(subset="order_id", keep="last")`

C `groupby().first()` without sorting

D `unique()`

Answer: B. Sorting decides which copy is "last".

5. Why add an `is_missing` flag before filling?

A Required by pandas

B The fact that a value was missing can itself be informative

C It speeds up filling

D To delete the column later

Answer: B. Models and analysts can use the flag.

6. Which dtype allows integers with missing values?

- A int64
- B Int64 (nullable)**
- C float32 only
- D object

Answer: B. The capital-I Int64 is pandas' nullable integer.

EXERCISE Cleaning practice

Using `hr_employees.csv` : (1) profile every column. (2) Convert `hire_date` to datetime and compute tenure in years. (3) Check whether `years_at_company` agrees with the hire date. (4) Make `education` an ordered category. (5) Find income outliers per department using the IQR rule within each department.

Show solution (worked answer)

Python

```
hr = pd.read_csv("hr_employees.csv")
print(profile(hr))
hr["hire_date"] = pd.to_datetime(hr["hire_date"])
hr["tenure"] = ((pd.Timestamp("2025-12-31") - hr.hire_date).dt.days / 365.25).round(1)
print((hr["tenure"].round() - hr["years_at_company"]).abs().describe())
print(hr["education"].unique())
def iqr_flag(x):
    q1, q3 = x.quantile([.25, .75]); i = q3 - q1
    return (x < q1 - 1.5 * i) | (x > q3 + 1.5 * i)
hr["income_outlier"] = hr.groupby("department")["monthly_income"].transform(iqr_flag)
print(hr.groupby("department")["income_outlier"].sum())
```

INTERVIEW CORNER Questions you may be asked

Q: How do you handle missing values? First find out why they are missing and how many. Then drop (few, random), impute (median/mode, group-wise, model-based, forward-fill for time series) or flag. Impute after the train/test split in ML to avoid leakage.

Q: How do you treat outliers? Check whether they are errors or real. Fix or remove errors; for real extremes use robust statistics (median), transformations (log), capping, or models robust to outliers.

Q: What checks do you run before trusting a dataset? Row counts vs source, uniqueness of keys, missing rates, value ranges, valid categories, date ranges, and totals reconciled to a trusted report.

pandas III: GroupBy, Pivot Tables, Merge/Join and Reshape (melt/unpivot)

Summarise and combine data. Use groupby with agg, transform and filter, build pivot tables and crosstabs, join tables with merge (inner, left, right, outer, anti), stack with concat, and reshape between wide and long with melt, pivot, stack and unstack.

● Python class 14

✦ pandas 2.3 / 3.0

Intermediate

🕒 120 min

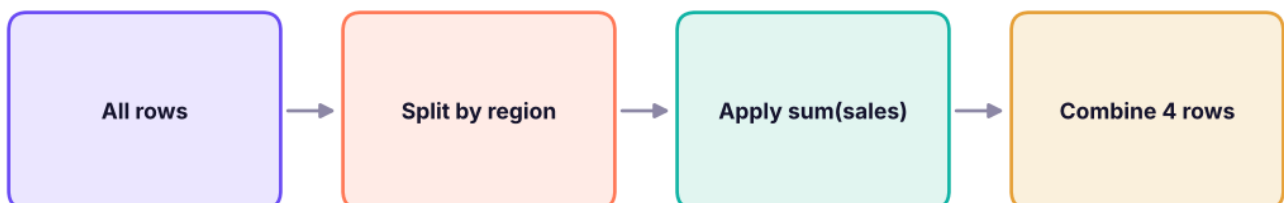
By the end of this lesson you can

- Apply split-apply-combine with groupby, named aggregations and multiple keys
- Use transform for group-level columns and filter for group-level conditions
- Build pivot_table and crosstab summaries with totals and percentages
- Join DataFrames with merge, choose the join type, and validate the result
- Reshape wide ↔ long with melt (unpivot) and pivot, and combine with concat

Split-apply-combine

groupby **splits** rows into groups, **applies** a calculation to each group, and **combines** the results. It is SQL's GROUP BY and Excel's PivotTable.

Split → apply → combine



groupby splits rows by key, applies an aggregation per group, and combines the answers.

Python

```
import pandas as pd, numpy as np
pd.set_option("display.max_columns", 20); pd.set_option("display.width", 160)
s = pd.read_csv("superstore_sales.csv", parse_dates=["order_date"])
print(s.groupby("region")["sales"].sum().sort_values(ascending=False))
print(s.groupby(["region", "category"])["profit"].sum().round(0).unstack())
```

OUTPUT

```
region
South    53952768.5
West     51958520.5
North    49018161.0
East     46484119.0
Name: sales, dtype: float64
category Furniture Office Supplies Technology
region
East      -209426.0          434731.0    4149185.0
North     -236825.0          331848.0    4460806.0
South     -259193.0          349958.0    4611247.0
West      -112710.0          398257.0    4434286.0
```

Named aggregations

Python

```
summary = (s.groupby("category")
            .agg(orders=("order_id", "nunique"), lines=("row_id", "size"), sales=("sales", "sum"),
                profit=("profit", "sum"), avg_discount=("discount", "mean"), max_line=("sales",
"max"))
            .assign(margin_pct=lambda d: (100 * d.profit / d.sales).round(1))
            .sort_values("sales", ascending=False))
print(summary.round(2))
print(s.groupby("segment", as_index=False)["sales"].agg(["sum", "mean", "median", "count"]).round(0))
```

OUTPUT

		orders	lines	sales	profit	avg_discount	max_line	margin_pct
category								
	Technology	1518	1892	156438412.5	17655523.58	0.07	619290.0	11.3
	Furniture	1511	1917	34841853.0	-818153.31	0.13	93120.0	-2.3
	Office Supplies	1870	2514	10133303.5	1514793.58	0.07	60830.0	14.9
segment								
	sum		mean	median	count			
0	Consumer	98603292.0	32066.0	8288.0	3075			
1	Corporate	59113619.0	31427.0	8480.0	1881			
2	Home Office	43696658.0	31965.0	9079.0	1367			

Common aggregations: `sum`, `mean`, `median`, `min`, `max`, `count` (non-null), `size` (rows), `nunique`, `std`, `first`, `last`, or your own function.

transform and filter

`transform` returns a value **for every row** (the group result broadcast back), perfect for "share of group" and group-wise filling. `filter` keeps whole groups that meet a condition.

Python

```
s["region_sales"] = s.groupby("region")["sales"].transform("sum")
s["share_of_region_%"] = (100 * s["sales"] / s["region_sales"]).round(3)
s["rank_in_region"] = s.groupby("region")["sales"].rank(ascending=False, method="first").astype(int)
print(s.loc[s.rank_in_region ≤ 2, ["region", "product_name", "sales", "share_of_region_%",
"rank_in_region"]].sort_values(["region", "rank_in_region"]))
loyal = s.groupby("customer_id").filter(lambda g: g["order_id"].nunique() ≥ 8)
print(loyal["customer_id"].nunique(), "customers placed 8+ orders")
```

OUTPUT

	region	product_name	sales	share_of_region_%	rank_in_region
6244	East	Vertex Copier 262	472740.0	1.017	1
4219	East	Orbit Copier 746	442350.0	0.952	2
3669	North	Orbit Copier 746	619290.0	1.263	1
1653	North	Orbit Copier 746	526396.5	1.074	2
1142	South	Vertex Copier 262	551530.0	1.022	1
2087	South	Vertex Copier 262	551530.0	1.022	2
1779	West	Apex Copier 347	493080.0	0.949	1
3317	West	Orbit Copier 746	477738.0	0.919	2

41 customers placed 8+ orders

Pivot tables and crosstabs

Python

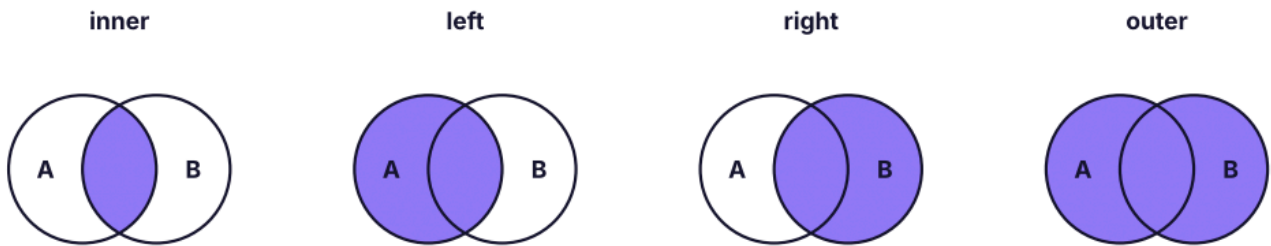
```
s["year"] = s["order_date"].dt.year
pt = pd.pivot_table(s, values="sales", index="category", columns="year", aggfunc="sum", margins=True,
margins_name="Total")
print((pt / 1e6).round(2))
print(pd.crosstab(s["segment"], s["ship_mode"], normalize="index").mul(100).round(1)) # row %
```

OUTPUT

year	2023	2024	2025	Total
category				
Furniture	9.95	12.13	12.76	34.84
Office Supplies	2.61	3.55	3.97	10.13
Technology	45.08	49.72	61.65	156.44
Total	57.64	65.40	78.38	201.41
ship_mode	First Class	Same Day	Second Class	Standard Class
segment				
Consumer	15.2	5.5	19.5	59.8
Corporate	13.2	5.0	20.3	61.5
Home Office	16.9	4.3	19.5	59.3

Merging (joins)

Real analysis needs several tables. `pd.merge` (or `df.merge`) joins on key columns like SQL joins.



Join types: which rows survive.

Python

```
orders = pd.DataFrame({"order_id": [1, 2, 3, 4], "cust": ["C1", "C2", "C2", "C9"], "amount": [500, 250, 900, 120]})
cust = pd.DataFrame({"cust": ["C1", "C2", "C3"], "name": ["Asha", "Ravi", "Meera"], "city": ["Pune", "Delhi", "Kochi"]})
for how in ["inner", "left", "right", "outer"]:
    print(how, "→", len(orders.merge(cust, on="cust", how=how)), "rows")
print(orders.merge(cust, on="cust", how="left", indicator=True))
anti = orders.merge(cust, on="cust", how="left", indicator=True).query("_merge == 'left_only'")
print("orders with unknown customer:", anti["order_id"].tolist())
```

OUTPUT

inner → 3 rows

left → 4 rows

right → 4 rows

outer → 5 rows

	order_id	cust	amount	name	city	_merge
0	1	C1	500	Asha	Pune	both
1	2	C2	250	Ravi	Delhi	both
2	3	C2	900	Ravi	Delhi	both
3	4	C9	120	NaN	NaN	left_only

orders with unknown customer: [4]

Argument	Use
<code>on="key" / on=["a", "b"]</code>	Same column name(s) in both
<code>left_on=, right_on=</code>	Different key names
<code>how=</code>	<code>inner</code> , <code>left</code> , <code>right</code> , <code>outer</code> , <code>cross</code>
<code>suffixes=("_x", "_y")</code>	Clashing column names
<code>indicator=True</code>	Adds <code>_merge</code> to see where each row came from
<code>validate="many_to_one"</code>	Raises an error if the key is not unique on the right

A real three-table join

Python

```
products = pd.read_csv("products.csv"); customers = pd.read_csv("customers.csv")
lines = s[["order_id", "customer_id", "product_id", "sales", "quantity"]]
full = (lines.merge(products[["product_id", "list_price"]], on="product_id", how="left",
validate="many_to_one")
        .merge(customers[["customer_id", "city", "segment"]], on="customer_id", how="left",
validate="many_to_one"))
print(len(lines), len(full), full["list_price"].isna().sum(), "unmatched products")
full["realised_vs_list_%"] = (100 * full["sales"] / (full["list_price"] * full["quantity"])).round(1)
print(full.groupby("segment")["realised_vs_list_%"].mean().round(1))
```

OUTPUT

```
6323 6323 0 unmatched products
segment
Consumer      91.0
Corporate      91.1
Home Office    90.4
Name: realised_vs_list_%, dtype: float64
```

WATCH OUT Row explosion

If the key is duplicated on **both** sides, a merge multiplies rows (many-to-many). Always compare row counts before and after, and use `validate=` to catch it.

concat: stacking tables

Python

```
jan = pd.DataFrame({"store": ["A", "B"], "sales": [100, 80]}); feb = pd.DataFrame({"store": ["A", "C"],
"sales": [120, 60]})
print(pd.concat([jan.assign(month="Jan"), feb.assign(month="Feb")], ignore_index=True)) # rows
(UNION ALL)
print(pd.concat([jan.set_index("store"), feb.set_index("store")], axis=1, keys=["Jan", "Feb"])) #
columns, aligned on index
```

OUTPUT

```
   store  sales month
0     A    100   Jan
1     B     80   Jan
2     A    120   Feb
3     C     60   Feb

      Jan   Feb
sales sales
store
A    100.0 120.0
```

B	80.0	NaN
C	NaN	60.0

Reshaping: wide ↔ long

Wide tables are easy to read (one column per month); **long** (tidy) tables are easy to analyse and chart (one row per observation). `melt` is pandas' **unpivot** (the same as Power Query's "Unpivot columns").

melt pivot

Wide

- region | Jan | Feb | Mar
- North | 10 | 12 | 9
- South | 8 | 11 | 14
- Readable report layout

Long (tidy)

- region | month | sales
- North | Jan | 10
- North | Feb | 12 ... (6 rows)
- Best for groupby, charts, models

melt turns columns into rows; pivot turns rows into columns.

Python

```
wide = pd.DataFrame({"region": ["North", "South"], "Jan": [10, 8], "Feb": [12, 11], "Mar": [9, 14]})
long = wide.melt(id_vars="region", var_name="month", value_name="sales")
print(long)
back = long.pivot(index="region", columns="month", values="sales")[["Jan", "Feb", "Mar"]]
print(back)
st = back.stack(); print(st.head(3)); print(st.unstack().equals(back))
```

OUTPUT

```
   region month  sales
0  North   Jan     10
1  South   Jan      8
2  North   Feb     12
3  South   Feb     11
4  North   Mar      9
5  South   Mar     14
month  Jan  Feb  Mar
region
North    10   12    9
South     8   11   14
region  month
North   Jan     10
        Feb     12
```

Mar 9
dtype: int64
True

`pivot` requires unique index/column pairs; `pivot_table` aggregates duplicates.

Python

```
monthly = (s.assign(month=s.order_date.dt.to_period("M").astype(str))
            .pivot_table(index="month", columns="category", values="sales", aggfunc="sum").tail(3) /
            1e6).round(2)
print(monthly)
print(monthly.reset_index().melt(id_vars="month", var_name="category", value_name="sales_m").head(4))
```

OUTPUT

category	Furniture	Office Supplies	Technology
month			
2025-10	1.15	0.31	4.68
2025-11	1.07	0.59	8.62
2025-12	1.31	0.40	5.94

	month	category	sales_m
0	2025-10	Furniture	1.15
1	2025-11	Furniture	1.07
2	2025-12	Furniture	1.31
3	2025-10	Office Supplies	0.31

Summary

KEY TAKEAWAYS

- `groupby(keys).agg(name=(col, func))` builds clean summaries; `transform` returns group values per row; `filter` keeps whole groups.
- `pivot_table` (with `margins=True`) and `crosstab` (with `normalize`) mirror Excel PivotTables.
- `merge(on=, how=)` joins tables; check row counts, use `indicator=True` for anti-joins and `validate=` to prevent row explosions.
- `concat` stacks rows (or aligns columns).
- `melt` = unpivot (wide → long); `pivot` / `pivot_table` = long → wide; `stack` / `unstack` move index levels.

Quick quiz

6 questions · tap an answer to check it

1. Which returns one value per original row?

A `groupby().agg()`

B `groupby().transform()`

C `groupby().size()`

D `pivot_table()`

Answer: B. transform broadcasts the group result back to each row.

2. Which merge keeps every order even when the customer is missing?

A inner

B left (orders on the left)

C right

D cross

Answer: B. Unmatched customer columns become NaN.

3. How do you find rows in A with no match in B?

A inner merge

B left merge with `indicator=True` and keep `_merge == "left_only"`

C outer merge without indicator

D concat

Answer: B. This is an anti-join.

4. What does `melt` do?

- A Pivots long to wide
- B Unpivots wide columns into rows (long format)**
- C Merges tables
- D Deletes columns

Answer: B. `pivot` goes the other way.

5. A merge produced more rows than the left table. Most likely cause?

- A A bug in pandas
- B Duplicate keys on the right side**
- C Missing values
- D Wrong dtype

Answer: B. Use `validate="many_to_one"` to catch this.

6. How do you add a grand-total row and column to a pivot table?

- A `totals=True`
- B `margins=True`**
- C `sum=True`
- D `fill_value=0`

Answer: B. `margins_name` changes the label.

EXERCISE GroupBy and merge practice

(1) Sales and profit by year and region, with margin %. (2) For each customer, the share of their spend in each category (use `transform`). (3) Top product per sub-category by profit. (4) Customers in `customers.csv` with no orders (anti-join). (5) Melt the year columns of a category × year pivot back to long format.

Show solution (worked answer)

Python

```
yr = s.groupby(["year", "region"]).agg(sales=("sales", "sum"), profit=("profit", "sum"))
yr["margin_%"] = (100 * yr.profit / yr.sales).round(1); print(yr)
cc = s.groupby(["customer_id", "category"])["sales"].sum().reset_index()
cc["share"] = cc.sales / cc.groupby("customer_id")["sales"].transform("sum")
top = s.groupby(["sub_category", "product_name"])
["profit"].sum().reset_index().sort_values("profit").groupby("sub_category").tail(1)
no_orders = customers.merge(s[["customer_id"]].drop_duplicates(), how="left",
indicator=True).query("_merge == 'left_only'")
print(len(no_orders))
pt = s.pivot_table(index="category", columns="year", values="sales", aggfunc="sum")
print(pt.reset_index().melt(id_vars="category", var_name="year", value_name="sales"))
```

INTERVIEW CORNER Questions you may be asked

Q: Difference between `agg` and `transform` ? `agg` reduces each group to one row; `transform` returns an output aligned to the original rows.

Q: How would you compute each product's share of its category's sales? `df.sales / df.groupby("category").sales.transform("sum")` .

Q: merge vs join vs concat? `merge` joins on columns (SQL-style); `join` is a shortcut that joins on the index; `concat` stacks along an axis without matching keys.

Q: What is tidy data? Each variable is a column, each observation a row, each value a cell; long format produced by `melt` is tidy.

pandas IV: Dates, Time Series, Resampling and Rolling Windows

Work with time. Parse dates, extract parts with `.dt`, do date arithmetic, set a `DatetimeIndex`, resample to weeks, months and quarters, compute rolling and expanding windows, shifts and growth rates, and build a cohort-style monthly report with the daily sales data.

● Python class 15

✦ pandas 2.3 / 3.0

Intermediate

🕒 110 min

By the end of this lesson you can

- Parse and format dates and extract year, month, weekday and quarter
- Do date arithmetic with `Timedelta` and `DateOffset`, including business days
- Resample time series and use rolling, expanding and exponentially weighted windows
- Compute period-over-period change, year-over-year growth and cumulative totals with `shift`, `pct_change` and `cumsum`
- Handle time zones and missing dates

Parsing dates

Python

```
import pandas as pd, numpy as np
pd.set_option("display.max_columns", 20); pd.set_option("display.width", 160)
d = pd.to_datetime(pd.Series(["2025-01-15", "2025-03-01", "2025-12-31"]))
print(d.dt.year.tolist(), d.dt.month_name().tolist(), d.dt.day_name().tolist(), d.dt.quarter.tolist(),
      d.dt.isocalendar().week.tolist())
print(pd.to_datetime("05/01/2025", dayfirst=True).date(), pd.to_datetime("05/01/2025").date())      # 5
Jan vs 1 May!
print(pd.to_datetime("2025-01-15 14:30").strftime("%d %b %Y, %I:%M %p"), pd.Timestamp("2025-02-28").is_month_end)
```

OUTPUT

```
[2025, 2025, 2025] ['January', 'March', 'December'] ['Wednesday', 'Saturday', 'Wednesday'] [1,
1, 4] [3, 9, 1]
2025-01-05 2025-05-01
15 Jan 2025, 02:30 PM True
```

WATCH OUT Day-first dates

Indian and European data often write 05/01/2025 for 5 January. Pass `dayfirst=True` or, better, an explicit `format="%d/%m/%Y"`. Check a few parsed values every time.

<code>.dt</code> accessor	Gives
<code>.year</code> , <code>.month</code> , <code>.day</code> , <code>.hour</code>	Parts
<code>.day_name()</code> , <code>.month_name()</code>	Text names
<code>.dayofweek</code> (Mon = 0), <code>.quarter</code> , <code>.isocalendar().week</code>	Calendar positions
<code>.to_period("M")</code> , <code>.to_period("Q")</code>	Month / quarter periods
<code>.floor("D")</code> , <code>.normalize()</code>	Truncate time
<code>.is_month_end</code> , <code>.days_in_month</code>	Calendar flags

Date arithmetic

Python

```
s = pd.read_csv("superstore_sales.csv", parse_dates=["order_date", "ship_date"])
s["ship_days"] = (s["ship_date"] - s["order_date"]).dt.days
print(s.groupby("ship_mode")["ship_days"].agg(["mean", "max"]).round(1))
t = pd.Timestamp("2025-01-31")
print(t + pd.Timedelta(days=30), t + pd.DateOffset(months=1), t + pd.offsets.MonthEnd(1), t +
pd.offsets.BDay(3))
print(np.busday_count("2025-10-01", "2025-11-01"), "working days in October 2025 (Mon-Fri)")
print(pd.date_range("2025-01-01", periods=4, freq="W-MON").date.tolist(), pd.date_range("2025-01-01",
"2025-06-30", freq="QE").date.tolist())
```

OUTPUT

```
          mean  max
ship_mode
First Class    1.5    2
Same Day       0.0    0
Second Class   3.0    4
Standard Class  5.6    7
2025-03-02 00:00:00 2025-02-28 00:00:00 2025-02-28 00:00:00 2025-02-05 00:00:00
23 working days in October 2025 (Mon-Fri)
[datetime.date(2025, 1, 6), datetime.date(2025, 1, 13), datetime.date(2025, 1, 20),
datetime.date(2025, 1, 27)] [datetime.date(2025, 3, 31), datetime.date(2025, 6, 30)]
```

`Timedelta` adds exact durations; `DateOffset` and offsets follow the calendar (month ends, business days).

A time series with a DatetimeIndex

Python

```
daily = pd.read_csv("daily_sales.csv", parse_dates=["date"]).set_index("date").sort_index()
print(daily.shape, daily.index.min().date(), "to", daily.index.max().date())
print(daily.loc["2023-03"].head(3)) # partial-string indexing: all of March 2023
print(daily.loc["2023-12-24":"2023-12-27"])
full = pd.date_range(daily.index.min(), daily.index.max(), freq="D")
print("missing days:", len(full.difference(daily.index)))
```

OUTPUT

(1461, 1) 2022-01-01 to 2025-12-31

	units_sold
date	
2023-03-01	260
2023-03-02	272
2023-03-03	234

	units_sold
date	
2023-12-24	343
2023-12-25	270
2023-12-26	232
2023-12-27	257

missing days: 0

Resampling

`resample` is `groupby` for time: change the frequency and aggregate.

Python

```
monthly = daily["units_sold"].resample("ME").sum()
print(monthly.head(4))
q = daily["units_sold"].resample("QE").agg(["sum", "mean", "max"]).round(1)
print(q.tail(4))
wk = daily["units_sold"].resample("W-SUN").sum()
print(wk.describe().round(0))
print(daily.groupby(daily.index.day_name())["units_sold"].mean().round(1).sort_values(ascending=False))
```

OUTPUT

date	
2022-01-31	5998
2022-02-28	5797
2022-03-31	6756
2022-04-30	7156

Freq: ME, Name: units_sold, dtype: int64

	sum	mean	max
date			
2025-03-31	30462	338.5	405
2025-06-30	34592	380.1	440

```

2025-09-30  34786  378.1  432
2025-12-31  33080  359.6  415
count      210.0
mean       2073.0
std        391.0
min        420.0
25%       1766.0
50%       2073.0
75%       2362.0
max        2801.0
Name: units_sold, dtype: float64
date
Sunday      323.3
Saturday    320.4
Friday      289.5
Tuesday     288.6
Thursday    288.5
Monday      287.9
Wednesday   287.3
Name: units_sold, dtype: float64

```

Frequencies: **D** day, **W** week, **ME** month end, **MS** month start, **QE** quarter, **YE** year, **h** hour, **min** minute.

Rolling and expanding windows

Python

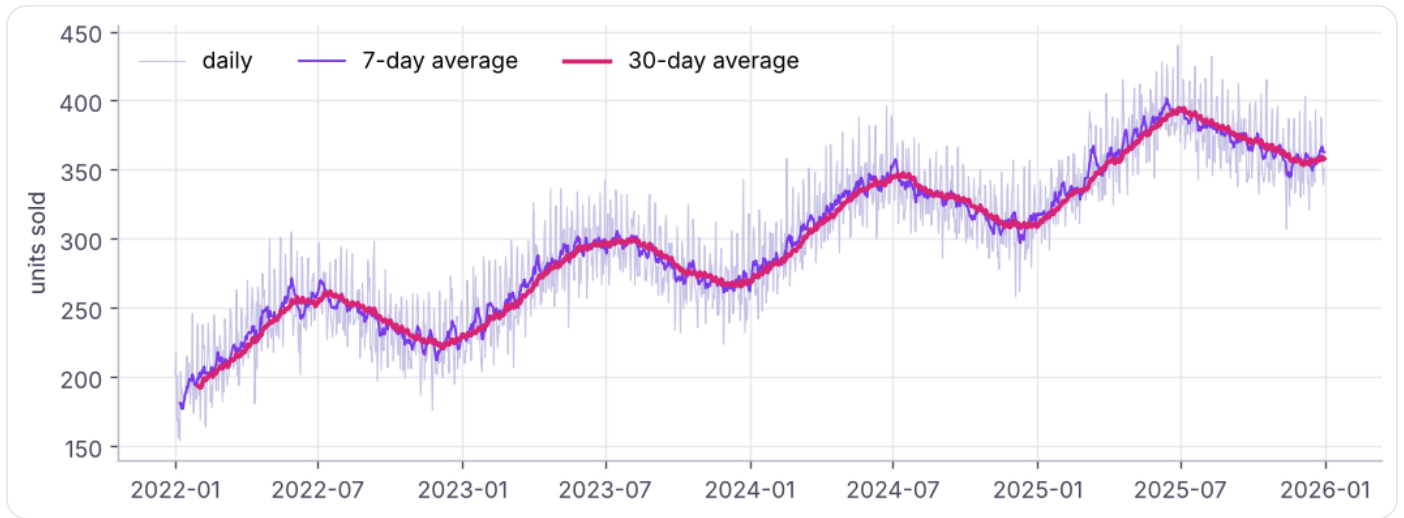
```

ts = daily["units_sold"]
df = pd.DataFrame({"units": ts, "ma_7": ts.rolling(7).mean(), "ma_30": ts.rolling(30).mean(), "ewm_14":
ts.ewm(span=14).mean(),
                  "max_so_far": ts.expanding().max(), "cum_units": ts.cumsum()})
print(df.iloc[[0, 6, 29, 400, -1]].round(1))

```

OUTPUT

	units	ma_7	ma_30	ewm_14	max_so_far	cum_units
date						
2022-01-01	202	NaN	NaN	202.0	202.0	202
2022-01-07	154	181.1	NaN	176.0	218.0	1268
2022-01-30	239	198.0	193.7	203.2	246.0	5811
2023-02-05	273	252.6	244.1	251.9	305.0	93681
2025-12-31	351	362.6	357.9	359.0	440.0	435285



Daily units with 7-day and 30-day moving averages: averages smooth out daily noise and reveal trend and seasonality.

Shifts and growth

Python

```
m = monthly.to_frame("units")
m["prev_month"] = m["units"].shift(1)
m["mom_%"] = (100 * m["units"].pct_change()).round(1)
m["yoy_%"] = (100 * m["units"].pct_change(12)).round(1) # same month last year
m["ytd"] = m.groupby(m.index.year)["units"].cumsum()
print(m.tail(4))
yr = daily["units_sold"].resample("YE").sum()
print(yr, (100 * yr.pct_change()).round(1).tolist())
```

OUTPUT

	units	prev_month	mom_%	yoy_%	ytd
date					
2025-09-30	11071	11798.0	-6.2	12.3	99840
2025-10-31	11291	11071.0	2.0	15.3	111131
2025-11-30	10721	11291.0	-5.0	14.7	121852
2025-12-31	11068	10721.0	3.2	15.1	132920
date					
2022-12-31	85031				
2023-12-31	100536				
2024-12-31	116798				
2025-12-31	132920				

Freq: YE-DEC, Name: units_sold, dtype: int64 [nan, 18.2, 16.2, 13.8]

`shift(1)` is SQL's `LAG` ; `shift(-1)` is `LEAD` ; `pct_change(12)` on monthly data is year-over-year growth.

Time zones

Python

```
ts_utc = pd.Timestamp("2025-03-10 09:00", tz="UTC")
print(ts_utc.tz_convert("Asia/Kolkata"), ts_utc.tz_convert("America/New_York"))
naive = pd.Series(pd.to_datetime(["2025-03-10 09:00"]))
print(naive.dt.tz_localize("Asia/Kolkata").dt.tz_convert("UTC").tolist())
```

OUTPUT

```
2025-03-10 14:30:00+05:30 2025-03-10 05:00:00-04:00
[Timestamp('2025-03-10 03:30:00+0000', tz='UTC')]
```

Store timestamps in UTC; convert to local time only for reporting.

Mini case: monthly KPI report

Python

```
s["month"] = s["order_date"].dt.to_period("M")
kpi = s.groupby("month").agg(sales=("sales", "sum"), orders=("order_id", "nunique"), customers=
("customer_id", "nunique"))
kpi["aov"] = (kpi.sales / kpi.orders).round(0)
kpi["sales_3m_avg"] = kpi.sales.rolling(3).mean().round(0)
kpi["yoy_%"] = (100 * kpi.sales.pct_change(12)).round(1)
print((kpi.tail(6).assign(sales=lambda d: (d.sales / 1e6).round(2), sales_3m_avg=lambda d:
(d.sales_3m_avg / 1e6).round(2))))
first = s.groupby("customer_id")["order_date"].min().dt.to_period("M").rename("first_month")
s2 = s.join(first, on="customer_id")
new_vs_repeat = s2.assign(kind=np.where(s2.month == s2.first_month, "new", "repeat")).groupby(["month",
"kind"])[["customer_id"]].nunique().unstack().tail(3)
print(new_vs_repeat)
```

OUTPUT

	sales	orders	customers	aov	sales_3m_avg	yoy_%
month						
2025-07	7.19	99	91	72627.0	5.86	49.2
2025-08	5.80	99	91	58608.0	5.65	0.8
2025-09	8.47	107	101	79141.0	7.15	58.9
2025-10	6.14	124	115	49518.0	6.80	-15.4
2025-11	10.28	152	139	67661.0	8.30	41.2
2025-12	7.65	134	121	57106.0	8.03	15.3
kind	new	repeat				
month						
2025-10	8.0	107.0				
2025-11	4.0	135.0				
2025-12	5.0	116.0				

Summary

KEY TAKEAWAYS

- Parse with `pd.to_datetime` (give format or dayfirst); use `.dt` for parts and `to_period` for months/quarters.
- `Timedelta` = exact duration; `DateOffset` / `MonthEnd` / `BDay` = calendar-aware moves; `date_range` builds calendars.
- A sorted `DatetimeIndex` allows partial-string slicing (`df.loc["2025-03"]`) and `resample` .
- Rolling, expanding and EWM windows smooth series; `shift` = lag/lead, `pct_change(n)` = growth, `cumsum` = running totals.
- Keep timestamps in UTC, check for missing dates, and convert for display.

Quick quiz

6 questions · tap an answer to check it

1. What does `daily.resample("ME").sum()` produce?

A Daily totals

B Monthly totals (labelled at month end)

C A 7-day average

D An error

Answer: B. ME = month end frequency.

2. Which gives year-over-year growth on monthly data?

A `pct_change(1)`

B `pct_change(12)`

C `shift(-12)`

D `rolling(12).mean()`

Answer: B. Compare each month with the same month a year earlier.

3. `shift(1)` is equivalent to which SQL function?

A LAG

B LEAD

C ROW_NUMBER

D NTILE

Answer: A. `shift(-1)` is LEAD.

4. How do you parse "05/01/2025" as 5 January?

A `pd.to_datetime(x)`

B `pd.to_datetime(x, dayfirst=True)` or `format="%d/%m/%Y"`

C `.dt.day`

D `astype(str)`

Answer: B. The default assumes month first.

5. What does `ts.rolling(7).mean()` give for the first 6 rows?

A 0

B NaN (not enough values yet)

C The value itself

D An error

Answer: B. Use `min_periods=1` to allow partial windows.

6. What is the difference between `Timedelta(days=30)` and `DateOffset(months=1)` from 31 Jan?

- A** `Timedelta` adds exactly 30 days (2 Mar); `DateOffset` moves to the same day next month, clipped to 28 Feb
- B** They are identical
- C** `Timedelta` is calendar-aware
- D** `DateOffset` adds 30 days

Answer: A. Use calendar offsets for month logic.

EXERCISE Time-series practice (daily_sales.csv)

(1) Which month had the highest total units? (2) Compute the 28-day rolling sum and its maximum. (3) Average units by weekday and month (a heatmap-ready pivot). (4) Year-over-year growth by quarter. (5) Flag days where units exceed the 30-day average by 50%.

[Show solution \(worked answer\)](#)

Python

```
print(monthly.idxmax(), monthly.max())
print(ts.rolling(28).sum().max())
pv = daily.assign(wd=daily.index.day_name(), m=daily.index.month).pivot_table(index="wd",
columns="m", values="units_sold", aggfunc="mean").round(0)
qq = ts.resample("QE").sum(); print((100 * qq.pct_change(4)).round(1).dropna())
spikes = daily[ts > 1.5 * ts.rolling(30).mean()]; print(len(spikes))
```

INTERVIEW CORNER Questions you may be asked

Q: resample vs groupby? `resample` is a time-based groupby on a `DatetimeIndex` that also creates empty periods; `groupby` with `dt.to_period` is similar but only returns periods present in the data.

Q: How do you compute a 7-day moving average in pandas and in SQL? `s.rolling(7).mean()`; in SQL `AVG(x) OVER (ORDER BY d ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)`.

Q: How do you handle missing dates in a daily series? Reindex to a full `date_range`, then decide: fill with 0 (no sales), forward-fill (stock levels), or interpolate (sensor readings).

Data Visualisation I: Matplotlib and Choosing the Right Chart

Pick the right chart for the question and build it with Matplotlib's object-oriented API. Line, bar, histogram, scatter, box and pie charts, subplots, annotations, styling and saving, plus design rules that make charts honest and readable.

● Python class 16

✦ Matplotlib 3.10

Intermediate

🕒 110 min

By the end of this lesson you can

- Choose a chart type from the analytical question (trend, comparison, distribution, relationship, composition)
- Build charts with `fig, ax = plt.subplots()` and customise titles, labels, ticks, legends and colours
- Combine several charts with subplots and annotate key points
- Plot directly from pandas and save figures at publication quality
- Apply design rules: start bars at zero, remove clutter, label directly, use colour with purpose

Choose the chart from the question

Which chart?

Trend over time

Line chart (area for cumulative)

Compare categories

Bar / column chart, sorted

Distribution

Histogram, box plot, violin

Relationship

Scatter (+ trend line),
heatmap of correlations

Part of a whole

Stacked bar; pie only for
2–4 parts

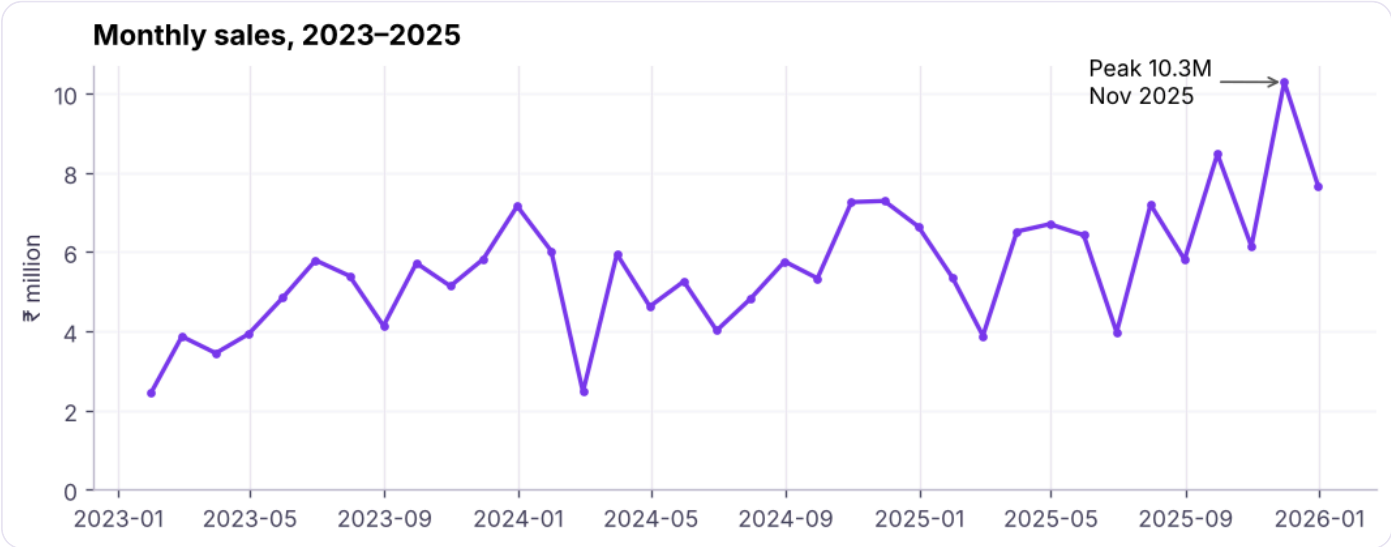
Geography

Map / choropleth

Start from the question, not the chart type.

The object-oriented API

Matplotlib has two styles. Use the **object-oriented** one: create a **Figure** (the canvas) and one or more **Axes** (plots), then call methods on the axes.



Monthly Superstore sales: a line chart shows the trend and the seasonal peaks.

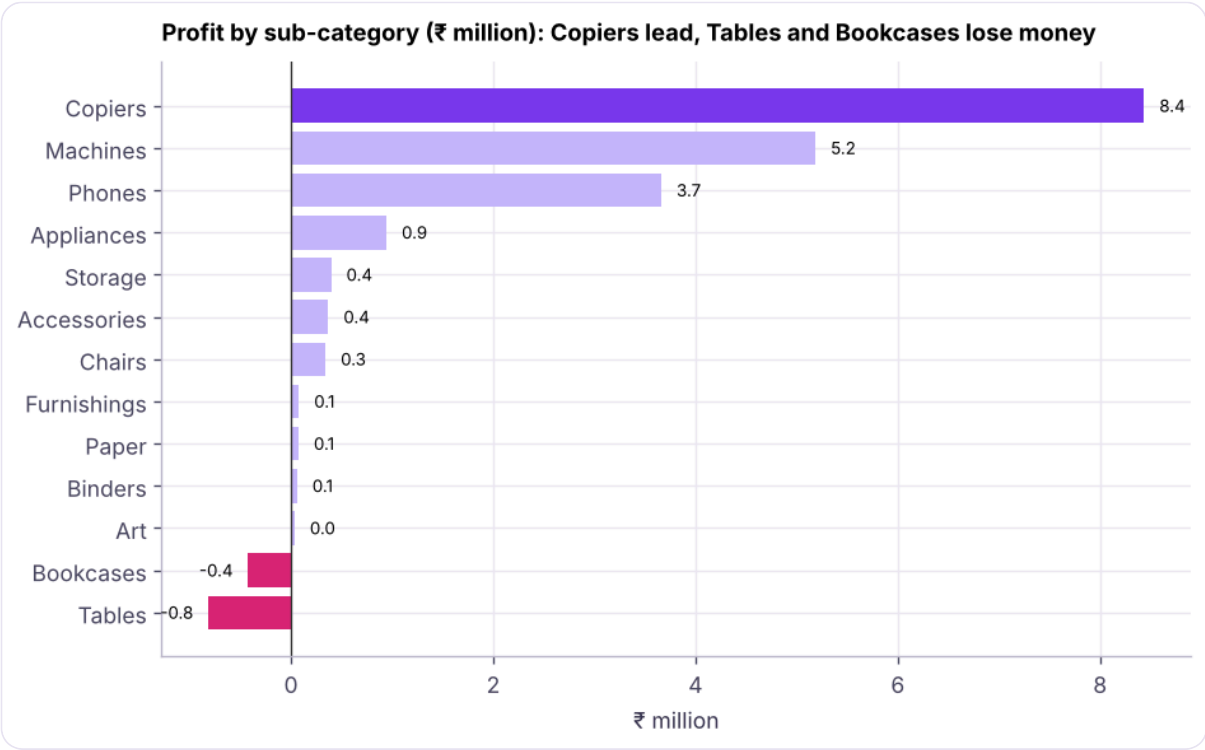
Python

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(9, 3.6)) # one figure, one axes
ax.plot(x, y) # draw
ax.set_title("..."); ax.set_xlabel("..."); ax.set_ylabel("...")
fig.savefig("chart.png", dpi=200, bbox_inches="tight")
plt.show() # in scripts; notebooks display automatically
```

Element	Method
Title, axis labels	<code>ax.set_title</code> , <code>ax.set_xlabel</code> , <code>ax.set_ylabel</code>
Limits, ticks	<code>ax.set_ylim(0)</code> , <code>ax.set_xticks</code> , <code>ax.tick_params(rotation=45)</code>
Legend	<code>label="..."</code> in the plot call, then <code>ax.legend()</code>
Grid, spines	<code>ax.grid(axis="y")</code> , <code>ax.spines[["top","right"]].set_visible(False)</code>
Text, arrows	<code>ax.text</code> , <code>ax.annotate</code>
Reference lines	<code>ax.axhline(y)</code> , <code>ax.axvline(x)</code> , <code>ax.axvspan</code>
Number format	<code>ax.yaxis.set_major_formatter(...)</code>

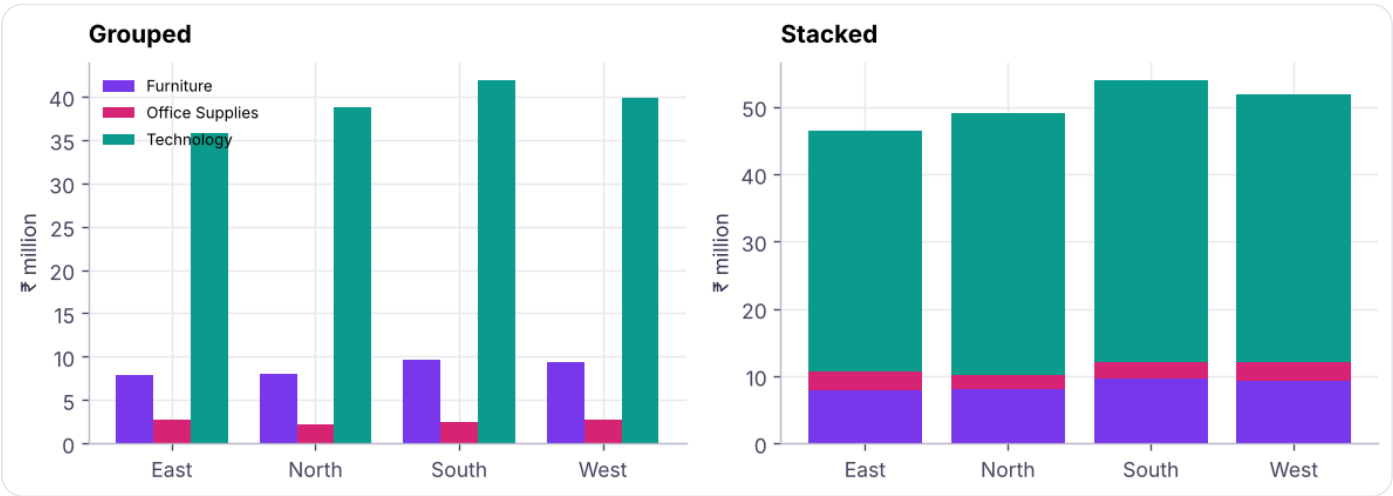


Bar charts



Sorted horizontal bars make category comparisons easy; one highlight colour draws the eye.

Grouped and stacked bars compare two dimensions:



Sales by region and category, as grouped bars (compare within region) and stacked bars (compare totals).



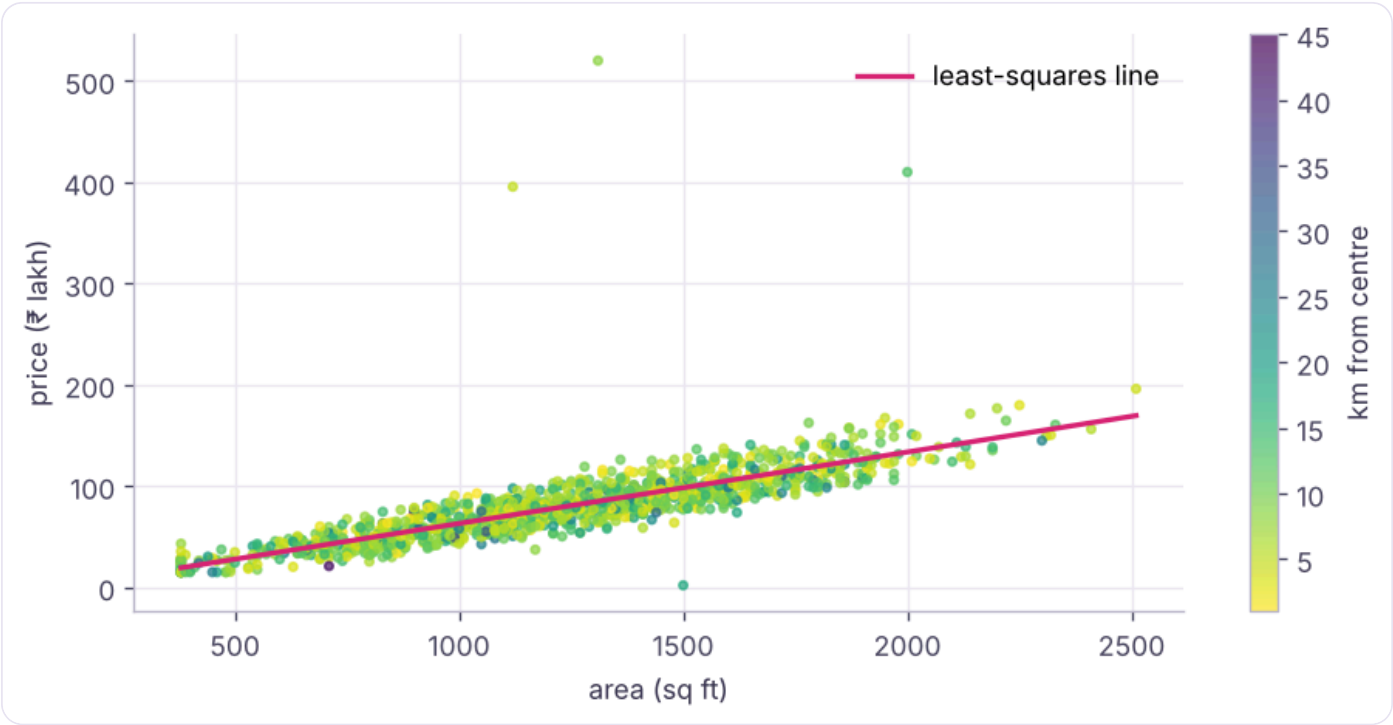
Distributions: histogram and box plot



House prices are right-skewed: the mean sits above the median. Box plots compare distributions across cities.

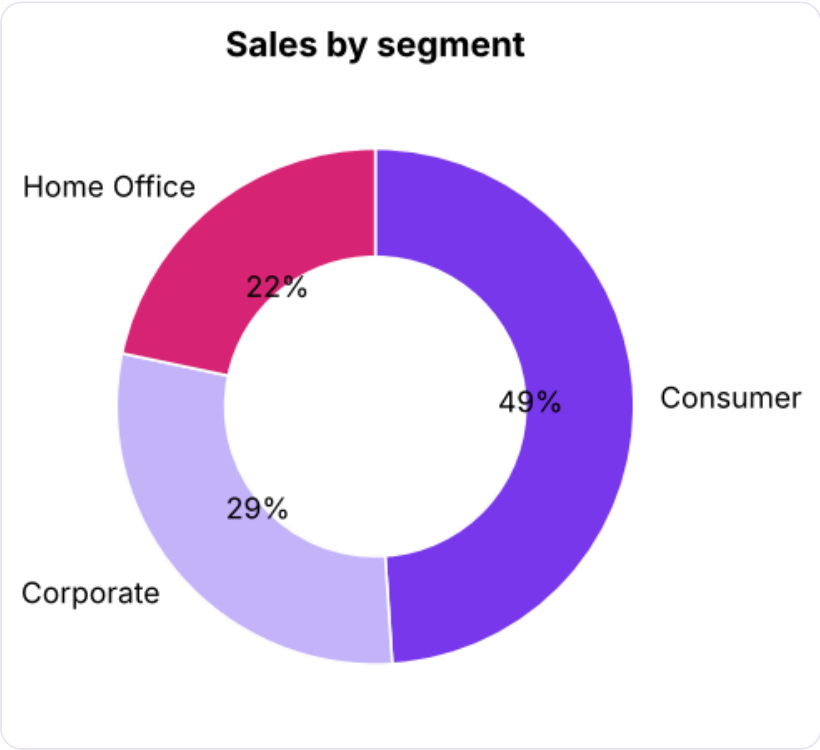
Bins matter: too few hide the shape, too many look noisy. Try `bins="auto"` or a few values.

Relationships: scatter



Larger houses cost more ($r \approx 0.8$); colour shows a third variable, the distance to the centre.

Part of a whole

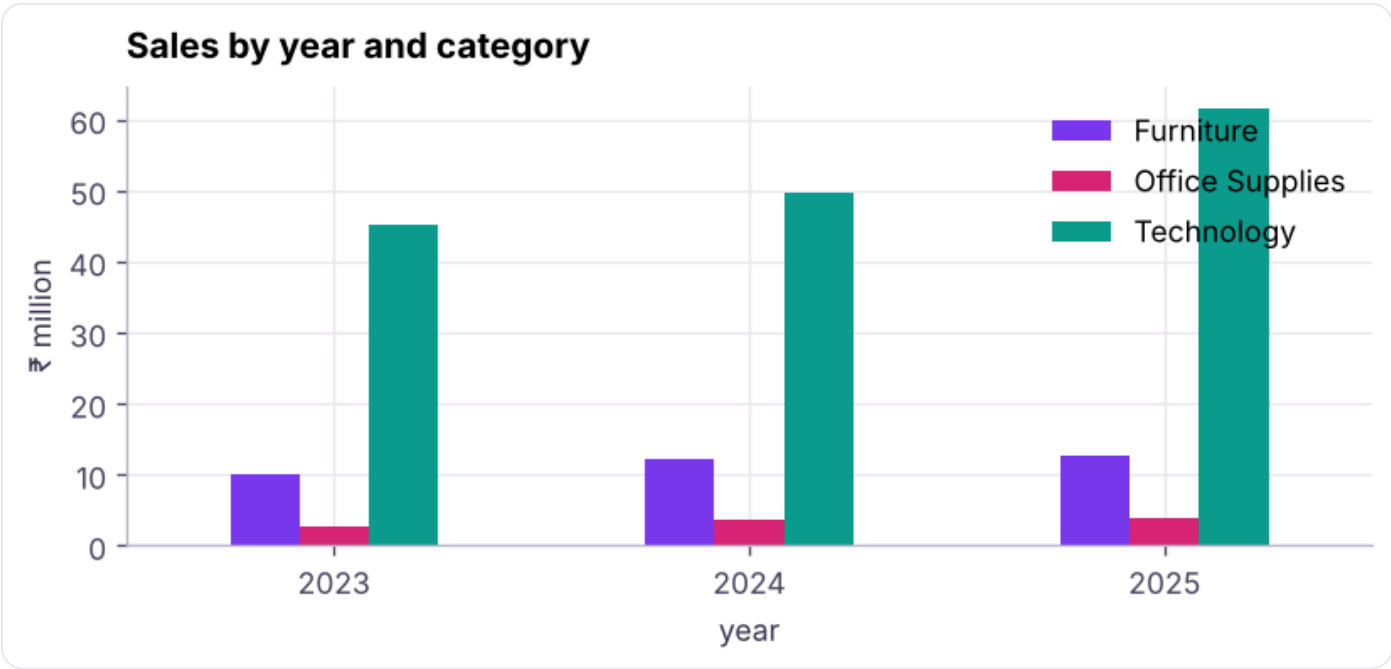


Pie charts work for a few parts; the donut variant leaves room for a label.

For more than 4 parts, or to compare parts across groups, use (stacked) bars.

Plotting from pandas

DataFrames and Series have `.plot()`, a quick wrapper around Matplotlib that returns an `ax` you can keep styling.



One line of pandas plotting, then a little Matplotlib styling.

`kind=` can be `line` , `bar` , `barh` , `hist` , `box` , `scatter` (with `x=` , `y=`), `area` , `pie` .

Subplots dashboards and saving

Python

```
fig, axes = plt.subplots(2, 2, figsize=(11, 7), constrained_layout=True)
axes[0, 0].plot(...); axes[0, 1].barh(...); axes[1, 0].hist(...); axes[1, 1].scatter(...)
fig.suptitle("Sales overview", fontsize=15, weight="bold")
fig.savefig("overview.png", dpi=200, bbox_inches="tight") # also .svg / .pdf for crisp vector output
```

Design rules that make charts honest

Rule	Why
Bars start at zero	Truncated bars exaggerate differences
Sort categories by value	Faster comparison
Title states the insight	"Tables lose money" beats "Profit by sub-category"
Label directly, remove chart junk	Fewer eye movements
One highlight colour, grey for context	Colour carries meaning
Colour-blind safe palettes (viridis, Okabe-Ito)	About 1 in 12 men are colour-blind
Same scale when comparing panels	Avoid misleading comparisons
Avoid 3-D and dual axes when possible	They distort perception

Summary

KEY TAKEAWAYS

- Pick the chart from the question: trend → line, comparison → sorted bar, distribution → histogram/box, relationship → scatter, composition → stacked bar (pie only for a few parts).
- Use `fig, ax = plt.subplots()` and style the axes: titles, labels, limits, spines, legends, annotations.
- `plt.subplots(rows, cols)` builds small multiples; `fig.savefig(..., dpi=200, bbox_inches="tight")` exports.
- `pandas .plot()` is a fast start that returns a Matplotlib axes.
- Honest design: zero-based bars, sorted categories, insight titles, purposeful colour.

Quick quiz

6 questions · tap an answer to check it

1. Which chart best shows a trend over 36 months?

A Pie

B Line

C Histogram

D Box plot

Answer: B. Lines connect ordered time points.

2. Which chart shows the distribution of one numeric variable?

A Scatter

B Histogram

C Pie

D Stacked bar

Answer: B. Box and violin plots also work.

3. What does `fig, ax = plt.subplots(1, 2)` create?

A One chart

B One figure with two axes side by side

C Two separate windows

D A 2×2 grid

Answer: B. `ax` becomes an array of two axes.

4. Why should bar charts start at zero?

- A Matplotlib requires it
- B Bar length encodes value; a cut axis exaggerates differences
- C It looks better
- D It saves space

Answer: B. Line charts can zoom; bars should not.

5. When is a pie chart acceptable?

- A For 12 categories
- B For a few parts of one whole, when exact comparison is not needed
- C For trends
- D Never

Answer: B. Otherwise use bars.

6. How do you save a sharp image without cut-off labels?

- A `plt.show()`
- B `fig.savefig("c.png", dpi=200, bbox_inches="tight")`
- C Screenshot
- D `fig.close()`

Answer: B. SVG/PDF give vector output.

EXERCISE Chart practice

(1) Line chart of monthly profit with a horizontal line at zero. (2) Sorted horizontal bar of sales by state (top 10). (3) Histogram of `discount`. (4) Scatter of `discount` vs `profit` (what do you see?). (5) A 2×2 dashboard of these four charts saved as PNG.

Show solution (worked answer)

Python

```
mp = s.set_index("order_date")["profit"].resample("ME").sum()
fig, axes = plt.subplots(2, 2, figsize=(11, 7), constrained_layout=True)
axes[0, 0].plot(mp.index, mp / 1e6); axes[0, 0].axhline(0, color="grey")
st = s.groupby("state")["sales"].sum().nlargest(10).sort_values(); axes[0, 1].barh(st.index, st / 1e6)
axes[1, 0].hist(s.discount, bins=9)
axes[1, 1].scatter(s.discount, s.profit, s=4, alpha=.3) # profit falls as discount rises
fig.savefig("dashboard.png", dpi=150, bbox_inches="tight")
```

INTERVIEW CORNER Questions you may be asked

Q: How do you choose a chart? Start from the question (trend, comparison, distribution, relationship, composition), the number of variables and the audience, then remove anything that does not help the message.

Q: What makes a chart misleading? Truncated bar axes, inconsistent scales, 3-D effects, cherry-picked ranges, dual axes with arbitrary scaling, and areas that do not match values.

Q: Matplotlib vs Seaborn vs Plotly? Matplotlib is the low-level foundation with full control; Seaborn adds statistical charts with good defaults on top; Plotly makes interactive web charts.

Data Visualisation II: Seaborn Statistical Charts and Interactive Plotly

Make statistical charts quickly with Seaborn (distributions, categorical comparisons, relationships, correlation heatmaps, pair plots and facets) using the HR attrition data, and build interactive charts and simple dashboards with Plotly Express and Streamlit.

● Python class 16

✦ Seaborn 0.13 · Plotly 6

Intermediate

⌚ 100 min

By the end of this lesson you can

- Use Seaborn's figure-level and axes-level functions with hue, col and row
- Compare distributions across groups with histplot, kdeplot, boxplot, violinplot and countplot
- Show relationships with scatterplot, regplot, heatmaps of correlations and pairplot
- Build interactive charts with Plotly Express and know when to use Streamlit
- Turn charts into a clear data story

Seaborn in one idea

Seaborn works directly with **long DataFrames**: you name the columns for `x`, `y`, `hue` (colour), `col` / `row` (small multiples), and it groups, aggregates and adds legends for you.

Python

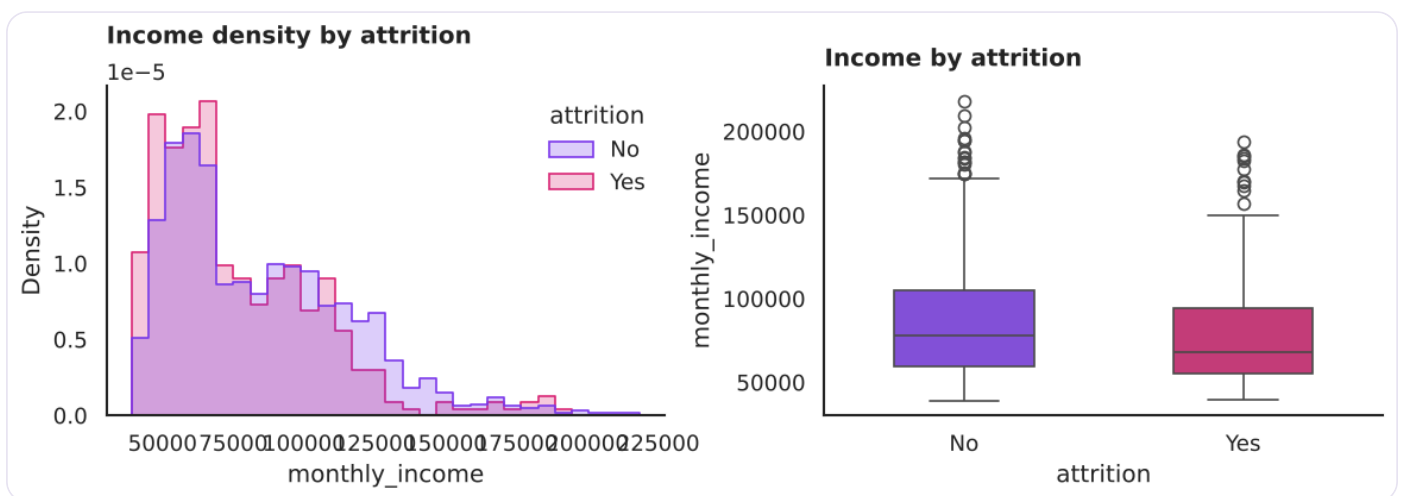
```
import pandas as pd, numpy as np, seaborn as sns, matplotlib.pyplot as plt
hr = pd.read_csv("hr_employees.csv")
hr["left"] = (hr["attrition"] == "Yes").astype(int)
print(hr.shape, f"attrition rate {hr.left.mean():.1%}", sns.__version__)
print(hr.groupby("overtime")["left"].mean().round(3).to_dict(), hr.groupby("department")
["left"].mean().round(3).sort_values().to_dict())
```

OUTPUT

```
(1500, 16) attrition rate 25.9% 0.13.2
{'No': 0.216, 'Yes': 0.367} {'Ooperations': 0.219, 'Finance': 0.227, 'HR': 0.229, 'IT': 0.243,
'R&D': 0.26, 'Sales': 0.304}
```

Question	Seaborn function
Distribution of one variable	<code>histplot</code> , <code>kdeplot</code> , <code>ecdfplot</code>
Distribution by group	<code>boxplot</code> , <code>violinplot</code> , <code>stripplot</code> , <code>boxenplot</code>
Counts of categories	<code>countplot</code>
Average per category (with CI)	<code>barplot</code> , <code>pointplot</code>
Two numeric variables	<code>scatterplot</code> , <code>regplot</code> , <code>lmplot</code> , <code>jointplot</code>
Many variables	<code>pairplot</code> , <code>heatmap</code> of <code>df.corr()</code>
Small multiples	<code>displot</code> , <code>catplot</code> , <code>relplot</code> with <code>col=</code> / <code>row=</code>

Distributions



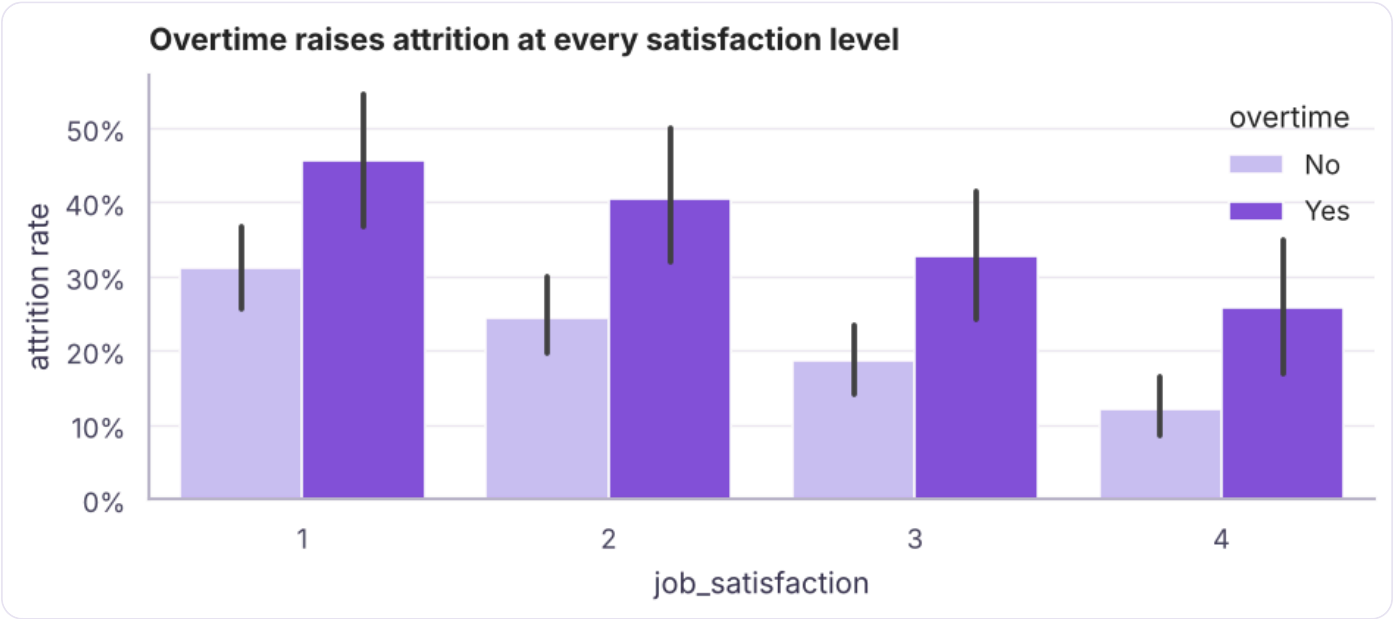
Income distribution for leavers vs stayers: leavers cluster at lower incomes.

`common_norm=False` scales each group separately, so a small group (leavers) is not dwarfed by a big one.



Violin plots combine a box plot with the distribution shape; split by overtime.

Categorical comparisons



Attrition rate by overtime and job satisfaction. Bars show the mean with a 95% confidence interval.

Python

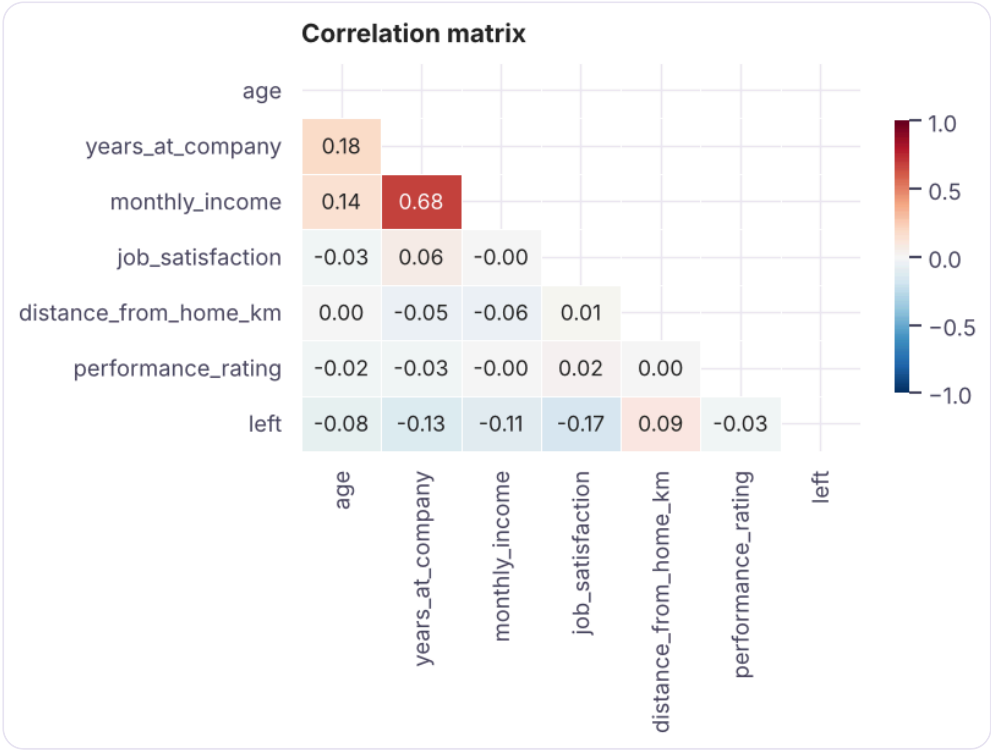
```
print((hr.pivot_table(index="job_satisfaction", columns="overtime", values="left", aggfunc="mean") * 100).round(1))
```

OUTPUT

overtime	No	Yes
job_satisfaction		
1	31.1	45.5
2	24.4	40.5
3	18.7	32.8
4	12.0	25.8



Relationships and correlation



Correlation heatmap of numeric HR variables (Pearson r). Only the lower triangle is shown.

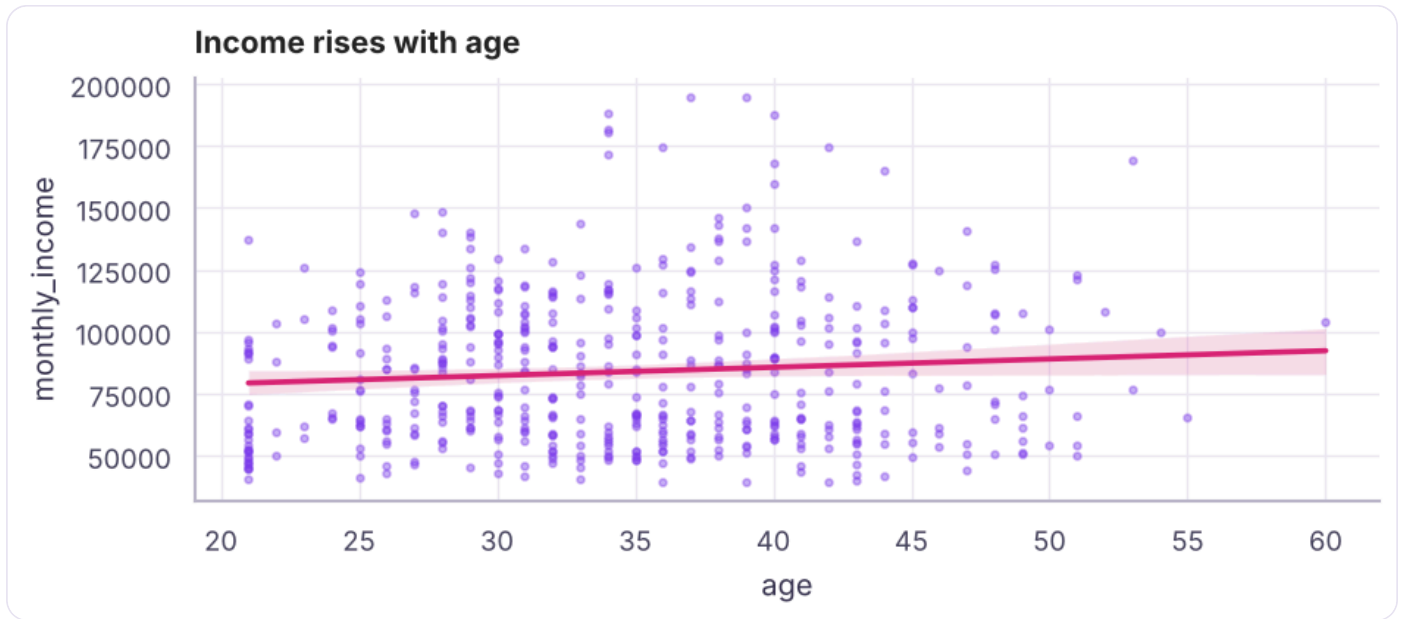
Python

```
print(corr["left"].drop("left").sort_values())
```

OUTPUT

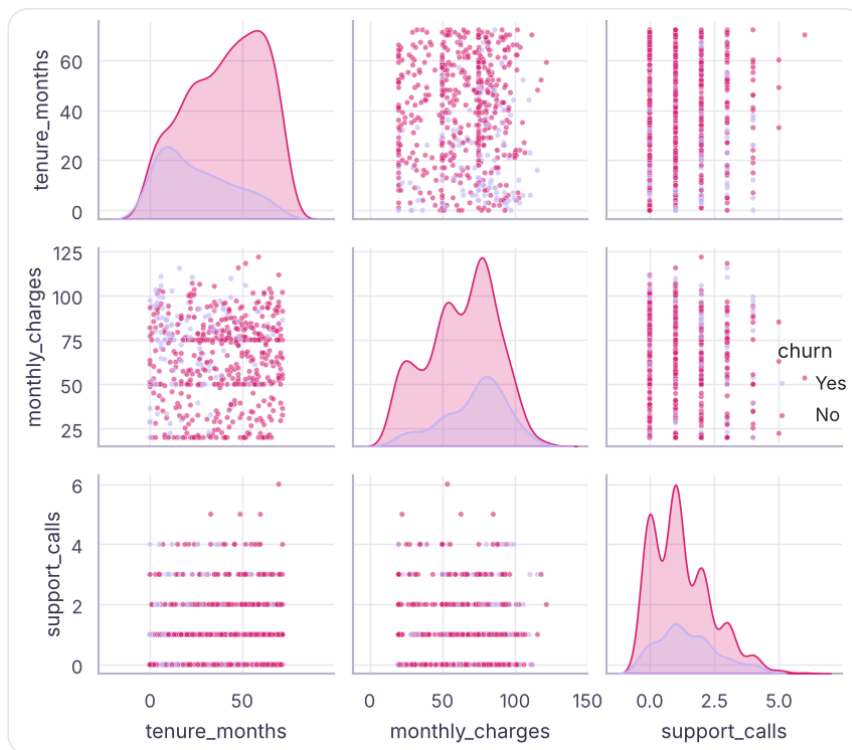
```
job_satisfaction      -0.17
years_at_company      -0.13
monthly_income        -0.11
age                   -0.08
performance_rating    -0.03
distance_from_home_km  0.09
Name: left, dtype: float64
```

Correlation measures **linear** association between -1 and +1. It does not prove causation, and it can miss non-linear patterns: always look at the scatter too.

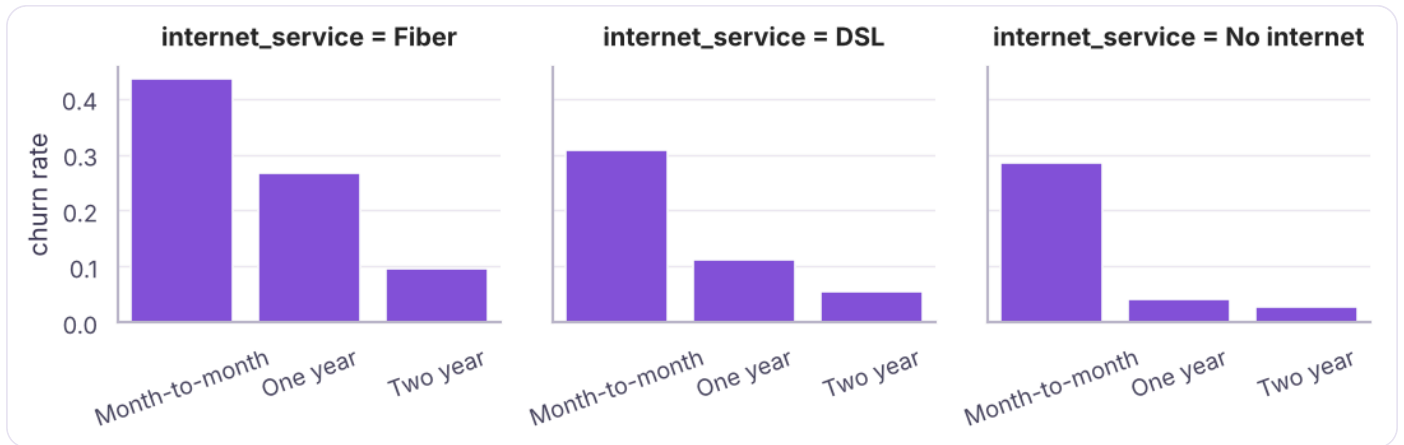


regplot adds a fitted line with a confidence band; Implot repeats it per group.

Pair plots and facets



A pair plot shows every pairwise scatter and each distribution, coloured by churn (telecom data).



Facets (small multiples) with catplot: churn rate by contract type, split by internet service.

Python

```
print((tc.pivot_table(index="contract", columns="internet_service", values="churned", aggfunc="mean") *
100).round(1))
```

OUTPUT

internet_service	DSL	Fiber	No internet
contract			
Month-to-month	30.8	43.5	28.5
One year	11.1	26.7	4.1
Two year	5.6	9.5	2.8

Interactive charts with Plotly Express

Plotly makes charts you can hover, zoom and filter in a browser or notebook. The syntax mirrors Seaborn:

Python

```
import plotly.express as px
fig = px.scatter(tc, x="tenure_months", y="monthly_charges", color="churn",
                 hover_data=["contract", "payment_method"], opacity=.6,
                 title="Churn by tenure and charges")
fig.show() # interactive in notebooks
fig.write_html("churn.html") # share as a standalone web page

px.line(monthly_df, x="month", y="sales", color="category")
px.bar(region_df, x="region", y="sales", text_auto=".2s")
px.histogram(hr, x="monthly_income", color="attrition", barmode="overlay")
px.choropleth(...) # maps
```

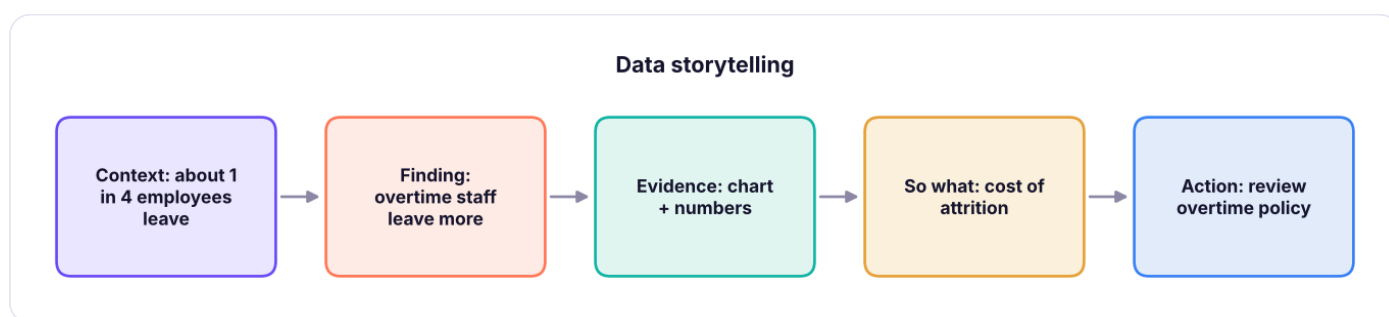
A mini dashboard app with Streamlit

Python

```
# app.py → run with: streamlit run app.py
import streamlit as st, pandas as pd, plotly.express as px
df = pd.read_csv("telecom_churn.csv")
contract = st.sidebar.multiselect("Contract", df.contract.unique(), default=list(df.contract.unique()))
d = df[df.contract.isin(contract)]
st.title("Churn dashboard")
st.metric("Churn rate", f"{(d.churn == 'Yes').mean():.1%}")
st.plotly_chart(px.histogram(d, x="tenure_months", color="churn", barmode="overlay"))
```

Streamlit turns a Python script into a web app with filters, which is useful for sharing analyses and ML demos (see the GenAI track for chat apps).

From charts to a data story



The story arc: context, finding, evidence, recommendation.

1. **One message per chart**, written as the title.
2. Show the comparison that proves it (overtime vs not), not every chart you made.
3. Quantify impact (people, ₹, %).
4. End with a recommendation and its owner.

Summary

KEY TAKEAWAYS

- Seaborn plots long DataFrames by column name with `hue`, `col` and `row`; it sits on Matplotlib, so you can still style the axes.
- Distributions: `histplot` (with `common_norm=False`), `kdeplot`, `boxplot`, `violinplot`; categories: `countplot`, `barplot` with confidence intervals.
- Relationships: `scatterplot`, `regplot`, `pairplot`, `correlation heatmap` (correlation $\neq$ causation).
- Facets (`catplot`, `relplot`, `displot`) create small multiples.
- Plotly Express gives interactive charts; Streamlit turns scripts into dashboards. Tell a story: one message per chart.

Quick quiz

6 questions · tap an answer to check it

1. What does `hue="attrition"` do in Seaborn?

- A Changes the theme
- B Colours the marks by the attrition category
- C Filters to attrition = Yes
- D Adds a title

Answer: B. `col=` would create separate panels instead.

2. Which chart shows the mean per category with a confidence interval?

- A countplot
- B barplot
- C histplot
- D heatmap

Answer: B. `countplot` shows counts.

3. A correlation of -0.8 means...

- A No relationship
- B A strong negative linear relationship
- C One variable causes the other to fall
- D A weak relationship

Answer: B. Correlation does not imply causation.

4. Which function creates small multiples of bar charts?

A barplot

B catplot(kind="bar", col=...)

C heatmap

D pairplot

Answer: B. Figure-level functions accept `col` and `row`.

5. What is the main advantage of Plotly over Matplotlib?

A Faster to print

B Interactive hover, zoom and filtering in the browser

C More chart types for PDFs

D No code needed

Answer: B. Export with `write_html`.

6. Why use `common_norm=False` in a histplot with hue?

A So each group's density is scaled to its own size and small groups stay visible

B To remove the legend

C To use counts

D To normalise to 100

Answer: A. Otherwise the larger group dominates.

EXERCISE Seaborn practice (telecom_churn.csv)

(1) Histogram of `monthly_charges` by churn. (2) Box plot of `tenure_months` by contract. (3) Churn rate by payment method, sorted. (4) Correlation heatmap of numeric columns plus `churned`. (5) Write three insight titles from what you see.

Show solution (worked answer)

Python

```
sns.histplot(tc, x="monthly_charges", hue="churn", common_norm=False, stat="density")
sns.boxplot(tc, x="contract", y="tenure_months")
order = tc.groupby("payment_method")["churned"].mean().sort_values(ascending=False).index
sns.barplot(tc, x="payment_method", y="churned", order=order, errorbar=None)
sns.heatmap(tc.select_dtypes("number").corr(), annot=True, fmt=".2f", cmap="RdBu_r", center=0)
# e.g. "Month-to-month customers churn far more", "Churners have short tenure", "Churn rises with support calls"
```

INTERVIEW CORNER Questions you may be asked

Q: How would you visualise the relationship between a categorical and a numeric variable? Box or violin plots per category (distribution), or bar/point plots of the mean with confidence intervals.

Q: How do you show correlations among many numeric variables? A correlation heatmap (masked upper triangle) followed by scatter plots for the interesting pairs; a pair plot for a handful of variables.

Q: How do you present findings to non-technical stakeholders? Lead with the recommendation, show one clear chart per point with an insight title, quantify impact, and keep methods in an appendix.

Exploratory Data Analysis (EDA) Case Study: Telecom Customer Churn

Run a complete, structured EDA on 3,000 telecom customers. Frame the question, check data quality, do univariate, bivariate and segment analysis, engineer features, quantify revenue at risk and finish with findings and recommendations: the template you will reuse for every dataset and for the ML churn model.

● Python class 17

✦ pandas 3 · Seaborn 0.13

Intermediate

🕒 150 min

By the end of this lesson you can

- Follow a repeatable EDA checklist from question to recommendation
- Check data quality and consistency between related columns
- Analyse univariate distributions and bivariate relationships with the target
- Build segments and quantify business impact (revenue at risk)
- Write findings as clear, numbered insights with actions

The EDA checklist

EDA workflow



A structured EDA moves from the business question to actions.

Business question. A telecom operator loses customers every month. *Which customers churn, why, and where should retention money go first?* **Target:** churn (Yes/No). **Metric:** churn rate = churned ÷ customers.

1. Load and first look

Python

```
import pandas as pd, numpy as np, seaborn as sns, matplotlib.pyplot as plt
pd.set_option("display.max_columns", 20); pd.set_option("display.width", 160)
tc = pd.read_csv("telecom_churn.csv")
tc["churned"] = (tc["churn"] == "Yes").astype(int)
print(tc.shape); print(tc.head(3))
print(tc.dtypes.astype(str).to_dict())
print(f"Overall churn rate: {tc.churned.mean():.1%} ({tc.churned.sum()} of {len(tc)})")
```

OUTPUT

(3000, 11)

	customer_id	senior_citizen	tenure_months	contract	internet_service	payment_method
0	T00001	0	71	Two year	Fiber	Net banking
81.28		5848.06	1	No	0	
1	T00002	1	56	Month-to-month	Fiber	Auto-debit
115.12		6482.44	0	No	0	
2	T00003	0	38	Two year	Fiber	Auto-debit
85.90		3186.34	1	No	0	

{'customer_id': 'str', 'senior_citizen': 'int64', 'tenure_months': 'int64', 'contract': 'str', 'internet_service': 'str', 'payment_method': 'str', 'monthly_charges': 'float64', 'total_charges': 'float64', 'support_calls': 'int64', 'churn': 'str', 'churned': 'int64'}

Overall churn rate: 25.7% (772 of 3000)

2. Data quality

Python

```
print("missing:", tc.isna().sum().sum(), "| duplicate ids:", tc.customer_id.duplicated().sum())
print(tc.describe().round(1).T[["min", "50%", "max", "mean"]])
for c in ["contract", "internet_service", "payment_method"]: print(c, tc[c].value_counts().to_dict())
gap = tc.total_charges - tc.tenure_months * tc.monthly_charges # consistency check between columns
print("total_charges - tenure * monthly: median", round(gap.median(), 1), "| 95% within ±", round(gap.abs().quantile(.95), 0))
print("tenure 0 customers:", (tc.tenure_months == 0).sum())
```

OUTPUT

missing: 0 | duplicate ids: 0

	min	50%	max	mean
senior_citizen	0.0	0.0	1.0	0.2
tenure_months	0.0	36.0	72.0	36.1
monthly_charges	20.0	73.8	131.7	67.1
total_charges	19.6	2161.3	8724.4	2418.3
support_calls	0.0	1.0	6.0	1.3
churned	0.0	0.0	1.0	0.3

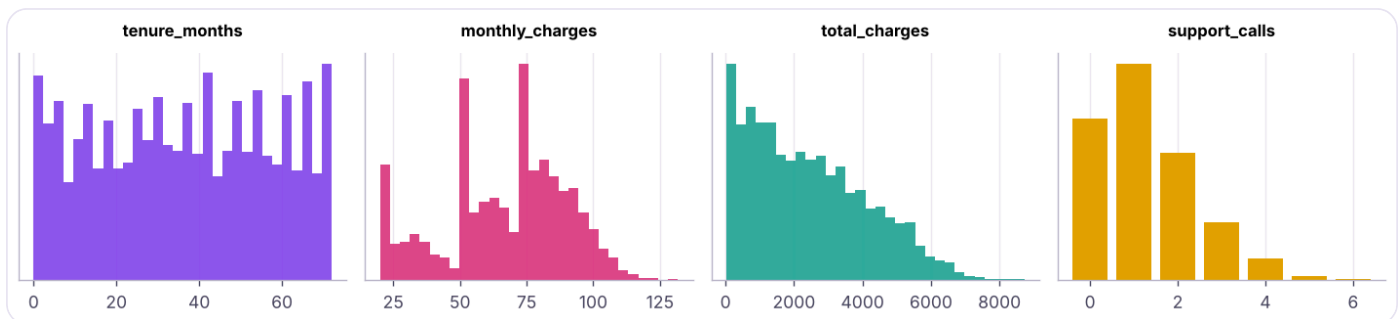
contract {'Month-to-month': 1690, 'One year': 739, 'Two year': 571}

internet_service {'Fiber': 1368, 'DSL': 1047, 'No internet': 585}


```
payment_method {'Card': 767, 'UPI': 753, 'Auto-debit': 746, 'Net banking': 734}
total_charges - tenure × monthly: median -0.8 | 95% within ± 189.0
tenure 0 customers: 45
```

No missing values or duplicate IDs. `total_charges` roughly equals `tenure × monthly charges` (small gaps from price changes), so the columns are consistent. A few customers have tenure 0 (new joiners), which is valid.

3. Univariate analysis



Distributions of the numeric variables. Tenure is spread across 0–72 months; monthly charges have a bump of low-price plans.

4. Bivariate: each driver vs churn

Python

```
def churn_by(col):
    g = tc.groupby(col, observed=True)["churned"].agg(customers="size", churn_rate="mean")
    g["churn_rate"] = (100 * g.churn_rate).round(1); g["lift"] = (g.churn_rate / (100 *
tc.churned.mean())).round(2)
    return g
for col in ["contract", "internet_service", "payment_method", "senior_citizen"]: print(churn_by(col),
"\n")
```

OUTPUT

	customers	churn_rate	lift
contract			
Month-to-month	1690	35.9	1.40
One year	739	16.9	0.66
Two year	571	7.0	0.27
internet_service			
DSL	1047	21.8	0.85
Fiber	1368	32.2	1.25
No internet	585	17.6	0.68
payment_method			
Auto-debit	746	23.5	0.91

Card	767	26.3	1.02
Net banking	734	28.5	1.11
UPI	753	24.7	0.96

	customers	churn_rate	lift
senior_citizen			
0	2490	25.3	0.98
1	510	27.8	1.08

Lift = segment churn ÷ overall churn. Lift 1.4 means 40% more likely to churn than average.

Python

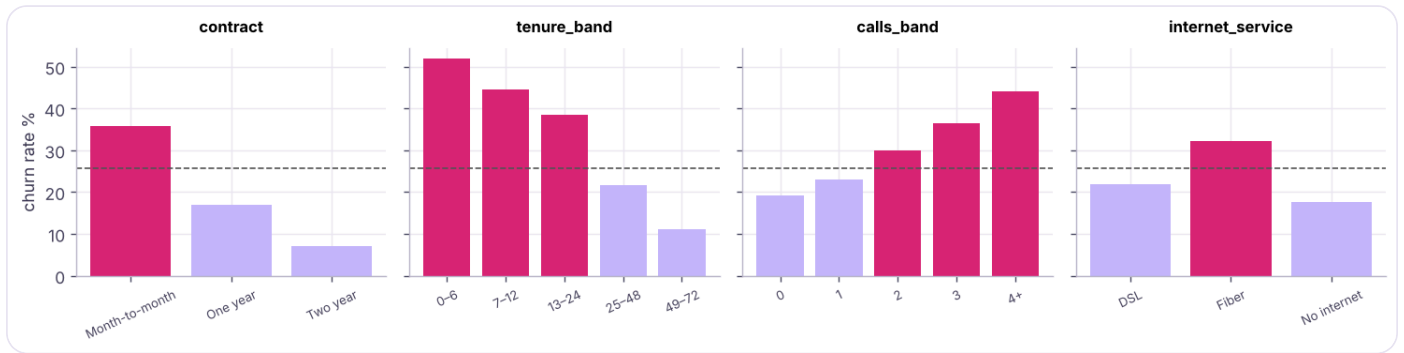
```
tc["tenure_band"] = pd.cut(tc.tenure_months, [0, 6, 12, 24, 48, 72], include_lowest=True, labels=["0-6", "7-12", "13-24", "25-48", "49-72"])
tc["calls_band"] = pd.cut(tc.support_calls, [-1, 0, 1, 2, 3, 10], labels=["0", "1", "2", "3", "4+"])
tc["charge_band"] = pd.qcut(tc.monthly_charges, 4, labels=["Q1 low", "Q2", "Q3", "Q4 high"])
for col in ["tenure_band", "calls_band", "charge_band"]: print(churn_by(col), "\n")
```

OUTPUT

	customers	churn_rate	lift
tenure_band			
0-6	318	51.9	2.02
7-12	238	44.5	1.73
13-24	452	38.5	1.50
25-48	1011	21.6	0.84
49-72	981	11.1	0.43

	customers	churn_rate	lift
calls_band			
0	822	19.1	0.74
1	1101	23.1	0.90
2	649	30.0	1.17
3	294	36.4	1.41
4+	134	44.0	1.71

	customers	churn_rate	lift
charge_band			
Q1 low	781	19.0	0.74
Q2	719	20.2	0.78
Q3	750	29.9	1.16
Q4 high	750	34.0	1.32



Churn rate by the four strongest drivers. The dashed line is the overall rate.

Numeric drivers

Python

```
print(tc.groupby("churn")[["tenure_months", "monthly_charges", "total_charges",
"support_calls"]].median())
print(tc[["tenure_months", "monthly_charges", "support_calls", "senior_citizen", "churned"]].corr()
["churned"].drop("churned").round(3).sort_values())
```

OUTPUT

	tenure_months	monthly_charges	total_charges	support_calls
churn				
No	42.0	68.010	2445.905	1.0
Yes	21.0	75.405	1357.910	1.0
tenure_months	-0.322			
senior_citizen	0.022			
monthly_charges	0.143			
support_calls	0.153			

Name: churned, dtype: float64

5. Segments: interactions matter

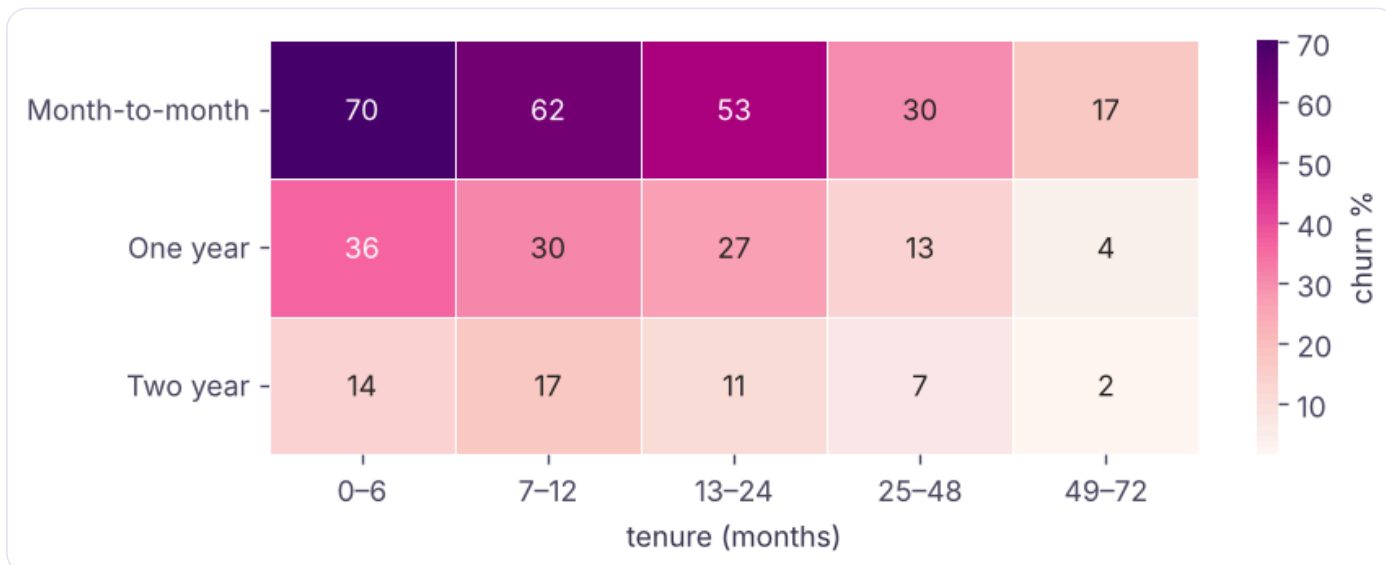
Python

```
seg = tc.pivot_table(index="contract", columns="tenure_band", values="churned", aggfunc="mean",
observed=True).mul(100).round(1)
print(seg)
cnt = tc.pivot_table(index="contract", columns="tenure_band", values="churned", aggfunc="size",
observed=True)
print(cnt)
```

OUTPUT

tenure_band	0-6	7-12	13-24	25-48	49-72
contract					
Month-to-month	70.2	62.4	53.4	29.7	17.4
One year	36.5	30.3	26.9	13.4	4.3
Two year	13.5	17.0	10.7	6.8	1.5
tenure_band	0-6	7-12	13-24	25-48	49-72

contract					
Month-to-month	181	125	249	583	552
One year	85	66	119	238	231
Two year	52	47	84	190	198



Churn rate (%) by contract and tenure: new month-to-month customers are the hot spot.

Python

```
hot = tc[(tc.contract == "Month-to-month") & (tc.tenure_months ≤ 12)]
fiber_calls = tc[(tc.internet_service == "Fiber") & (tc.support_calls ≥ 3)]
for name, d in [("Month-to-month, tenure ≤ 12", hot), ("Fiber with 3+ support calls", fiber_calls)]:
    print(f"{name:32s} customers {len(d):4d} | churn {d.churned.mean():.1%} | lift {d.churned.mean() / tc.churned.mean():.2f}")
```

OUTPUT

```
Month-to-month, tenure ≤ 12      customers  306 | churn 67.0% | lift 2.60
Fiber with 3+ support calls      customers  192 | churn 46.9% | lift 1.82
```

6. Business impact: revenue at risk

Python

```
tc["monthly_revenue_lost"] = tc.monthly_charges * tc.churned
lost = tc.monthly_revenue_lost.sum()
print(f"Monthly revenue of churned customers: {lost:,.0f} ({lost / tc.monthly_charges.sum():.1%} of all monthly revenue)")
imp = tc.groupby("contract").agg(churners=("churned", "sum"), revenue_lost=("monthly_revenue_lost", "sum"))
imp["share_of_loss_%"] = (100 * imp.revenue_lost / lost).round(1)
print(imp.round(0))
```

OUTPUT

Monthly revenue of churned customers: 56,173 (27.9% of all monthly revenue)

	churners	revenue_lost	share_of_loss_%
contract			
Month-to-month	607	43293.0	77.0
One year	125	9763.0	17.0
Two year	40	3117.0	6.0

Revenue numbers here are in the data's currency units per month; multiply by expected remaining lifetime to estimate lifetime value lost.

7. Findings and recommendations

KEY TAKEAWAYS

1. **About 1 in 4 customers churn** (25.7%), taking 27.9% of monthly revenue with them.
2. **Contract is the strongest driver:** month-to-month churn is about 36%, against about 7% on two-year contracts, and month-to-month customers account for 77% of the monthly revenue lost.
3. **The first year is critical:** churn is above 40% in the first 12 months and falls steadily with tenure.
4. **Support calls signal risk:** churn roughly doubles from 0 calls (about 19%) to 3 or more calls (about 36–44%).
5. **Fiber customers churn more** (about 32% against 22% on DSL), which is consistent with price or service-quality issues.
6. Payment method and senior status make only small differences.

Recommendations. (a) Offer month-to-month customers in their first year an incentive to move to an annual plan. (b) Trigger a proactive call after the second support ticket. (c) Investigate Fiber service quality and pricing. (d) Build a churn-score model (ML track, capstone) to rank customers each month and measure the retention campaign with an A/B test.

NOTE EDA is not causal proof

These are associations. Contract and churn are related, but customers who intend to leave may simply choose month-to-month plans. Experiments (A/B tests) or careful causal methods are needed before claiming that a change will cause lower churn.

Automated profiling tools

Libraries such as **ydata-profiling** (`ProfileReport(df)`), **sweetviz** and **D-Tale** generate an EDA report in one line, and AI assistants (Lesson ai-analytics) can draft charts. Use them for a quick first scan, then do the focused, question-driven analysis above.

Summary

KEY TAKEAWAYS

- EDA follows a checklist: question → quality → univariate → bivariate → segments → impact → findings.
- Check consistency between related columns, not just missing values.
- Compare churn rate and lift by category; band numeric variables with `cut / qcut`.
- Interactions (contract × tenure) reveal hot spots that single variables hide.
- Quantify impact in money and end with specific, owned recommendations; remember that association is not causation.

Quick quiz

6 questions · tap an answer to check it

1. What is "lift" in segment analysis?

A The number of customers in a segment

B Segment rate divided by the overall rate

C The difference between two means

D A chart type

Answer: B. Lift 2 = twice as likely as average.

2. Why band tenure into groups?

A pandas needs it

B To compare churn rates across ranges and spot non-linear patterns

C To delete outliers

D To save memory

Answer: B. `pd.cut` for fixed edges, `qcut` for equal-sized groups.

3. Which finding came from an interaction analysis?

- A Overall churn is 25.7%
- B New month-to-month customers have the highest churn**
- C Fiber churns more
- D No missing values

Answer: B. It combines two variables.

4. Why check `total_charges` against `tenure × monthly_charges`?

- A To verify consistency between related columns**
- B To compute churn
- C To fill missing values
- D To make a chart

Answer: A. Inconsistent columns hint at data errors.

5. What should an EDA conclude with?

- A All charts produced
- B Numbered findings with quantified impact and recommendations**
- C The raw data
- D A model

Answer: B. Stakeholders act on findings, not charts.

6. Can EDA alone prove that a two-year contract reduces churn?

A Yes

B No, it shows association; experiments or causal methods are needed

C Only with a heatmap

D Only if $r > 0.5$

Answer: B. Self-selection can explain the pattern.

EXERCISE Extend the case study

(1) Compute churn by payment method within month-to-month customers only. (2) Find the 10 highest-revenue customers at risk (month-to-month, tenure ≤ 12 , 2+ support calls). (3) Plot churn rate against monthly charge deciles. (4) Write a five-bullet executive summary.

Show solution (worked answer)

Python

```
print(tc[tc.contract == "Month-to-month"].groupby("payment_method").churned.mean().round(3))
risk = tc[(tc.contract == "Month-to-month") & (tc.tenure_months ≤ 12) & (tc.support_calls ≥ 2) &
(tc.churned == 0)]
print(risk.nlargest(10, "monthly_charges")[["customer_id", "monthly_charges", "support_calls"]])
dec = tc.groupby(pd.qcut(tc.monthly_charges, 10), observed=True).churned.mean()
dec.plot(marker="o")
```

INTERVIEW CORNER Questions you may be asked

Q: Walk me through how you would do EDA on a new dataset. Clarify the question and target, check quality (types, missing, duplicates, ranges, consistency), look at each variable, compare each variable against the target, explore interactions and segments, quantify impact, and summarise findings with next steps.

Q: How do you decide which variables matter most for churn? Compare churn rates/lift by category, correlations and group medians for numeric variables, then confirm with a model (feature importance) and, for causality, with experiments.

Q: Which chart would you use to show churn by two categorical variables? A heatmap of churn rates (with counts available) or grouped bars.

Python and SQL: sqlite3, DuckDB, SQLAlchemy and pandas.read_sql

Connect Python to databases. Create and query SQLite with the standard library, use parameterised queries safely, load DataFrames to tables and back, query CSV and DataFrames directly with DuckDB, connect to MySQL, PostgreSQL and SQL Server with SQLAlchemy, and decide what to do in SQL versus pandas.

● Python class 18

✦ DuckDB 1.x/2.x · SQLAlchemy 2

Intermediate

🕒 100 min

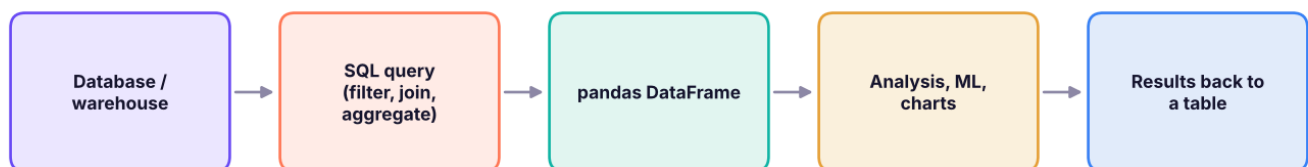
By the end of this lesson you can

- Create a SQLite database, run DDL/DML from Python and fetch results
- Write parameterised queries and explain SQL injection
- Move data between pandas and databases with `to_sql` and `read_sql`
- Query files and DataFrames with DuckDB
- Connect to server databases with SQLAlchemy connection URLs and secrets kept out of code

Why connect Python to SQL?

Company data lives in databases and warehouses (MySQL, PostgreSQL, SQL Server, Snowflake, BigQuery, Databricks). The usual pattern: **filter and aggregate in SQL** (close to the data), then **analyse, model and chart in Python**.

Python ↔ SQL workflow



SQL pulls and summarises; Python analyses, models and presents.

sqlite3: a database in a file

SQLite ships with Python. It is perfect for learning, prototypes and small apps.

Python

```
import sqlite3, pandas as pd
con = sqlite3.connect(":memory:") # or "shop.db" for a file
cur = con.cursor()
cur.executescript("""
CREATE TABLE customers (id INTEGER PRIMARY KEY, name TEXT NOT NULL, city TEXT);
CREATE TABLE orders (id INTEGER PRIMARY KEY, customer_id INTEGER REFERENCES customers(id), amount REAL,
order_date TEXT);
""")
cur.executemany("INSERT INTO customers (name, city) VALUES (?, ?)", [("Asha", "Pune"), ("Ravi",
"Delhi"), ("Meera", "Kochi")])
cur.executemany("INSERT INTO orders (customer_id, amount, order_date) VALUES (?, ?, ?)",
[(1, 1200, "2025-01-05"), (1, 450, "2025-02-11"), (2, 3000, "2025-02-14"), (3, 780,
"2025-03-01")])
con.commit()
cur.execute("SELECT c.name, COUNT(*), SUM(o.amount) FROM orders o JOIN customers c ON c.id =
o.customer_id GROUP BY c.name ORDER BY 3 DESC")
print(cur.fetchall())
cur.execute("SELECT * FROM customers WHERE city = ?", ("Pune",)); print(cur.fetchone())
```

OUTPUT

```
[('Ravi', 1, 3000.0), ('Asha', 2, 1650.0), ('Meera', 1, 780.0)]
(1, 'Asha', 'Pune')
```

Method	Use
<code>con.execute(sql, params)</code> / <code>cursor.execute</code>	Run one statement
<code>executemany(sql, rows)</code>	Insert many rows
<code>fetchone()</code> , <code>fetchall()</code> , <code>fetchmany(n)</code>	Read results
<code>con.commit()</code> / <code>con.rollback()</code>	Save / undo a transaction
<code>with con:</code>	Commits on success, rolls back on error

Parameterised queries and SQL injection

Python

```
user_input = "Pune' OR '1'='1"
unsafe = f"SELECT name FROM customers WHERE city = '{user_input}'" # NEVER build SQL with f-strings
print("unsafe returns:", con.execute(unsafe).fetchall())
print("safe returns: ", con.execute("SELECT name FROM customers WHERE city = ?",
(user_input,)).fetchall())
```

OUTPUT

```
unsafe returns: [('Asha',), ('Ravi',), ('Meera',)]
safe returns:  []
```

The unsafe query returned **every** customer because the input changed the SQL. Placeholders (`?` in SQLite, `%s` or `:name` in other drivers) send values separately, so input can never become code.

Transactions

Python

```
try:
    with con:
        con.execute("UPDATE orders SET amount = amount - 500 WHERE id = 1")
        con.execute("INSERT INTO orders (customer_id, amount) VALUES (2, 500)")
    change
    raise ValueError("payment gateway down")
except ValueError as e:
    print("rolled back:", e)
print(con.execute("SELECT id, amount FROM orders WHERE id = 1").fetchone(), con.execute("SELECT COUNT(*) FROM orders").fetchone())
```

OUTPUT

```
rolled back: payment gateway down
(1, 1200.0) (4,)
```

Both changes were undone together: that is atomicity (the A in ACID).

pandas ↔ database

Python

```
s = pd.read_csv("superstore_sales.csv")
s.to_sql("sales", con, index=False, if_exists="replace")
q = """
SELECT region, category, ROUND(SUM(sales) / 1e6, 2) AS sales_m, ROUND(100.0 * SUM(profit) / SUM(sales),
1) AS margin_pct
FROM sales WHERE discount ≤ :max_disc GROUP BY region, category ORDER BY sales_m DESC LIMIT 5
"""
print(pd.read_sql(q, con, params={"max_disc": 0.1}))
con.close()
```

OUTPUT

	region	category	sales_m	margin_pct
0	South	Technology	32.14	15.9
1	West	Technology	30.21	16.1
2	North	Technology	29.93	16.6
3	East	Technology	26.57	17.5
4	West	Furniture	5.36	10.3

For large tables, read in pieces with `chunksize=` or push the aggregation into SQL so only the summary travels.

DuckDB: SQL on files and DataFrames

DuckDB is an in-process analytical database: no server, very fast on columnar data, and it can query CSV, Parquet and pandas DataFrames directly.

Python

```
import duckdb
print(duckdb.sql("""
    SELECT category, COUNT(*) AS lines, ROUND(SUM(sales)/1e6, 2) AS sales_m
    FROM 'superstore_sales.csv' GROUP BY category ORDER BY sales_m DESC
""").df())
customers = pd.read_csv("customers.csv")
top = duckdb.sql("""
    SELECT c.state, COUNT(DISTINCT s.customer_id) AS customers, ROUND(SUM(s.sales)/1e6, 2) AS sales_m,
           RANK() OVER (ORDER BY SUM(s.sales) DESC) AS rnk
    FROM s JOIN customers c USING (customer_id)           -- s and customers are pandas
                                                         DataFrames!
    GROUP BY c.state QUALIFY rnk ≤ 5 ORDER BY rnk
""").df()
print(top, duckdb.__version__)
```

OUTPUT

	category	lines	sales_m
0	Technology	1892	156.44
1	Furniture	1917	34.84
2	Office Supplies	2514	10.13

	state	customers	sales_m	rnk
0	Maharashtra	93	25.67	1
1	Delhi	54	15.66	2
2	Kerala	48	14.71	3
3	Madhya Pradesh	57	14.41	4
4	West Bengal	54	13.89	5

2.0.0

DuckDB also writes Parquet (`COPY (SELECT ...) TO 'out.parquet'`), reads from S3, and powers tools such as MotherDuck and the Metabase/MinusX demos in the AI analytics track.

Server databases with SQLAlchemy

SQLAlchemy gives one interface to many databases. You need the database's driver package too.

Python

```
from sqlalchemy import create_engine, text
import os, pandas as pd

# Connection URLs: dialect+driver://user:password@host:port/database
url = os.environ["DB_URL"] # e.g. "postgresql+psycopg://analyst:**@db.company.com:5432/sales"
# MySQL: "mysql+pymysql://user:pw@host:3306/db"
# SQL Server: "mssql+pyodbc://user:pw@server/db?driver=ODBC+Driver+18+for+SQL+Server"
# SQLite: "sqlite:///shop.db"
engine = create_engine(url)

with engine.connect() as conn:
    df = pd.read_sql(text("SELECT * FROM orders WHERE order_date ≥ :d"), conn, params={"d": "2025-01-01"})

df_summary.to_sql("daily_summary", engine, if_exists="append", index=False, chunksize=10_000)
```

Warehouses have their own connectors (`snowflake-connector-python` , `google-cloud-bigquery` , `databricks-sql-connector`), and most also accept SQLAlchemy URLs.

WATCH OUT Keep credentials out of code

Read passwords from environment variables, a `.env` file (with `python-dotenv`) that is in `.gitignore` , or a secrets manager. Use a read-only database user for analysis.

SQL or pandas?

Do it in SQL when...	Do it in pandas when...
The data is large and lives in the database	The data fits in memory
Filtering, joining, aggregating	Complex reshaping, statistics, ML, charts
Many people need the same logic (views)	Exploratory, one-off analysis
You want the database's indexes and compute	You need Python libraries

Summary

KEY TAKEAWAYS

- `sqlite3` creates databases in a file or memory; `execute`, `executemany`, `fetchall`, `commit`; `with con:` wraps a transaction.
- Always use parameter placeholders, never f-strings, to prevent SQL injection.
- `df.to_sql` writes tables; `pd.read_sql(query, con, params=...)` reads results into DataFrames.
- DuckDB runs fast SQL directly on CSV/Parquet files and pandas DataFrames.
- SQLAlchemy connects to MySQL, PostgreSQL, SQL Server and warehouses with a URL; keep credentials in environment variables.

Quick quiz

6 questions · tap an answer to check it

1. How should user input be passed into a SQL query?

- A** With an f-string
- B** As parameters with placeholders (`?` , `%s` , `:name`)
- C** With string concatenation
- D** In a comment

Answer: B. Placeholders prevent SQL injection.

2. What does `with con:` do in `sqlite3`?

- A** Opens a file
- B** Commits if the block succeeds and rolls back on an exception
- C** Closes the database
- D** Creates a table

Answer: B. It makes the block a transaction.

3. Which writes a DataFrame to a database table?

A `pd.read_sql`

B `df.to_sql("table", con)`

C `df.to_csv`

D `con.fetchall`

Answer: B. `if_exists` controls replace/append.

4. What is special about DuckDB?

A It requires a server

B It runs in-process and can query CSV, Parquet and DataFrames directly with SQL

C It only stores text

D It replaces pandas entirely

Answer: B. It is optimised for analytics.

5. Where should a database password be stored?

A In the notebook

B In an environment variable or secrets manager

C In the SQL query

D On GitHub

Answer: B. Never commit credentials.

6. Why aggregate in SQL before loading into pandas?

- A SQL is always faster to write
- B Only the small summary travels over the network and into memory**
- C pandas cannot aggregate
- D It is required

Answer: B. Push heavy work to the database.

EXERCISE Practice

(1) Load `customers.csv` and `superstore_sales.csv` into SQLite. (2) In SQL, find the top 5 cities by profit and read the result into pandas. (3) Repeat with DuckDB directly on the CSV files. (4) Write a function `get_sales(con, region)` that uses a parameterised query.

Show solution (worked answer)

Python

```
con = sqlite3.connect(":memory:")
pd.read_csv("customers.csv").to_sql("customers", con, index=False)
pd.read_csv("superstore_sales.csv").to_sql("sales", con, index=False)
print(pd.read_sql("SELECT city, SUM(profit) p FROM sales GROUP BY city ORDER BY p DESC LIMIT 5",
con))
print(duckdb.sql("SELECT city, SUM(profit) p FROM 'superstore_sales.csv' GROUP BY city ORDER BY p
DESC LIMIT 5").df())
def get_sales(con, region):
    return pd.read_sql("SELECT SUM(sales) AS sales FROM sales WHERE region = ?", con, params=
(region,))
print(get_sales(con, "West"))
```

INTERVIEW CORNER Questions you may be asked

Q: What is SQL injection and how do you prevent it? Malicious input that changes a query's meaning; prevent it with parameterised queries (and least-privilege accounts, input validation, ORMs).

Q: How do you load a 50 GB table for analysis? Aggregate or sample in the database, read in chunks, or use a columnar engine (DuckDB, Spark, the warehouse itself) instead of pulling everything into pandas.

Q: What does ACID mean? Atomicity, Consistency, Isolation, Durability: transactions are all-or-nothing, keep rules valid, do not interfere, and survive crashes.

APIs, JSON and Web Scraping with requests and BeautifulSoup

Collect data from the web. Understand HTTP, call REST APIs with requests (GET/POST, query parameters, headers, authentication, pagination, retries and rate limits), flatten nested JSON with `json_normalize`, and scrape HTML tables and pages responsibly with BeautifulSoup and `pandas.read_html`.

● Python class 18

✦ requests 2.3x · httpx · BeautifulSoup 4

Intermediate

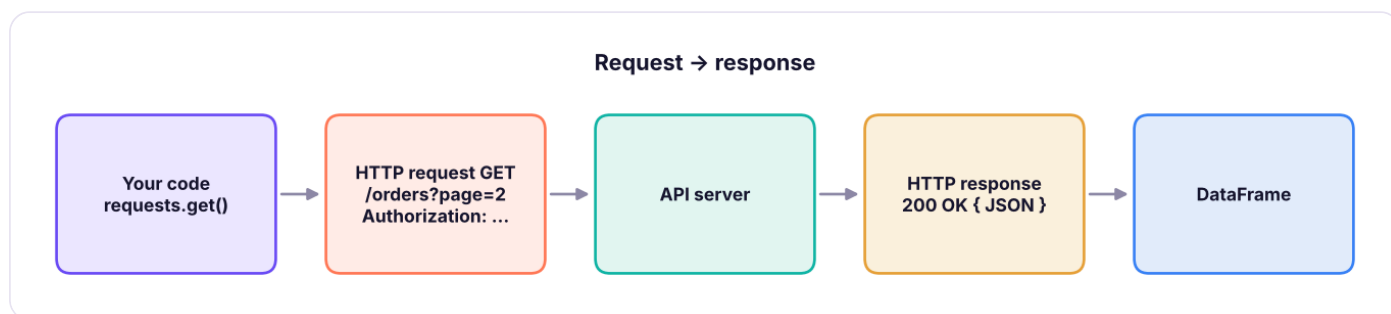
🕒 100 min

By the end of this lesson you can

- Explain HTTP methods, status codes, headers and JSON responses
- Call REST APIs with parameters, headers and API keys stored safely
- Handle pagination, timeouts, retries and rate limits
- Flatten nested JSON into DataFrames with `json_normalize`
- Scrape HTML with BeautifulSoup and `read_html`, respecting robots.txt and terms of use

HTTP in five minutes

An **API** (Application Programming Interface) lets programs ask another system for data. Web APIs use **HTTP**: the client sends a **request** (method, URL, headers, optional body) and receives a **response** (status code, headers, body, usually JSON).



A REST API call.

Method	Meaning	Status code	Meaning
GET	Read	200	OK
POST	Create / send data	201	Created
PUT / PATCH	Replace / update	400	Bad request
DELETE	Remove	401 / 403	Not authenticated / not allowed
		404	Not found
		429	Too many requests (rate limit)
		500 – 503	Server error: retry later

A practice API

To keep this lesson runnable offline, we start a tiny local API that behaves like a real one (paginated orders, an API key, a rate limit). Every call below works the same way against real services.

Python

```
import json, threading, time
from http.server import BaseHTTPRequestHandler, HTTPServer
from urllib.parse import urlparse, parse_qs
ORDERS = [{"id": i, "customer": {"name": n, "city": c}, "items": [{"sku": f"P{i}", "qty": i % 3 + 1,
"price": 100.0 * i}], "status": s}
            for i, (n, c, s) in enumerate([("Asha", "Pune", "shipped"), ("Ravi", "Delhi", "new"),
("Meera", "Kochi", "shipped"), ("Kabir", "Goa", "cancelled"), ("Zoya", "Pune", "new")], 1)]
HITS = {"n": 0}
class Api(BaseHTTPRequestHandler):
    def log_message(self, *a): pass
    def send(self, code, body, ctype="application/json"):
        self.send_response(code); self.send_header("Content-Type", ctype); self.end_headers();
self.wfile.write(body.encode())
    def do_GET(self):
        u = urlparse(self.path); q = parse_qs(u.query)
        if u.path == "/page": return self.send(200, "<html><body><h1>Prices</h1><table id='p'><tr>
<th>Product</th><th>Price</th></tr><tr><td>Chair</td><td>₹4,500</td></tr><tr><td>Desk</td>
<td>₹12,000</td></tr></table><a href='/about'>About</a><p class='note'>Updated daily</p></body>
</html>", "text/html; charset=utf-8")
        if self.headers.get("X-API-Key") != "demo-key": return self.send(401, json.dumps({"error":
"missing or bad API key"}))
        HITS["n"] += 1
        if u.path == "/flaky" and HITS["n"] % 2: return self.send(503, json.dumps({"error": "busy"}))
        if u.path in ("/orders", "/flaky"):
            page, size = int(q.get("page", [1])[0]), int(q.get("size", [2])[0]); st = q.get("status",
[None])[0]
            rows = [o for o in ORDERS if not st or o["status"] == st]
            return self.send(200, json.dumps({"data": rows[(page - 1) * size: page * size], "page":
page, "next": page + 1 if page * size < len(rows) else None}))
            self.send(404, json.dumps({"error": "not found"}))
    def do_POST(self):
        body = json.loads(self.rfile.read(int(self.headers["Content-Length"])))
        self.send(201, json.dumps({"created": True, "echo": body}))
srv = HTTPServer(("127.0.0.1", 0), Api); BASE = f"http://127.0.0.1:{srv.server_port}"
threading.Thread(target=srv.serve_forever, daemon=True).start()
print("practice API running on this machine at", BASE.rsplit(":", 1)[0])
```

OUTPUT

practice API running on this machine at http://127.0.0.1

GET requests with requests

Python

```
import requests, os
os.environ.setdefault("DEMO_API_KEY", "demo-key") # in real life, set this outside the code
headers = {"X-API-Key": os.environ["DEMO_API_KEY"], "Accept": "application/json"}
r = requests.get(f"{BASE}/orders", params={"page": 1, "size": 2}, headers=headers, timeout=10)
print(r.status_code, r.url.replace(BASE, "BASE"), r.headers["Content-Type"])
payload = r.json()
print(payload["page"], payload["next"], payload["data"][0]["customer"])
bad = requests.get(f"{BASE}/orders", timeout=10)
print(bad.status_code, bad.json())
try: bad.raise_for_status()
except requests.HTTPError as e: print("HTTPError:", str(e)[:40])
```

OUTPUT

```
200 BASE/orders?page=1&size=2 application/json
1 2 {'name': 'Asha', 'city': 'Pune'}
401 {'error': 'missing or bad API key'}
HTTPError: 401 Client Error: Unauthorized for url:
```

Always set a `timeout`, check the status (`raise_for_status()`), and keep keys in environment variables.

POST requests

Python

```
r = requests.post(f"{BASE}/orders", json={"customer": "Asha", "items": [{"sku": "P9", "qty": 1}]},
headers=headers, timeout=10)
print(r.status_code, r.json())
```

OUTPUT

```
201 {'created': True, 'echo': {'customer': 'Asha', 'items': [{'sku': 'P9', 'qty': 1}]}}
```

`json=` serialises the body and sets the `Content-Type` header for you. Use `data=` for form posts and `files=` for uploads.

Pagination, retries and rate limits

Python

```
def fetch_all(path, params=None, max_pages=50):
    rows, page = [], 1
    with requests.Session() as session:
        session.headers.update(headers)
        while page and page ≤ max_pages:
            resp = session.get(f"{BASE}/{path}", params={**(params or {}), "page": page, "size": 2},
                               timeout=10)
            resp.raise_for_status()
            body = resp.json(); rows += body["data"]; page = body["next"]
    return rows
orders = fetch_all("/orders"); print(len(orders), "orders fetched across pages")

def get_with_retry(url, tries=4, backoff=0.2, **kw):
    for attempt in range(1, tries + 1):
        resp = requests.get(url, timeout=10, **kw)
        if resp.status_code not in (429, 500, 502, 503, 504): return resp
        wait = float(resp.headers.get("Retry-After", backoff * 2 ** (attempt - 1)))
        print(f" attempt {attempt}: {resp.status_code}, retrying in {wait:.1f}s"); time.sleep(wait)
    resp.raise_for_status()
print(get_with_retry(f"{BASE}/flaky", headers=headers).status_code)
```

OUTPUT

```
5 orders fetched across pages
  attempt 1: 503, retrying in 0.2s
200
```

Respect rate limits: read the API's documented limits, honour `Retry-After`, cache results, and sleep between calls.

Flattening nested JSON

Python

```
import pandas as pd
pd.set_option("display.width", 160)
flat = pd.json_normalize(orders, sep="_")
print(flat.drop(columns="items"))
lines = pd.json_normalize(orders, record_path="items", meta=["id", "status", ["customer", "city"]],
                           sep="_")
lines["value"] = lines.qty * lines.price
print(lines)
```

OUTPUT

	id	status	customer_name	customer_city
0	1	shipped	Asha	Pune
1	2	new	Ravi	Delhi
2	3	shipped	Meera	Kochi

3	4	cancelled			Kabir	Goa	
4	5	new			Zoya	Pune	
	sku	qty	price	id	status	customer_city	value
0	P1	2	100.0	1	shipped	Pune	200.0
1	P2	3	200.0	2	new	Delhi	600.0
2	P3	1	300.0	3	shipped	Kochi	300.0
3	P4	2	400.0	4	cancelled	Goa	800.0
4	P5	3	500.0	5	new	Pune	1500.0

`record_path` explodes a nested list into rows; `meta` copies parent fields onto each row.

API	Data	Notes
data.gov.in	Indian government open data	Free API key
RBI DBIE, World Bank, IMF	Economic indicators	<code>wbgapi</code> package for the World Bank
Open-Meteo	Weather forecasts and history	No key
GitHub REST API	Repositories, issues	Token raises limits
Alpha Vantage, yfinance	Stock prices	Free tiers, rate-limited
OpenAI, Anthropic, Gemini	LLMs	Paid keys, see GenAI track

Python

```
# Example (needs internet): daily max temperature in Pune from Open-Meteo
r = requests.get("https://api.open-meteo.com/v1/forecast",
                 params={"latitude": 18.52, "longitude": 73.86, "daily": "temperature_2m_max",
                        "timezone": "Asia/Kolkata"}, timeout=10)
weather = pd.DataFrame(r.json()["daily"])
```

Copyright © 2015 Pearson Education, Inc. or its affiliate(s). All rights reserved.

When there is no API, you can parse HTML. **BeautifulSoup** navigates the HTML tree; `pandas.read_html` grabs `<table>` elements directly.

Python

```
from bs4 import BeautifulSoup
from io import StringIO
html = requests.get(f"{BASE}/page", timeout=10).text
soup = BeautifulSoup(html, "html.parser")
print(soup.h1.text, "|", soup.find("p", class_="note").get_text(strip=True), "|", [a["href"] for a in
soup.find_all("a")])
rows = [[td.get_text(strip=True) for td in tr.find_all(["td", "th"])] for tr in soup.select("table#p
tr")]
print(rows)
t = pd.read_html(StringIO(html))[0]
t["Price"] = t["Price"].str.replace(r"₹,", "", regex=True).astype(int)
print(t, t.Price.sum())
srv.shutdown()
```

OUTPUT

```
Prices | Updated daily | ['/about']
[['Product', 'Price'], ['Chair', '₹4,500'], ['Desk', '₹12,000']]
   Product  Price
0   Chair    4500
1    Desk   12000
```

BeautifulSoup	Finds
<code>soup.find("tag", class_="x")</code>	First match
<code>soup.find_all("a")</code>	All matches
<code>soup.select("div.card > h2")</code>	CSS selectors
<code>el.get_text(strip=True)</code> , <code>el["href"]</code>	Text and attributes

For pages built by JavaScript, use **Playwright** or Selenium to drive a real browser, or look in the browser's Network tab for the underlying JSON API.

WATCH OUT Scrape responsibly and legally

Check the site's **terms of use** and **robots.txt**, prefer official APIs, identify your scraper with a User-Agent, go slowly (one request every few seconds), cache pages, and never collect personal data without a lawful basis (India's DPDP Act 2023, GDPR). Logged-in or paywalled content is usually off-limits.

Summary

KEY TAKEAWAYS

- HTTP: methods (GET, POST...), status codes (200, 401, 404, 429, 5xx), headers and JSON bodies.
- `requests.get(url, params=, headers=, timeout=) ; r.json() ; r.raise_for_status() ;` keys from environment variables.
- Loop over pages with a `Session` ; retry 429/5xx with exponential backoff and honour `Retry-After` .
- `pd.json_normalize(data, record_path=, meta=)` flattens nested JSON.
- Scrape with BeautifulSoup or `pd.read_html` only when allowed; prefer APIs and be polite.

Quick quiz

6 questions · tap an answer to check it

1. What does status code 429 mean?

- A** Not found
- B** Too many requests (rate limit)
- C** Server error
- D** Success

Answer: B. Slow down and retry after the indicated time.

2. Why set `timeout=` on every request?

- A** It speeds up requests
- B** Without it, a request can hang forever
- C** It is required by APIs
- D** To retry automatically

Answer: B. Typical values are 5–30 seconds.

3. Where should an API key be kept?

- A In the URL in your notebook
- B In an environment variable or secrets store
- C In a GitHub README
- D In a comment

Answer: B. Read it with `os.getenv` .

4. What does `json_normalize(..., record_path="items")` do?

- A Deletes the items field
- B Creates one row per element of each record's items list
- C Converts to CSV
- D Sorts the data

Answer: B. `meta` keeps parent fields.

5. Which is the best first option to get data from a website?

- A An official API or data download
- B Scraping every page quickly
- C Copy-paste
- D A screenshot

Answer: A. Scraping is a fallback.

6. Which tool can read every `<table>` on a page into DataFrames?

A `requests.post`

B `pd.read_html`

C `json.loads`

D `soup.h1`

Answer: B. It returns a list of DataFrames.

EXERCISE Practice

(1) Using the Open-Meteo example (on your machine), fetch 7 days of forecasts for three cities and combine them into one DataFrame. (2) Write `fetch_all` for an API that uses `offset / limit` instead of pages. (3) Scrape the list of tables from a Wikipedia page of your choice with `read_html` and clean one of them.

Show solution (worked answer)

Python

```
cities = {"Pune": (18.52, 73.86), "Delhi": (28.61, 77.21), "Kochi": (9.93, 76.27)}
frames = []
for name, (lat, lon) in cities.items():
    r = requests.get("https://api.open-meteo.com/v1/forecast", params={"latitude": lat,
"longitude": lon, "daily": "temperature_2m_max", "timezone": "Asia/Kolkata"}, timeout=10)
    frames.append(pd.DataFrame(r.json()["daily"]).assign(city=name))
df = pd.concat(frames)
def fetch_offset(url, limit=100):
    out, offset = [], 0
    while True:
        batch = requests.get(url, params={"offset": offset, "limit": limit}, timeout=10).json()
        if not batch: return out
        out += batch; offset += limit
```

INTERVIEW CORNER**Questions you may be asked**

Q: How do you collect data from a paginated API reliably? Use a session, loop until there is no next page/cursor, set timeouts, retry 429/5xx with backoff, respect rate limits, log progress, and make the job restartable.

Q: API vs web scraping? APIs are stable, documented and permitted; scraping depends on page structure, can break, and has legal and ethical limits.

Q: How do you turn nested JSON into a table? `pd.json_normalize` with `record_path` and `meta`, or explode list columns and expand dict columns.

Performance and Bigger Data: Vectorisation, Memory, Parquet, Polars and DuckDB

Make Python data work fast and scale beyond memory. Measure with `timeit`, replace loops with vectorised code, shrink DataFrames with the right dtypes, store data as Parquet, use Polars lazy queries and DuckDB for out-of-core analytics, and know when to move to Spark and the cloud.

● Python class 19

✦ pandas 3 · Polars 1.x/2.x · DuckDB · PyArrow

Intermediate

🕒 100 min

By the end of this lesson you can

- Measure speed and memory before optimising
- Replace row loops and apply with vectorised pandas and NumPy operations
- Reduce memory with categories, smaller numeric types and selected columns
- Store and read data efficiently with Parquet and explain columnar formats
- Write equivalent queries in pandas, Polars and DuckDB and choose between them

Measure first

"Premature optimisation is the root of all evil" (Donald Knuth). Profile, find the slow part, fix that.

Python

```
import pandas as pd, numpy as np, time, timeit
s = pd.read_csv("superstore_sales.csv", parse_dates=["order_date"])
big = pd.concat([s] * 20, ignore_index=True)  # ~126k rows to make timings visible
print(big.shape)
def t(fn, n=3): return min(timeit.repeat(fn, number=1, repeat=n)) * 1000
loop = lambda: [r.sales * (1 - r.discount) for r in big.itertuples()]
apply_ = lambda: big.apply(lambda r: r["sales"] * (1 - r["discount"]), axis=1)
vec = lambda: big["sales"] * (1 - big["discount"])
print(f"itertuples loop {t(loop, 1):8.1f} ms\napply(axis=1) {t(apply_, 1):8.1f} ms\nvectorised {t(vec):8.2f} ms")
```

OUTPUT

```
(126460, 19)
itertuples loop    1070.5 ms
apply(axis=1)      659.5 ms
vectorised         0.65 ms
```

The vectorised version is more than a thousand times faster. `itertuples` and `apply(axis=1)` take about the same time because both are Python loops (`apply` with `axis=1` is a loop in disguise), and `iterrows` is slower still.

Vectorising conditional logic

Python

```
def band_loop(d): return "high" if d > 0.2 else "mid" if d > 0 else "none"
f1 = lambda: big["discount"].map(band_loop)
f2 = lambda: np.select([big.discount > 0.2, big.discount > 0], ["high", "mid"], default="none")
f3 = lambda: pd.cut(big.discount, [-1, 0, 0.2, 1], labels=["none", "mid", "high"])
print(f"map(func) {t(f1):.1f} ms | np.select {t(f2):.1f} ms | pd.cut {t(f3):.1f} ms")
print((f1().values == f2()).all())
```

OUTPUT

map(func) 15.6 ms | np.select 1.0 ms | pd.cut 1.5 ms
True

Slow pattern	Faster pattern
<code>for / iterrows</code> over rows	Column arithmetic, <code>np.where</code> , <code>np.select</code>
<code>apply(axis=1)</code>	Vectorised expression, or <code>groupby().transform</code>
Growing a DataFrame in a loop (<code>concat</code> each time)	Collect a list of dicts, build once
String ops in Python loops	<code>.str</code> methods
Repeated <code>df[df.x == k]</code> in a loop	One <code>groupby</code>
Reading a big CSV many times	Read once, save as Parquet

Memory

Python

```
def mb(df): return df.memory_usage(deep=True).sum() / 1e6
print(f"original: {mb(big):.1f} MB")
small = big.copy()
for c in ["region", "segment", "category", "sub_category", "ship_mode", "state", "city"]: small[c] =
small[c].astype("category")
small["quantity"] = pd.to_numeric(small["quantity"], downcast="integer")
small[["sales", "profit", "discount"]] = small[["sales", "profit", "discount"]].astype("float32")
print(f"categories + downcast: {mb(small):.1f} MB")
print(small.dtypes.astype(str).value_counts().to_dict())
```

OUTPUT

original: 34.7 MB
categories + downcast: 18.3 MB
{'category': 7, 'str': 6, 'float32': 3, 'int64': 1, 'datetime64[us]': 1, 'int8': 1}

Categories store each label once plus small integer codes: ideal for columns with few distinct values. Also read only needed columns (`usecols=`) and stream with `chunksize=` .

Python

```
total = 0.0
for chunk in pd.read_csv("superstore_sales.csv", usecols=["region", "sales"], chunksize=2000):
    total += chunk.loc[chunk.region == "West", "sales"].sum()
print(f"West sales via chunks: {total:,.0f}")
```

OUTPUT

West sales via chunks: 51,958,520

Parquet: the analytics file format

CSV vs Parquet

CSV

- Text, row by row
- No types: everything parsed on read
- Large, slow to read
- Readable in any editor

Parquet

- Binary, column by column
- Types and schema stored
- Compressed; read only the columns you need
- Standard for data lakes (Spark, DuckDB, Polars, Snowflake)

CSV is row-oriented text; Parquet stores each column together, typed and compressed.

Python

```
import os
big.to_csv("big.csv", index=False); big.to_parquet("big.parquet", index=False)
print(f"CSV {os.path.getsize('big.csv')/1e6:.1f} MB | Parquet {os.path.getsize('big.parquet')/1e6:.1f} MB")
print(f"read CSV {t(lambda: pd.read_csv('big.csv')):.0f} ms | read Parquet {t(lambda: pd.read_parquet('big.parquet')):.0f} ms | two columns {t(lambda: pd.read_parquet('big.parquet', columns=['region', 'sales'])):.0f} ms")
```

OUTPUT

CSV 21.7 MB | Parquet 1.9 MB

read CSV 249 ms | read Parquet 15 ms | two columns 5 ms

Polars: fast DataFrames with lazy queries

Polars is a DataFrame library written in Rust. It uses all CPU cores, the Arrow memory format, and a **lazy** mode that optimises the whole query before running it.

Python

```
import polars as pl
lazy = (pl.scan_parquet("big.parquet")
        .filter(pl.col("discount") <= 0.1)
        .group_by("region", "category")
        .agg(sales=pl.col("sales").sum(), profit=pl.col("profit").sum(), lines=pl.len())
        .with_columns(margin=(100 * pl.col("profit") / pl.col("sales")).round(1), sales_m=
(pl.col("sales") / 1e6).round(2))
        .drop("profit")
        .sort("sales", descending=True))
print(lazy.head(4).collect().drop("sales"))
pandas_way = lambda: (big[big.discount <= 0.1].groupby(["region", "category"]).agg(sales=("sales",
"sum"), profit=("profit", "sum")))
print(f"pandas {t(pandas_way):.1f} ms | polars lazy {t(lambda: lazy.collect()):.1f} ms | polars
{pl.__version__}")
```

OUTPUT

shape: (4, 5)

region	category	lines	margin	sales_m
---	---	---	---	---
str	str	u32	f64	f64
South	Technology	7220	15.9	642.79
West	Technology	7080	16.1	604.14
North	Technology	6720	16.6	598.63
East	Technology	6320	17.5	531.33

pandas 22.0 ms | polars lazy 3.9 ms | polars 2.1.0

pandas	Polars
df[df.x > 1]	df.filter(pl.col("x") > 1)
df.assign(y=df.x * 2)	df.with_columns(y=pl.col("x") * 2)
df.groupby("g").agg(s=("x", "sum"))	df.group_by("g").agg(s=pl.col("x").sum())
df.sort_values("x")	df.sort("x")
pd.read_csv	pl.read_csv / lazy pl.scan_csv
df.to_pandas() ↔ pl.from_pandas(df)	Convert when a library needs pandas

DuckDB on files larger than memory

Python

```
import duckdb
q = """
SELECT region, category, ROUND(SUM(sales)/1e6, 2) AS sales_m, ROUND(100*SUM(profit)/SUM(sales), 1) AS
margin
FROM 'big.parquet' WHERE discount ≤ 0.1 GROUP BY ALL ORDER BY sales_m DESC LIMIT 4
"""
print(duckdb.sql(q).df())
print(f"duckdb {t(lambda: duckdb.sql(q).fetchall()):.1f} ms")
for f in ("big.csv", "big.parquet"): os.remove(f)
```

OUTPUT

	region	category	sales_m	margin
0	South	Technology	642.79	15.9
1	West	Technology	604.14	16.1
2	North	Technology	598.63	16.6
3	East	Technology	531.33	17.5

duckdb 7.0 ms

DuckDB and Polars stream data in batches, so they can process files bigger than RAM on a laptop. Folders of Parquet files (`'data/*.parquet'`) work the same way.

Going further

Need	Tool
Use all CPU cores for custom Python	<code>concurrent.futures</code> , <code>joblib.Parallel</code> , multiprocessing
Speed up numeric Python loops	Numba (<code>@njit</code>), Cython
Data bigger than one machine	Apache Spark (PySpark), Databricks, Dask, Ray
Warehouse-scale SQL	BigQuery, Snowflake, Redshift, Databricks SQL
GPU DataFrames	RAPIDS cuDF (<code>cudf.pandas</code>)

Rule of thumb: up to a few GB → pandas with good dtypes; up to ~100 GB on one machine → Polars or DuckDB with Parquet; beyond that or many users → Spark or a cloud warehouse.

Summary

KEY TAKEAWAYS

- Measure with `timeit` / `%timeit` before optimising; fix the bottleneck.
- Vectorise: column arithmetic, `np.where` / `np.select` , `pd.cut` , `.str` , `groupby` ; avoid `iterrows` and `apply(axis=1)` .
- Save memory with categories, downcasting, `usecols` and `chunksize` .
- Parquet is columnar, typed and compressed: faster and smaller than CSV.
- Polars (lazy, multi-core) and DuckDB (SQL, out-of-core) handle data beyond pandas; Spark and warehouses go beyond one machine.

Quick quiz

6 questions · tap an answer to check it

1. Which is usually fastest for row-wise arithmetic?

- A `iterrows`
- B `apply(axis=1)`
- C A vectorised column expression
- D A while loop

Answer: C. Vectorised code runs in compiled loops.

2. Why convert a text column with 4 distinct values to `category` ?

- A It stores each label once with small integer codes, saving memory and speeding `groupby`
- B It sorts the data
- C It is required for charts
- D It removes missing values

Answer: A. Ideal for low-cardinality columns.

3. What is the main advantage of Parquet over CSV?

- A Human-readable
- B Columnar, typed and compressed, so it is smaller and faster to read
- C Supported only by Excel
- D No schema

Answer: B. You can read just the columns you need.

4. What does Polars' `scan_parquet(...)` do?

- A Reads the whole file immediately
- B Creates a lazy query that is optimised and executed on `.collect()`
- C Writes a file
- D Converts to CSV

Answer: B. Lazy evaluation enables predicate and projection push-down.

5. Which tool queries Parquet files with SQL without a server?

- A MySQL
- B DuckDB
- C Excel
- D requests

Answer: B. It also queries pandas DataFrames.

6. When should you consider Spark or a cloud warehouse?

- A For 10,000 rows
- B When data or users exceed what one machine can handle**
- C Always
- D Never

Answer: B. Start simple; scale when needed.

EXERCISE Practice

(1) Time `iterrows`, `apply(axis=1)` and a vectorised version of `profit / sales` on `big`. (2) Reduce the memory of `superstore_sales.csv` by at least 50% and report the percentage. (3) Rewrite a pandas groupby from Lesson 18 in Polars and in DuckDB SQL and check that the numbers match.

Show solution (worked answer)

Python

```
print(t(lambda: [r["profit"] / r["sales"] for _, r in big.head(20000).iterrows()], 1))
print(t(lambda: big.profit / big.sales))
before = mb(s); s2 = s.copy()
for c in s2.select_dtypes("object").columns.union(s2.select_dtypes("string").columns):
    if s2[c].nunique() < 100: s2[c] = s2[c].astype("category")
print(f"{100 * (1 - mb(s2) / before):.0f}% smaller")
pl_res = pl.from_pandas(s).group_by("region").agg(pl.col("sales").sum()).sort("region")
dd_res = duckdb.sql("SELECT region, SUM(sales) sales FROM s GROUP BY region ORDER BY region").df()
```

INTERVIEW CORNER Questions you may be asked

Q: Your pandas script is slow. What do you do? Profile to find the slow step; vectorise loops and `apply`; reduce data early (filter, `usecols`); fix dtypes; avoid repeated work; save intermediate results as Parquet; consider Polars/DuckDB or pushing work to SQL.

Q: Why is Parquet popular in data lakes? Columnar storage enables compression and reading only needed columns; it stores the schema; and every major engine reads it.

Q: pandas vs Polars? pandas has the biggest ecosystem; Polars is faster and more memory-efficient thanks to Rust, multi-threading and lazy optimisation. Many teams use both and convert where needed.

DSA I: Big-O, Arrays, Strings, Hashing, Two Pointers and Sliding Window

Data structures and algorithms for data-analyst, data-scientist and data-engineer interviews. Learn Big-O time and space complexity, then the patterns that solve most array and string questions (hash maps and sets, prefix sums, two pointers, sliding window), with worked interview problems and timings.

● DSA for data roles (added)

✦ Python 3.13

Intermediate

🕒 120 min

By the end of this lesson you can

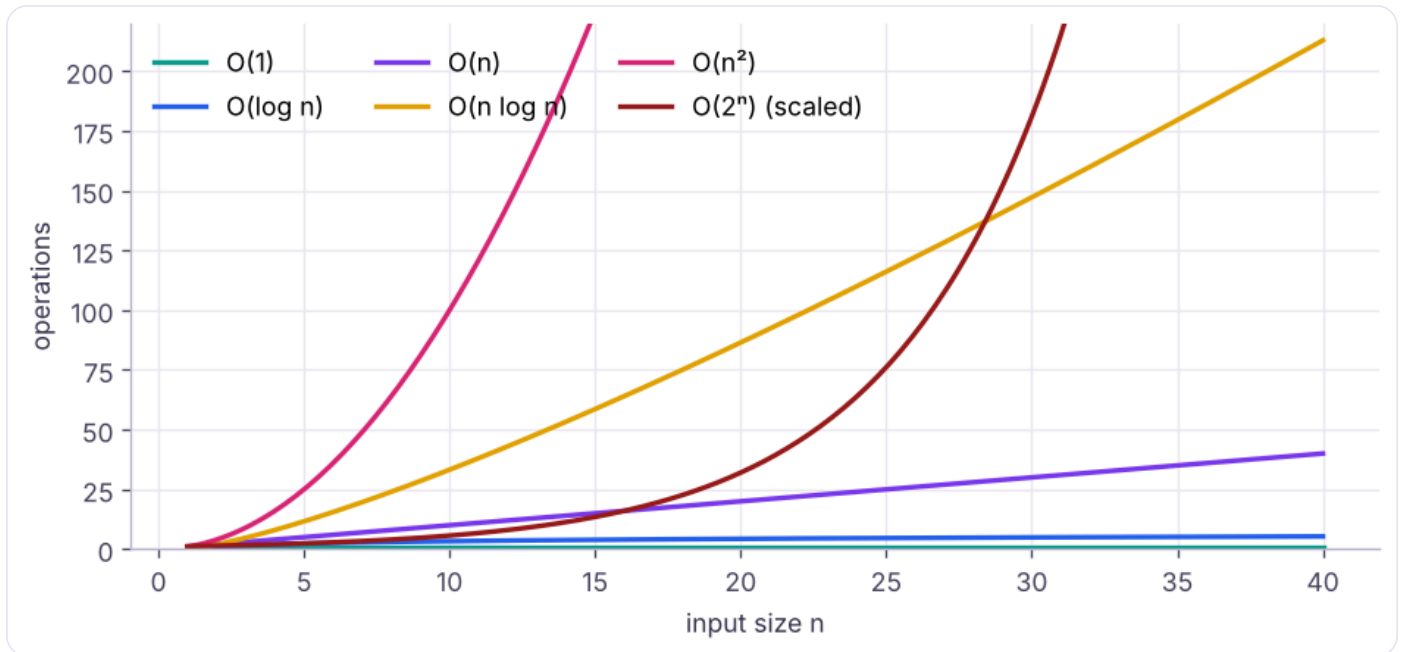
- Estimate time and space complexity with Big-O and compare common growth rates
- Know the cost of operations on lists, dicts, sets and deques
- Solve array and string problems with hash maps, counting and prefix sums
- Apply the two-pointer and sliding-window patterns
- Explain trade-offs out loud as in a coding interview

Why DSA for data roles?

Data interviews (Amazon, Flipkart, Walmart, fintechs, product companies) often include one or two coding rounds with easy-medium problems on arrays, strings, hash maps and SQL. Beyond interviews, complexity thinking explains why a join on a set is instant while a nested loop over two million rows never finishes.

Big-O notation

Big-O describes how running time (or memory) grows with input size n , ignoring constants.



Growth of common complexity classes. At $n = 1,000,000$ the difference between $O(n \log n)$ and $O(n^2)$ is days versus a second.

Big-O	Name	Example
$O(1)$	Constant	Dict lookup, list index, append
$O(\log n)$	Logarithmic	Binary search, balanced-tree lookup
$O(n)$	Linear	One loop, <code>x in list</code> , <code>sum</code>
$O(n \log n)$	Linearithmic	Sorting (<code>sorted</code> , Timsort), merge sort
$O(n^2)$	Quadratic	Nested loops over the same data
$O(2^n)$	Exponential	All subsets, naive recursion for Fibonacci

Rules: drop constants ($O(2n) \rightarrow O(n)$); keep the dominant term ($O(n^2 + n) \rightarrow O(n^2)$); sequential steps add, nested steps multiply.

Python

```
import time
def has_duplicate_quadratic(a):
    for i in range(len(a)):
        for j in range(i + 1, len(a)):
            if a[i] == a[j]: return True
    return False
def has_duplicate_set(a): return len(set(a)) != len(a)
for n in (1000, 2000, 4000):
    data = list(range(n))
    t0 = time.perf_counter(); has_duplicate_quadratic(data); t1 = time.perf_counter();
    has_duplicate_set(data); t2 = time.perf_counter()
    print(f"n={n:5d}: nested loops {1000*(t1-t0):7.1f} ms | set {1000*(t2-t1):.3f} ms")
```

OUTPUT

```
n= 1000: nested loops    15.1 ms | set 0.022 ms
n= 2000: nested loops    59.7 ms | set 0.042 ms
n= 4000: nested loops   261.3 ms | set 0.088 ms
```

Doubling n roughly quadruples the nested-loop time ($O(n^2)$) but only doubles the set version ($O(n)$).

Cost of Python operations

Operation	list	dict / set	deque
Index <code>a[i]</code>	$O(1)$	—	$O(n)$
Append / pop at end	$O(1)$	add $O(1)$	$O(1)$
Insert / pop at front	$O(n)$	—	$O(1)$
<code>x in c</code>	$O(n)$	$O(1)$	$O(n)$
Sort	$O(n \log n)$	—	—

Pattern 1: hash maps and counting

Turn "search inside a loop" ($O(n^2)$) into "look up in a dict/set" ($O(n)$).

Python

```
def two_sum(nums, target):
    """Return indices of two numbers adding up to target. O(n) time, O(n) space."""
    seen = {} # value → index
    for i, x in enumerate(nums):
        if target - x in seen: return [seen[target - x], i]
        seen[x] = i
    return []
print(two_sum([2, 7, 11, 15], 9), two_sum([3, 2, 4], 6), two_sum([1, 2], 7))

from collections import Counter, defaultdict
def is_anagram(a, b): return Counter(a) == Counter(b)
def group_anagrams(words):
    g = defaultdict(list)
    for w in words: g["".join(sorted(w))].append(w)
    return list(g.values())
def first_unique_char(s):
    c = Counter(s)
    return next((i for i, ch in enumerate(s) if c[ch] == 1), -1)
def top_k_frequent(nums, k): return [x for x, _ in Counter(nums).most_common(k)]
print(is_anagram("listen", "silent"), group_anagrams(["eat", "tea", "tan", "ate", "nat", "bat"]),
      first_unique_char("leetcode"), top_k_frequent([1, 1, 1, 2, 2, 3], 2))
```

OUTPUT

```
[0, 1] [1, 2] []
True [['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']] 0 [1, 2]
```

Pattern 2: prefix sums

Precompute running totals so any range sum is $O(1)$.

Python

```
from itertools import accumulate
sales = [5, 3, 8, 6, 2, 7]
prefix = [0] + list(accumulate(sales))      # prefix[i] = sum of first i items
range_sum = lambda i, j: prefix[j + 1] - prefix[i]  # sum of sales[i..j]
print(prefix, range_sum(1, 3), range_sum(0, 5))

def subarray_sum_equals_k(nums, k):
    """Count contiguous subarrays summing to k. O(n) with prefix sums + a counter."""
    count, running, seen = 0, 0, Counter({0: 1})
    for x in nums:
        running += x
        count += seen[running - k]
        seen[running] += 1
    return count
print(subarray_sum_equals_k([1, 1, 1], 2), subarray_sum_equals_k([1, 2, 3, -3, 3], 3))
```

OUTPUT

```
[0, 5, 8, 16, 22, 24, 31] 17 31
2 5
```

Pattern 3: two pointers

Two indices move toward each other (or in the same direction) through a **sorted** array or string.

Python

```
def is_palindrome(s):
    s = [c.lower() for c in s if c.isalnum()]
    i, j = 0, len(s) - 1
    while i < j:
        if s[i] != s[j]: return False
        i += 1; j -= 1
    return True

def pair_with_sum_sorted(a, target):
    i, j = 0, len(a) - 1
    while i < j:
        total = a[i] + a[j]
        if total == target: return a[i], a[j]
        if total < target: i += 1
        else: j -= 1
    return None

def remove_duplicates_sorted(a):          # in place, returns new length
    if not a: return 0
    w = 1
    for r in range(1, len(a)):
        if a[r] != a[w - 1]: a[w] = a[r]; w += 1
    return w

def merge_sorted(a, b):
    i = j = 0; out = []
    while i < len(a) and j < len(b):
        if a[i] <= b[j]: out.append(a[i]); i += 1
        else: out.append(b[j]); j += 1
    return out + a[i:] + b[j:]

arr = [1, 1, 2, 3, 3, 3, 4]; n = remove_duplicates_sorted(arr)
print(is_palindrome("A man, a plan, a canal: Panama"), pair_with_sum_sorted([1, 3, 4, 6, 9], 10),
      arr[:n], merge_sorted([1, 4, 9], [2, 3, 10]))
```

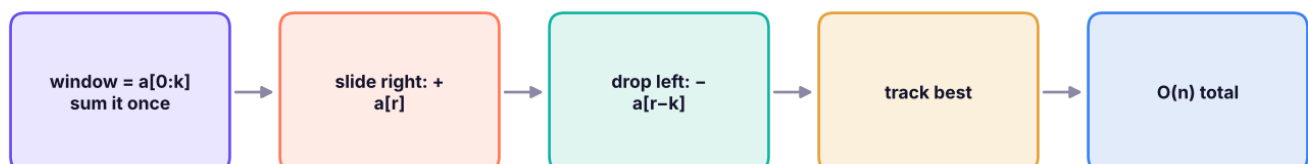
OUTPUT

True (1, 9) [1, 2, 3, 4] [1, 2, 3, 4, 9, 10]

Pattern 4: sliding window

Keep a window `[left, right]` and slide it, updating a running state instead of recomputing.

Fixed-size sliding window



A sliding window adds the new element on the right and removes the old one on the left.

Python

```
def max_sum_window(a, k):
    window = best = sum(a[:k])
    for r in range(k, len(a)):
        window += a[r] - a[r - k]
        best = max(best, window)
    return best

def longest_unique_substring(s):
    last, left, best = {}, 0, 0
    for right, ch in enumerate(s):
        if ch in last and last[ch] ≥ left: left = last[ch] + 1    # shrink past the repeat
        last[ch] = right; best = max(best, right - left + 1)
    return best

def min_window_at_least(nums, target):    # shortest subarray with sum ≥ target (positive
numbers)
    left = total = 0; best = float("inf")
    for right, x in enumerate(nums):
        total += x
        while total ≥ target:
            best = min(best, right - left + 1); total -= nums[left]; left += 1
    return 0 if best == float("inf") else best

print(max_sum_window([2, 1, 5, 1, 3, 2], 3), longest_unique_substring("abcabcbb"),
longest_unique_substring("pwwkew"), min_window_at_least([2, 3, 1, 2, 4, 3], 7))
```

OUTPUT

9 3 3 2

The same idea appears in SQL window functions and in pandas `rolling` : a 7-day moving average is a sliding window.

Classic array problems

Python

```
def max_profit(prices):                                # best single buy then sell
    lo, best = float("inf"), 0
    for p in prices: lo = min(lo, p); best = max(best, p - lo)
    return best

def max_subarray(nums):                                # Kadane's algorithm
    cur = best = nums[0]
    for x in nums[1:]: cur = max(x, cur + x); best = max(best, cur)
    return best

def product_except_self(nums):
    n = len(nums); out = [1] * n; left = right = 1
    for i in range(n): out[i] = left; left *= nums[i]
    for i in range(n - 1, -1, -1): out[i] *= right; right *= nums[i]
    return out

def move_zeros(a):
    w = 0
    for x in a:
        if x != 0: a[w] = x; w += 1
    a[w:] = [0] * (len(a) - w); return a

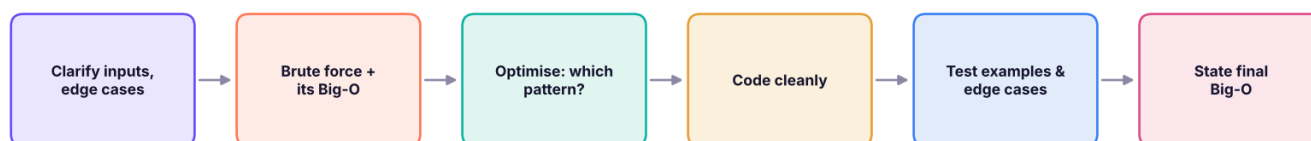
def rotate(a, k): k %= len(a); return a[-k:] + a[:-k]
print(max_profit([7, 1, 5, 3, 6, 4]), max_subarray([-2, 1, -3, 4, -1, 2, 1, -5, 4]),
      product_except_self([1, 2, 3, 4]), move_zeros([0, 1, 0, 3, 12]), rotate([1, 2, 3, 4, 5], 2))
```

OUTPUT

5 6 [24, 12, 8, 6] [1, 3, 12, 0, 0] [4, 5, 1, 2, 3]

How to answer in an interview

Interview approach



A calm, structured approach scores better than jumping into code.

Patterns to try, in order: **sort? hash map? two pointers? sliding window? prefix sum? stack? binary search? BFS/DFS? DP?** (the next two lessons cover the rest).

Summary

KEY TAKEAWAYS

- Big-O measures growth: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$; drop constants, keep the dominant term.
- `in` on a list is $O(n)$, on a set/dict $O(1)$; inserting at the front of a list is $O(n)$, use `deque`.
- Hash maps/Counters turn searches into lookups (two-sum, anagrams, frequency).
- Prefix sums give $O(1)$ range sums; two pointers work on sorted data and palindromes; sliding windows handle subarray/substring problems in $O(n)$.
- In interviews: clarify, brute force, optimise with a pattern, code, test, and state complexity.

Quick quiz

6 questions · tap an answer to check it

1. What is the time complexity of checking `x` in `my_set` ?

A $O(1)$ on average

B $O(n)$

C $O(\log n)$

D $O(n^2)$

Answer: A. Sets are hash tables.

2. Two nested loops over the same list of `n` items are...

A $O(n)$

B $O(n \log n)$

C $O(n^2)$

D $O(2n)$

Answer: C. Nested steps multiply.

3. Which pattern solves "longest substring without repeating characters" in $O(n)$?

A Binary search

B Sliding window with a dictionary of last positions

C Sorting

D Prefix sums only

Answer: B. Shrink the window past repeated characters.

4. What does the two-sum hash map store?

A Each value seen so far mapped to its index

B All pairs

C The sorted list

D Only the target

Answer: A. Check whether $\text{target} - x$ was already seen.

5. Which algorithm finds the maximum subarray sum in $O(n)$?

A Merge sort

B Kadane's algorithm

C Binary search

D Bubble sort

Answer: B. Keep the best sum ending at each position.

6. What is $O(3n + n^2 + 10)$ simplified?

A $O(3n)$

B $O(n^2)$

C $O(n)$

D $O(10)$

Answer: B. Keep the dominant term.

EXERCISE Practice set (try before opening the solutions)

(1) Return the indices of all pairs that sum to a target. (2) Check if a string has all unique characters. (3) Find the majority element (appears $> n/2$ times). (4) Longest run of consecutive 1s after flipping at most one 0 (sliding window). (5) Count customers whose total spend is in the top 10% (prefix sums not needed: think sorting).

Show solution (worked answer)

Python

```
def all_pairs(nums, t):
    seen, out = defaultdict(list), []
    for i, x in enumerate(nums):
        out += [(j, i) for j in seen[t - x]]; seen[x].append(i)
    return out
def all_unique(s): return len(set(s)) == len(s)
def majority(nums):                                # Boyer-Moore voting,  $O(1)$  space
    cand = cnt = 0
    for x in nums:
        if cnt == 0: cand = x
        cnt += 1 if x == cand else -1
    return cand
def longest_ones(a, k=1):
    left = zeros = best = 0
    for right, x in enumerate(a):
        zeros += x == 0
        while zeros > k: zeros -= a[left] == 0; left += 1
        best = max(best, right - left + 1)
    return best
print(all_pairs([1, 5, 3, 3, 7], 6), all_unique("data"), majority([2, 2, 1, 1, 2]),
      longest_ones([1, 0, 1, 1, 0, 1]))
```

INTERVIEW CORNER**Questions you may be asked**

Q: Find duplicates in a list of 10 million customer IDs. Use a set (or `Counter`) in one pass, $O(n)$ time and $O(n)$ space; in pandas `df.id.duplicated()`; in SQL `GROUP BY id HAVING COUNT(*) > 1`.

Q: Why is sorting $O(n \log n)$? Comparison sorts split the problem in half repeatedly ($\log n$ levels) and do $O(n)$ work per level; Python uses Timsort.

Q: Space vs time trade-off example? Two-sum: $O(n^2)$ time with $O(1)$ space, or $O(n)$ time with $O(n)$ extra space for a hash map.

DSA II: Stacks, Queues, Linked Lists, Recursion, Sorting and Binary Search

Learn linear data structures (stack, queue, deque, linked list), recursion and backtracking, the main sorting algorithms with their trade-offs, and binary search including "search on the answer", with interview problems such as valid parentheses, next greater element, reverse a linked list and search in a rotated array.

● DSA for data roles (added)

✦ Python 3.13

Intermediate

🕒 120 min

By the end of this lesson you can

- Implement and use stacks, queues and deques, and solve bracket and monotonic-stack problems
- Build and manipulate a singly linked list (reverse, detect a cycle, find the middle)
- Write recursive solutions with base cases and use backtracking for combinations
- Explain bubble, insertion, merge and quick sort and their complexities
- Apply binary search to sorted arrays, boundaries and answer spaces

Stacks: last in, first out

A **stack** adds and removes at the same end (LIFO). In Python a list works: `append` to push, `pop()` to pop, both $O(1)$. Uses: undo, browser back, function calls, parsing brackets.

Stack vs queue

Stack (LIFO)

- push → [a, b, c]
- pop() returns c (last in)
- list.append / list.pop
- undo, recursion, brackets

Queue (FIFO)

- enqueue → [a, b, c]
- dequeue returns a (first in)
- collections.deque: append / popleft
- task queues, BFS, streaming

Stack (LIFO) vs queue (FIFO).

Python

```
def valid_parentheses(s):
    pairs, stack = {"(": ")", "[": "]", "{": "}"}: [], []
    for ch in s:
        if ch in "([{": stack.append(ch)
        elif ch in pairs:
            if not stack or stack.pop() != pairs[ch]: return False
    return not stack
print([valid_parentheses(x) for x in ["()[]{}", "(]", "{[]}", "(((", "SUM(IF(a>1,(b),c))"]])

def eval_rpn(tokens):
    # reverse Polish notation: "3 4 + 2 *" = 14
    st = []
    for t in tokens:
        if t in "+-*/":
            b, a = st.pop(), st.pop()
            st.append({"+": a + b, "-": a - b, "*": a * b, "/": int(a / b)}[t])
        else: st.append(int(t))
    return st[0]
print(eval_rpn("3 4 + 2 *".split()))
```

OUTPUT

```
[True, False, True, False, True]
14
```

Monotonic stack

Keep the stack sorted to answer "next greater/smaller element" in $O(n)$ instead of $O(n^2)$.

Python

```
def next_greater(nums):
    res, st = [-1] * len(nums), []
    for i, x in enumerate(nums):
        while st and nums[st[-1]] < x: res[st.pop()] = x
        st.append(i)
    return res

def days_until_warmer(temps):
    res, st = [0] * len(temps), []
    for i, t in enumerate(temps):
        while st and temps[st[-1]] < t:
            j = st.pop(); res[j] = i - j
        st.append(i)
    return res
print(next_greater([2, 1, 2, 4, 3]), days_until_warmer([73, 74, 75, 71, 69, 72, 76, 73]))
```

OUTPUT

```
[4, 2, 4, -1, -1] [1, 1, 4, 2, 1, 1, 0, 0]
```


Queues and dequeues

Python

```
from collections import deque
q = deque(); q.append("job1"); q.append("job2"); q.appendleft("urgent")
print(q.popleft(), q.popleft(), list(q))
recent = deque(maxlen=3) # keeps only the last 3 items: a rolling buffer
for x in range(1, 7): recent.append(x)
print(list(recent))
def max_sliding_window(nums, k): # monotonic deque, O(n)
    dq, out = deque(), []
    for i, x in enumerate(nums):
        while dq and nums[dq[-1]] ≤ x: dq.pop()
        dq.append(i)
        if dq[0] ≤ i - k: dq.popleft()
        if i ≥ k - 1: out.append(nums[dq[0]])
    return out
print(max_sliding_window([1, 3, -1, -3, 5, 3, 6, 7], 3))
```

OUTPUT

```
urgent job1 ['job2']
[4, 5, 6]
[3, 3, 5, 5, 6, 7]
```

Never use `list.pop(0)` for a queue: it is $O(n)$. Use `deque.popleft()` ($O(1)$). `queue.Queue` adds thread safety; `heapq` gives a priority queue (next lesson).

Linked lists

A **linked list** stores nodes that point to the next node. Insert/delete at a known node is $O(1)$; access by index is $O(n)$. Python rarely needs them, but interviews love them.

Python

```
class Node:
    def __init__(self, val, nxt=None): self.val, self.next = val, nxt
def build(vals):
    head = None
    for v in reversed(vals): head = Node(v, head)
    return head
def to_list(head):
    out = []
    while head: out.append(head.val); head = head.next
    return out
def reverse(head):
    prev = None
    while head: head.next, prev, head = prev, head, head.next
    return prev
def middle(head): # slow/fast pointers
    slow = fast = head
    while fast and fast.next: slow, fast = slow.next, fast.next.next
    return slow.val
def has_cycle(head): # Floyd's tortoise and hare
    slow = fast = head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast: return True
    return False
def merge_two(a, b):
    dummy = tail = Node(0)
    while a and b:
        if a.val <= b.val: tail.next, a = a, a.next
        else: tail.next, b = b, b.next
        tail = tail.next
    tail.next = a or b; return dummy.next
lst = build([1, 2, 3, 4, 5])
print(to_list(reverse(build([1, 2, 3, 4, 5]))), middle(lst), has_cycle(lst),
to_list(merge_two(build([1, 4, 7]), build([2, 3, 9]))))
cyc = build([1, 2, 3]); cyc.next.next.next = cyc.next; print(has_cycle(cyc))
```

OUTPUT

```
[5, 4, 3, 2, 1] 3 False [1, 2, 3, 4, 7, 9]
True
```

Recursion and backtracking

A recursive function solves a problem by calling itself on a smaller version. It needs a **base case** and must move toward it.

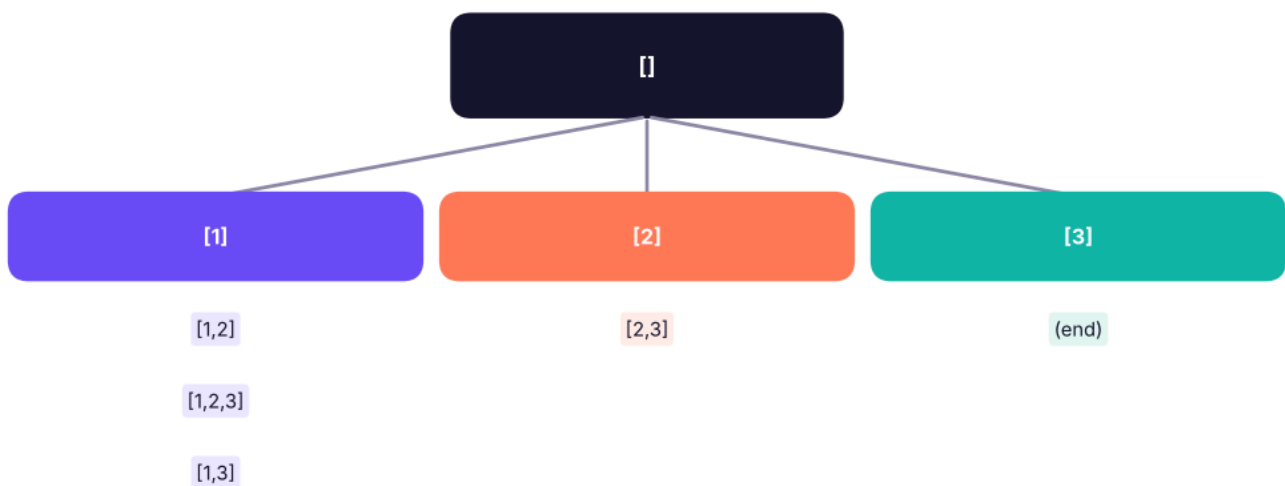
Python

```
def power(x, n):                                # fast exponentiation, O(log n)
    if n == 0: return 1
    half = power(x, n // 2)
    return half * half * (x if n % 2 else 1)
def flatten(nested):
    out = []
    for item in nested: out.extend(flatten(item) if isinstance(item, list) else [item])
    return out
def subsets(nums):                              # backtracking: explore, choose, un-choose
    res = []
    def bt(start, path):
        res.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i]); bt(i + 1, path); path.pop()
    bt(0, []); return res
def permutations(s):
    if len(s) <= 1: return [s]
    return [ch + p for i, ch in enumerate(s) for p in permutations(s[:i] + s[i + 1:])]
print(power(2, 10), flatten([1, [2, [3, 4]], 5]), subsets([1, 2, 3]), permutations("abc"))
```

OUTPUT

```
1024 [1, 2, 3, 4, 5] [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]] ['abc', 'acb',
'bac', 'bca', 'cab', 'cba']
```

Backtracking tree (subsets)



The recursion tree for subsets of [1, 2, 3]: each level decides whether to include the next item; $2^3 = 8$ leaves.

Sorting algorithms

Python

```
def bubble_sort(a):
    a = a[:]
    for i in range(len(a)):
        swapped = False
        for j in range(len(a) - 1 - i):
            if a[j] > a[j + 1]: a[j], a[j + 1] = a[j + 1], a[j]; swapped = True
        if not swapped: break
    return a

def insertion_sort(a):
    a = a[:]
    for i in range(1, len(a)):
        key, j = a[i], i - 1
        while j >= 0 and a[j] > key: a[j + 1] = a[j]; j -= 1
        a[j + 1] = key
    return a

def merge_sort(a):
    if len(a) <= 1: return a
    mid = len(a) // 2; left, right = merge_sort(a[:mid]), merge_sort(a[mid:])
    out, i, j = [], 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]: out.append(left[i]); i += 1
        else: out.append(right[j]); j += 1
    return out + left[i:] + right[j:]

def quick_sort(a):
    if len(a) <= 1: return a
    pivot = a[len(a) // 2]
    return quick_sort([x for x in a if x < pivot]) + [x for x in a if x == pivot] + quick_sort([x for x in a if x > pivot])

data = [38, 27, 43, 3, 9, 82, 10]
print(bubble_sort(data), insertion_sort(data), merge_sort(data), quick_sort(data), sorted(data) ==
merge_sort(data))
import random, time
big = [random.random() for _ in range(3000)]
for f in (insertion_sort, merge_sort, quick_sort, sorted):
    t0 = time.perf_counter(); f(big); print(f"{f.__name__:15s} {1000 * (time.perf_counter() - t0):7.1f}
ms")
```

OUTPUT

```
[3, 9, 10, 27, 38, 43, 82] [3, 9, 10, 27, 38, 43, 82] [3, 9, 10, 27, 38, 43, 82] [3, 9, 10, 27,
38, 43, 82] True
insertion_sort      133.5 ms
merge_sort          3.5 ms
quick_sort          3.5 ms
sorted              0.2 ms
```

Algorithm	Best	Average	Worst	Space	Stable	Notes
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Teaching only
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Great for small / nearly sorted
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Predictable; external sorting
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Fast in practice
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	
Timsort (Python sorted)	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Merge + insertion hybrid

Stable means equal keys keep their original order, so you can sort by city, then stably by region. In practice, use `sorted(items, key=...)` .

Binary search

On **sorted** data, halve the search space each step: $O(\log n)$. One million items need at most 20 comparisons.

Python

```
def binary_search(a, target):
    lo, hi = 0, len(a) - 1
    while lo ≤ hi:
        mid = (lo + hi) // 2
        if a[mid] == target: return mid
        if a[mid] < target: lo = mid + 1
        else: hi = mid - 1
    return -1

import bisect
a = [3, 8, 15, 15, 15, 23, 42]
print(binary_search(a, 23), binary_search(a, 5), bisect.bisect_left(a, 15), bisect.bisect_right(a, 15),
bisect.bisect_right(a, 15) - bisect.bisect_left(a, 15))

def search_rotated(nums, target):
    lo, hi = 0, len(nums) - 1
    while lo ≤ hi:
        mid = (lo + hi) // 2
        if nums[mid] == target: return mid
        if nums[lo] ≤ nums[mid]: # left half sorted
            if nums[lo] ≤ target < nums[mid]: hi = mid - 1
            else: lo = mid + 1
        else: # right half sorted
            if nums[mid] < target ≤ nums[hi]: lo = mid + 1
            else: hi = mid - 1
    return -1

print(search_rotated([4, 5, 6, 7, 0, 1, 2], 0), search_rotated([4, 5, 6, 7, 0, 1, 2], 3))

def min_capacity_to_ship(weights, days): # "binary search on the answer"
    def ok(cap):
        d, load = 1, 0
        for w in weights:
            if load + w > cap: d += 1; load = 0
            load += w
        return d ≤ days
    lo, hi = max(weights), sum(weights)
    while lo < hi:
        mid = (lo + hi) // 2
        if ok(mid): hi = mid
        else: lo = mid + 1
    return lo

print(min_capacity_to_ship([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 5))
grades = ["F", "D", "C", "B", "A"]; cutoffs = [40, 50, 65, 80]
print([grades[bisect.bisect_right(cutoffs, s)] for s in (35, 50, 79, 92)]) # banding with bisect
```

OUTPUT

```
5 -1 2 5 3
4 -1
15
['F', 'C', 'B', 'A']
```

`bisect` is how pandas `cut` and SQL range lookups work under the hood: find the band for a value in $O(\log n)$.

Summary

KEY TAKEAWAYS

- Stack = LIFO (list append/pop); queue = FIFO (deque append/popleft); monotonic stacks/deques solve next-greater and window-max in $O(n)$.
- Linked lists: pointer manipulation; slow/fast pointers find the middle and detect cycles.
- Recursion needs a base case; backtracking = choose, explore, un-choose.
- Merge sort is $O(n \log n)$ and stable; quick sort is fast on average; insertion sort suits small inputs; use sorted .
- Binary search halves the space: $O(\log n)$. Use it on sorted arrays, boundaries (bisect) and "minimum value that works" answers.

Quick quiz

6 questions · tap an answer to check it

1. Which structure checks balanced brackets?

A Stack

B Queue

C Set

D Heap

Answer: A. Push openings, pop on closings.

2. Why use deque.popleft() instead of list.pop(0) ?

A It sorts the data

B It is $O(1)$; list.pop(0) is $O(n)$

C It is thread safe

D No difference

Answer: B. Lists shift every element when removing from the front.

3. How do you find the middle of a linked list in one pass?

A Count then walk

B Slow and fast pointers

C Sort it

D Binary search

Answer: B. When fast reaches the end, slow is in the middle.

4. What is the worst-case complexity of quick sort?

A $O(n \log n)$

B $O(n^2)$

C $O(n)$

D $O(\log n)$

Answer: B. With bad pivots; randomised pivots make it unlikely.

5. How many comparisons does binary search need for about 1,000,000 sorted items?

A 1,000,000

B 1,000

C About 20

D 2

Answer: C. $\log_2(1,000,000) \approx 20$.

6. What does a "stable" sort guarantee?

- A** It never fails
- B** Items with equal keys keep their original relative order
- C** $O(n \log n)$ always
- D** It uses no memory

Answer: B. Merge sort and Timsort are stable.

EXERCISE Practice set

(1) Implement a queue using two stacks. (2) Remove adjacent duplicate letters repeatedly ("abbaca" → "ca"). (3) Find the first and last position of a target in a sorted array with `bisect`. (4) Return all combinations of k numbers from 1..n. (5) Compute the integer square root with binary search.

Show solution (worked answer)

Python

```
class QueueTwoStacks:
    def __init__(self): self.inbox, self.outbox = [], []
    def push(self, x): self.inbox.append(x)
    def pop(self):
        if not self.outbox:
            while self.inbox: self.outbox.append(self.inbox.pop())
        return self.outbox.pop()

def remove_adjacent(s):
    st = []
    for ch in s:
        if st and st[-1] == ch: st.pop()
        else: st.append(ch)
    return "".join(st)

def first_last(a, t):
    i, j = bisect.bisect_left(a, t), bisect.bisect_right(a, t) - 1
    return [i, j] if i <= j else [-1, -1]

def combine(n, k):
    res = []
    def bt(start, path):
        if len(path) == k: res.append(path[:]); return
        for i in range(start, n + 1): path.append(i); bt(i + 1, path); path.pop()
    bt(1, []); return res

def isqrt(x):
    lo, hi = 0, x
    while lo < hi:
        mid = (lo + hi + 1) // 2
        if mid * mid <= x: lo = mid
        else: hi = mid - 1
    return lo

print(remove_adjacent("abbaca"), first_last([5, 7, 7, 8, 8, 10], 8), combine(4, 2), isqrt(80))
```

INTERVIEW CORNER Questions you may be asked

Q: Explain merge sort. Split the list in half recursively until single items remain, then merge sorted halves by repeatedly taking the smaller front item; $O(n \log n)$ time, $O(n)$ extra space, stable.

Q: When is binary search applicable? When the data is sorted, or more generally when a yes/no condition is monotonic over the search space (search on the answer).

Q: Stack vs queue in real systems? Stacks: call stacks, undo, expression parsing, DFS. Queues: job and message queues (Kafka, SQS), BFS, rate-limited processing.

DSA III: Trees, Heaps, Graphs (BFS/DFS) and Dynamic Programming

Finish the DSA toolkit. Binary trees and binary search trees with traversals, heaps and priority queues with heapq (top-k, merge k lists, running median), graphs with adjacency lists, BFS and DFS, shortest paths and topological sort (as used in data pipelines), and dynamic programming from memoisation to tables.

● DSA for data roles (added)

✦ Python 3.13

Advanced

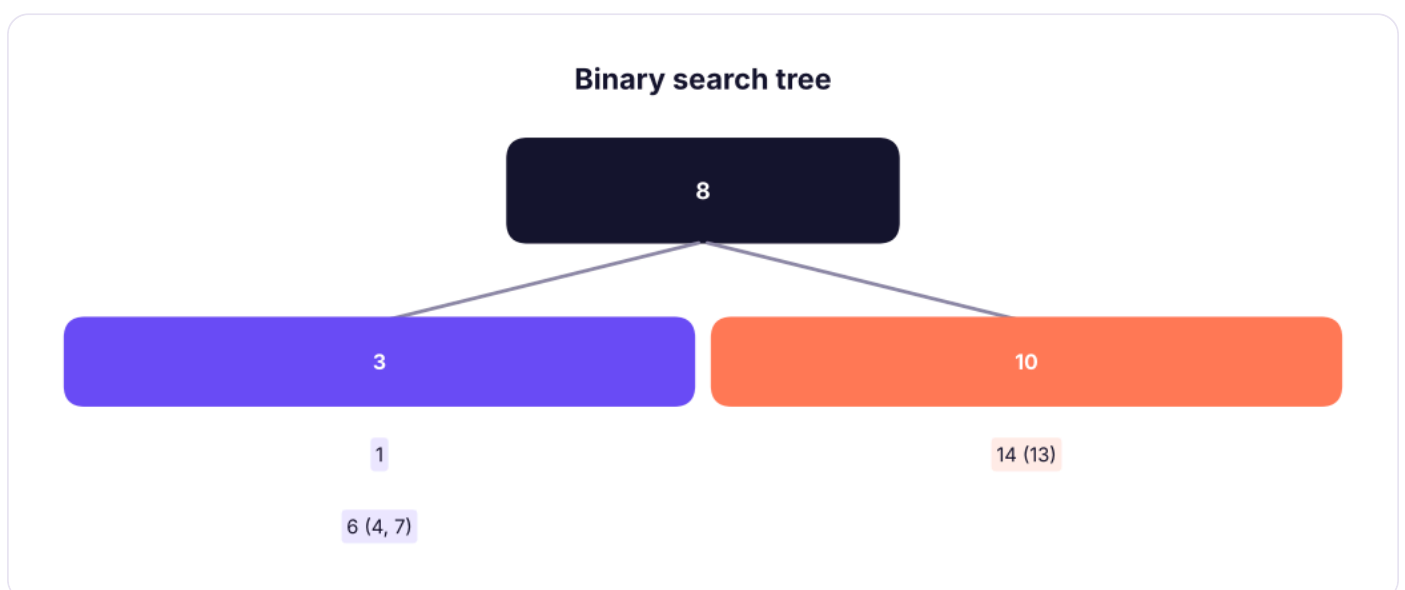
🕒 130 min

By the end of this lesson you can

- Traverse binary trees (pre-, in-, post- and level-order) and use a binary search tree
- Use heapq for top-k, k-way merge, scheduling and running medians
- Represent graphs and apply BFS, DFS, Dijkstra and topological sort
- Recognise dynamic-programming problems and solve them with memoisation and tabulation
- Connect these structures to data tools (B-tree indexes, DAG schedulers, recommendation graphs)

Trees

A **tree** is a hierarchy of nodes: one root, each node with children, no cycles. Examples: folders, org charts, JSON documents, decision trees in ML, and B-tree indexes in databases.



A binary search tree: every left child is smaller and every right child larger than its parent.

Python

```
class TreeNode:
    def __init__(self, val, left=None, right=None): self.val, self.left, self.right = val, left, right
def insert(root, val):
    # BST insert, O(height)
    if not root: return TreeNode(val)
    if val < root.val: root.left = insert(root.left, val)
    else: root.right = insert(root.right, val)
    return root
root = None
for v in [8, 3, 10, 1, 6, 14, 4, 7, 13]: root = insert(root, v)
def inorder(n): return inorder(n.left) + [n.val] + inorder(n.right) if n else []
def preorder(n): return [n.val] + preorder(n.left) + preorder(n.right) if n else []
def postorder(n): return postorder(n.left) + postorder(n.right) + [n.val] if n else []
from collections import deque
def level_order(n):
    out, q = [], deque([n])
    while q:
        level = []
        for _ in range(len(q)):
            node = q.popleft(); level.append(node.val)
            q.extend(c for c in (node.left, node.right) if c)
        out.append(level)
    return out
def height(n): return 1 + max(height(n.left), height(n.right)) if n else 0
def contains(n, x):
    while n:
        if x == n.val: return True
        n = n.left if x < n.val else n.right
    return False
print("in-order  :", inorder(root), "(sorted!)")
print("pre-order :", preorder(root)); print("post-order:", postorder(root)); print("levels    :",
level_order(root))
print("height", height(root), "| contains 7?", contains(root, 7), "| contains 5?", contains(root, 5))
```

OUTPUT

```
in-order  : [1, 3, 4, 6, 7, 8, 10, 13, 14] (sorted!)
pre-order : [8, 3, 1, 6, 4, 7, 10, 14, 13]
post-order: [1, 4, 7, 6, 3, 13, 14, 10, 8]
levels    : [[8], [3, 10], [1, 6, 14], [4, 7, 13]]
height 4 | contains 7? True | contains 5? False
```

Traversal	Order	Use
In-order	left, node, right	BST → sorted output
Pre-order	node, left, right	Copy/serialise a tree
Post-order	left, right, node	Delete, compute sizes bottom-up
Level-order (BFS)	level by level	Shortest path in unweighted trees, org levels

Python

```
def is_valid_bst(n, lo=float("-inf"), hi=float("inf")):
    return not n or (lo < n.val < hi and is_valid_bst(n.left, lo, n.val) and is_valid_bst(n.right,
n.val, hi))
def lowest_common_ancestor(n, a, b):          # in a BST
    while n:
        if a < n.val and b < n.val: n = n.left
        elif a > n.val and b > n.val: n = n.right
        else: return n.val
def max_path_sum_root_to_leaf(n):
    if not n: return float("-inf")
    if not n.left and not n.right: return n.val
    return n.val + max(max_path_sum_root_to_leaf(n.left), max_path_sum_root_to_leaf(n.right))
print(is_valid_bst(root), lowest_common_ancestor(root, 4, 7), lowest_common_ancestor(root, 1, 14),
max_path_sum_root_to_leaf(root))
```

OUTPUT

True 6 8 45

A BST is $O(\log n)$ only when **balanced**; inserting sorted data creates a "linked list" with $O(n)$ lookups. Databases use balanced **B-trees**, which is why an index makes `WHERE id = 42` fast.

Heaps and priority queues

A **heap** always gives the smallest item in $O(1)$ and inserts/removes in $O(\log n)$. Python's `heapq` is a min-heap on a list.

Python

```
import heapq
h = []
for x in [5, 1, 8, 3, 9, 2]: heapq.heappush(h, x)
print(h[0], [heapq.heappop(h) for _ in range(3)])
sales = {"Pune": 120, "Delhi": 340, "Kochi": 90, "Goa": 210, "Jaipur": 175}
print(heapq.nlargest(3, sales.items(), key=lambda kv: kv[1]))
def top_k_stream(stream, k):
    # keep a size-k min-heap: O(n log k), constant memory
    heap = []
    for x in stream:
        if len(heap) < k: heapq.heappush(heap, x)
        elif x > heap[0]: heapq.heapreplace(heap, x)
    return sorted(heap, reverse=True)
print(top_k_stream(iter([15, 3, 99, 42, 7, 64, 23, 88]), 3))
print(list(heapq.merge([1, 4, 9], [2, 3, 10], [0, 7]))) # k-way merge of sorted lists
                                                    (external sort)
tasks = [(2, "send report"), (1, "fix pipeline"), (3, "update deck")] # (priority, task)
heapq.heapify(tasks); print([heapq.heappop(tasks)[1] for _ in range(3)])

class RunningMedian:
    # two heaps: max-heap (negated) + min-heap
    def __init__(self): self.lo, self.hi = [], []
    def add(self, x):
        heapq.heappush(self.lo, -x); heapq.heappush(self.hi, -heapq.heappop(self.lo))
        if len(self.hi) > len(self.lo): heapq.heappush(self.lo, -heapq.heappop(self.hi))
    def median(self): return -self.lo[0] if len(self.lo) > len(self.hi) else (-self.lo[0] + self.hi[0])
    / 2
rm = RunningMedian(); out = []
for x in [5, 15, 1, 3, 8, 7]: rm.add(x); out.append(rm.median())
print(out)
```

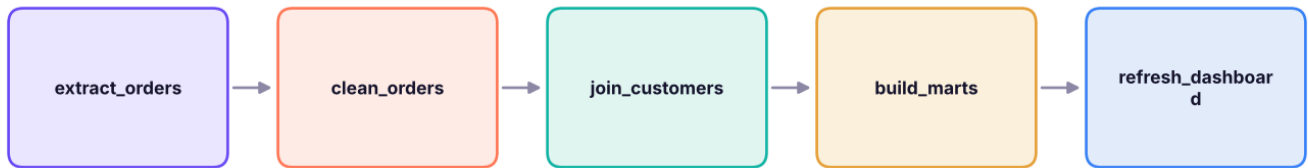
OUTPUT

```
1 [1, 2, 3]
[('Delhi', 340), ('Goa', 210), ('Jaipur', 175)]
[99, 88, 64]
[0, 1, 2, 3, 4, 7, 9, 10]
['fix pipeline', 'send report', 'update deck']
[5, 10.0, 5, 4.0, 5, 6.0]
```

Graphs

A **graph** has nodes (vertices) and edges, optionally directed and weighted. Social networks, road maps, product co-purchases, and data-pipeline dependencies are graphs.

Pipeline DAG (simplified)



A directed acyclic graph (DAG) of pipeline tasks: a task runs only after its dependencies. Airflow, dbt and Dagster schedule exactly this.

Python

```

from collections import defaultdict, deque
edges = [("A", "B"), ("A", "C"), ("B", "D"), ("C", "D"), ("D", "E"), ("F", "G")]
g = defaultdict(list)
for u, v in edges: g[u].append(v); g[v].append(u)  # undirected adjacency list
def bfs(start):
    seen, order, q = {start}, [], deque([start])
    while q:
        node = q.popleft(); order.append(node)
        for nb in g[node]:
            if nb not in seen: seen.add(nb); q.append(nb)
    return order
def dfs(node, seen=None):
    seen = seen if seen is not None else set(); seen.add(node); out = [node]
    for nb in g[node]:
        if nb not in seen: out += dfs(nb, seen)
    return out
def shortest_hops(a, b):  # BFS gives shortest path in unweighted graphs
    q, prev = deque([a]), {a: None}
    while q:
        n = q.popleft()
        if n == b: break
        for nb in g[n]:
            if nb not in prev: prev[nb] = n; q.append(nb)
    path = []
    while b is not None: path.append(b); b = prev.get(b)
    return path[::-1]
def components():
    seen, comps = set(), []
    for n in list(g):
        if n not in seen: c = dfs(n, seen); comps.append(sorted(c))
    return comps
print("BFS from A:", bfs("A"), "| DFS from A:", dfs("A"), "| path A→E:", shortest_hops("A", "E"), "| components:", components())

```

OUTPUT

```

BFS from A: ['A', 'B', 'C', 'D', 'E'] | DFS from A: ['A', 'B', 'D', 'C', 'E'] | path A→E: ['A', 'B', 'D', 'E'] | components: [['A', 'B', 'C', 'D', 'E'], ['F', 'G']]

```

Python

```
def number_of_islands(grid):  
    # classic grid DFS  
    rows, cols, count = len(grid), len(grid[0]), 0  
    def sink(r, c):  
        if 0 ≤ r < rows and 0 ≤ c < cols and grid[r][c] == "1":  
            grid[r][c] = "0"  
            for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)): sink(r + dr, c + dc)  
    for r in range(rows):  
        for c in range(cols):  
            if grid[r][c] == "1": count += 1; sink(r, c)  
    return count  
print(number_of_islands([list("11000"), list("11000"), list("00100"), list("00011"))])
```

OUTPUT

3

Dijkstra's shortest path (weighted)

Python

```
def dijkstra(graph, src):  
    dist = {src: 0}; pq = [(0, src)]  
    while pq:  
        d, u = heapq.heappop(pq)  
        if d > dist.get(u, float("inf")): continue  
        for v, w in graph.get(u, []):  
            if d + w < dist.get(v, float("inf")): dist[v] = d + w; heapq.heappush(pq, (d + w, v))  
    return dist  
roads = {"Mumbai": [("Pune", 150), ("Nashik", 170)], "Pune": [("Satara", 110), ("Nashik", 210)],  
         "Nashik": [("Aurangabad", 180)], "Satara": [("Kolhapur", 120)], "Aurangabad": [], "Kolhapur": []}  
print(dijkstra(roads, "Mumbai"))
```

OUTPUT

{'Mumbai': 0, 'Pune': 150, 'Nashik': 170, 'Satara': 260, 'Aurangabad': 350, 'Kolhapur': 380}

Topological sort: ordering a pipeline

Python

```
deps = {"extract_orders": [], "extract_customers": [], "clean_orders": ["extract_orders"],
"join_customers": ["clean_orders", "extract_customers"],
      "build_marts": ["join_customers"], "refresh_dashboard": ["build_marts"], "train_model":
["join_customers"]}
def topo_sort(deps):
    # Kahn's algorithm
    indeg = {t: len(d) for t, d in deps.items()}; children = defaultdict(list)
    for t, ds in deps.items():
        for d in ds: children[d].append(t)
    q = deque(sorted(t for t, k in indeg.items() if k == 0)); order = []
    while q:
        t = q.popleft(); order.append(t)
        for c in sorted(children[t]):
            indeg[c] -= 1
            if indeg[c] == 0: q.append(c)
    if len(order) != len(deps): raise ValueError("cycle detected")
    return order
print(topo_sort(deps))
from graphlib import TopologicalSorter # standard-library version
print(list(TopologicalSorter(deps).static_order())[:3])
```

OUTPUT

```
['extract_customers', 'extract_orders', 'clean_orders', 'join_customers', 'build_marts',
'train_model', 'refresh_dashboard']
['extract_orders', 'extract_customers', 'clean_orders']
```

Dynamic programming

Dynamic programming (DP) solves problems with **overlapping subproblems** and **optimal substructure** by storing answers to subproblems instead of recomputing them.

Python

```
import functools, time
def fib_naive(n): return n if n < 2 else fib_naive(n - 1) + fib_naive(n - 2)
@functools.cache
def fib_memo(n): return n if n < 2 else fib_memo(n - 1) + fib_memo(n - 2)
def fib_table(n):
    a, b = 0, 1
    for _ in range(n): a, b = b, a + b
    return a
t0 = time.perf_counter(); fib_naive(27); t1 = time.perf_counter(); fib_memo(27); t2 =
time.perf_counter()
print(f"naive {1000*(t1-t0):.0f} ms (O(2^n)) | memoised {1000*(t2-t1):.3f} ms (O(n)) | values equal:
{fib_naive(20) == fib_memo(20) == fib_table(20)}")
```

OUTPUT

```
naive 31 ms (O(2^n)) | memoised 0.022 ms (O(n)) | values equal: True
```

Two ways to do DP

Top-down (memoisation)

- Write the natural recursion
- Cache results (@functools.cache)
- Computes only needed states
- Risk: recursion depth

Bottom-up (tabulation)

- Fill a table from small to large
- Loops, no recursion
- Often $O(1)$ extra space with rolling variables
- Must know the order of states

Top-down (memoisation) vs bottom-up (tabulation) dynamic programming.

Python

```
def climb_stairs(n):                                # 1 or 2 steps at a time
    a, b = 1, 1
    for _ in range(n): a, b = b, a + b
    return a

def coin_change(coins, amount):                     # fewest coins
    dp = [0] + [float("inf")] * amount
    for x in range(1, amount + 1):
        dp[x] = min((dp[x - c] + 1 for c in coins if c ≤ x), default=float("inf"))
    return dp[amount] if dp[amount] ≠ float("inf") else -1

def house_robber(nums):                             # no two adjacent
    take = skip = 0
    for x in nums: take, skip = skip + x, max(take, skip)
    return max(take, skip)

def lis(nums):                                       # longest increasing subsequence,  $O(n \log n)$ 
    import bisect; tails = []
    for x in nums:
        i = bisect.bisect_left(tails, x)
        tails[i:i + 1] = [x]
    return len(tails)

def edit_distance(a, b):                             # Levenshtein: used in fuzzy matching of names
    dp = list(range(len(b) + 1))
    for i in range(1, len(a) + 1):
        prev, dp[0] = dp[0], i
        for j in range(1, len(b) + 1):
            prev, dp[j] = dp[j], min(dp[j] + 1, dp[j - 1] + 1, prev + (a[i - 1] ≠ b[j - 1]))
    return dp[-1]

def knapsack(items, cap):                           # 0/1 knapsack: (weight, value)
    dp = [0] * (cap + 1)
    for w, v in items:
        for c in range(cap, w - 1, -1): dp[c] = max(dp[c], dp[c - w] + v)
    return dp[cap]

print(climb_stairs(10), coin_change([1, 2, 5], 11), coin_change([2], 3), house_robber([2, 7, 9, 3, 1]),
      lis([10, 9, 2, 5, 3, 7, 101, 18]))
print(edit_distance("Bangalore", "Bengaluru"), edit_distance("kitten", "sitting"), knapsack([(1, 1),
(3, 4), (4, 5), (5, 7)], 7))
```

OUTPUT

```
89 3 -1 12 4
3 3 9
```

How to spot DP: "minimum/maximum/number of ways", choices at each step, and the same subproblem appearing again. Define the **state** (`dp[x]` = best answer for amount x), the **transition**, and the **base case**.

Where these appear in data work

Structure	In data tools
B-trees	Database indexes
Hash tables	Hash joins, <code>GROUP BY</code> , Python dicts, Spark shuffles
Heaps	Top-N queries, schedulers, k-way merge in external sorting
DAGs + topological sort	Airflow, dbt, Dagster, Spark execution plans
Graphs + BFS	Recommendations ("people also bought"), fraud rings, knowledge graphs for GenAI
Edit distance (DP)	Fuzzy matching, deduplicating customer names
Trees	Decision trees, random forests, gradient boosting

Summary

KEY TAKEAWAYS

- Trees: four traversals; a BST's in-order traversal is sorted; balanced trees give $O(\log n)$ search (database B-tree indexes).
- Heaps (`heapq`) give the minimum in $O(1)$ and push/pop in $O(\log n)$: top-k, k-way merge, priority scheduling, running median.
- Graphs as adjacency lists; BFS (queue) for levels and shortest unweighted paths, DFS (recursion/stack) for components and islands; Dijkstra for weighted paths; topological sort for DAG pipelines.
- DP = overlapping subproblems + optimal substructure; use memoisation (`@functools.cache`) or tabulation; define state, transition and base case.

Quick quiz

6 questions · tap an answer to check it

1. Which traversal of a BST returns the values in sorted order?

A Pre-order

B In-order

C Post-order

D Level-order

Answer: B. Left subtree, node, right subtree.

2. What does `heapq` give you in $O(1)$?

A The smallest item

B The largest item

C The median

D A sorted list

Answer: A. Negate values for a max-heap.

3. Which search finds the shortest path in an unweighted graph?

A BFS

B DFS

C Binary search

D Sorting

Answer: A. BFS explores level by level.

4. Which algorithm orders tasks so each runs after its dependencies?

A Dijkstra

B Topological sort

C Merge sort

D Kadane

Answer: B. It works on directed acyclic graphs.

5. Two signs that a problem suits dynamic programming are...

A Overlapping subproblems and optimal substructure

B Sorted input and unique values

C A graph and a heap

D Strings only

Answer: A. Store and reuse subproblem answers.

6. Why does a database index speed up lookups?

A It stores data in CSV

B It keeps keys in a balanced tree (B-tree) so lookups are $O(\log n)$

C It caches every query

D It compresses the table

Answer: B. Without an index the database scans every row.

EXERCISE Practice set

(1) Return the maximum depth and the left view of a binary tree. (2) Find the k closest points to the origin with a heap. (3) Detect a cycle in a directed graph of task dependencies. (4) Count the number of ways to make change for an amount. (5) Longest common subsequence of two strings.

Show solution (worked answer)

Python

```
def left_view(n): return [lvl[0] for lvl in level_order(n)]
def k_closest(points, k): return heapq.nsmallest(k, points, key=lambda p: p[0]**2 + p[1]**2)
def has_cycle_directed(deps):
    try: topo_sort(deps); return False
    except ValueError: return True
def ways_to_change(coins, amount):
    dp = [1] + [0] * amount
    for c in coins:
        for x in range(c, amount + 1): dp[x] += dp[x - c]
    return dp[amount]
def lcs(a, b):
    dp = [[0] * (len(b) + 1) for _ in range(len(a) + 1)]
    for i in range(1, len(a) + 1):
        for j in range(1, len(b) + 1):
            dp[i][j] = dp[i - 1][j - 1] + 1 if a[i - 1] == b[j - 1] else max(dp[i - 1][j], dp[i][j - 1])
    return dp[-1][-1]
print(height(root), left_view(root), k_closest([(1, 3), (-2, 2), (5, 8)], 2),
has_cycle_directed({"a": ["b"], "b": ["a"]}), ways_to_change([1, 2, 5], 5), lcs("abcde", "ace"))
```

INTERVIEW CORNER Questions you may be asked

Q: BFS vs DFS? BFS uses a queue, explores by distance and finds shortest unweighted paths; DFS uses recursion or a stack, goes deep first, and suits connectivity, cycle detection and topological ordering. Both are $O(V + E)$.

Q: How would you get the top 10 products from a stream of billions of sales? Maintain a size-10 min-heap of running totals per product (or approximate with Count-Min Sketch), or aggregate in SQL/Spark and take the top 10.

Q: Memoisation vs tabulation? Memoisation caches results of a top-down recursion; tabulation fills a table bottom-up. Same complexity; tabulation avoids recursion limits and often allows space optimisation.

Python Capstone: End-to-End Retail Sales Analytics Project

Bring the whole Python track together on the Superstore data. Structure a project, build a validated data pipeline, compute KPIs, growth and Pareto analysis, cohort retention and discount impact, create a chart pack and an Excel report, and write the README and executive summary you can put in your portfolio.

● Python capstone

✦ pandas 3 · Matplotlib 3.10 · openpyxl

Advanced

⌚ 180 min

By the end of this lesson you can

- Organise an analysis as a reproducible project with functions, checks and outputs
- Build a load → validate → transform pipeline with assertions
- Analyse KPIs, growth, Pareto concentration, cohorts and discount effects
- Produce a multi-sheet Excel report and a chart pack
- Write an executive summary and a portfolio-ready README

The brief

The sales director of an Indian office-supplies retailer asks: **How did 2025 go, where do we make and lose money, how loyal are our customers, and what should we change in 2026?** Deliver a short report with charts and an Excel workbook her team can filter.

Project pipeline



Capstone pipeline.

Project layout (Lesson 13): `data/raw` , `src/` (functions below), `notebooks/` (exploration), `reports/` (outputs), `README.md` .

1. Load and validate

Python

```
import pandas as pd, numpy as np, matplotlib.pyplot as plt
pd.set_option("display.max_columns", 20); pd.set_option("display.width", 160)

def load(path="superstore_sales.csv"):
    df = pd.read_csv(path, parse_dates=["order_date", "ship_date"])
    return df

def validate(df):
    checks = {
        "row_id unique": df["row_id"].is_unique,
        "no missing values": df.isna().sum().sum() == 0,
        "sales positive": (df["sales"] > 0).all(),
        "discount between 0 and 1": df["discount"].between(0, 1).all(),
        "ship after order": (df["ship_date"] ≥ df["order_date"]).all(),
        "quantity 1-20": df["quantity"].between(1, 20).all(),
    }
    for name, ok in checks.items(): print(f" {'PASS' if ok else 'FAIL'} {name}")
    assert all(checks.values()), "validation failed"
    return df

s = validate(load())
print(s.shape, s.order_date.min().date(), "→", s.order_date.max().date())
```

OUTPUT

```
PASS row_id unique
PASS no missing values
PASS sales positive
PASS discount between 0 and 1
PASS ship after order
PASS quantity 1-20
(6323, 19) 2023-01-01 → 2025-12-31
```

2. Transform and enrich

Python

```
def enrich(df):
    out = df.copy()
    out["year"] = out.order_date.dt.year
    out["month"] = out.order_date.dt.to_period("M")
    out["margin_pct"] = 100 * out.profit / out.sales
    out["ship_days"] = (out.ship_date - out.order_date).dt.days
    out["discount_band"] = pd.cut(out.discount, [-0.01, 0, 0.1, 0.2, 1], labels=["0%", "1-10%", "11-20%", ">20%"])
    return out

s = enrich(s)
print(s[["order_id", "year", "month", "sales", "profit", "margin_pct", "discount_band"]].head(3))
```

OUTPUT

	order_id	year	month	sales	profit	margin_pct	discount_band
0	IN-2023-00001	2023	2023-01	36000.0	3472.55	9.645972	0%
1	IN-2023-00001	2023	2023-01	3850.0	1169.54	30.377662	0%
2	IN-2023-00001	2023	2023-01	62080.0	6044.47	9.736582	0%

3. Headline KPIs and growth

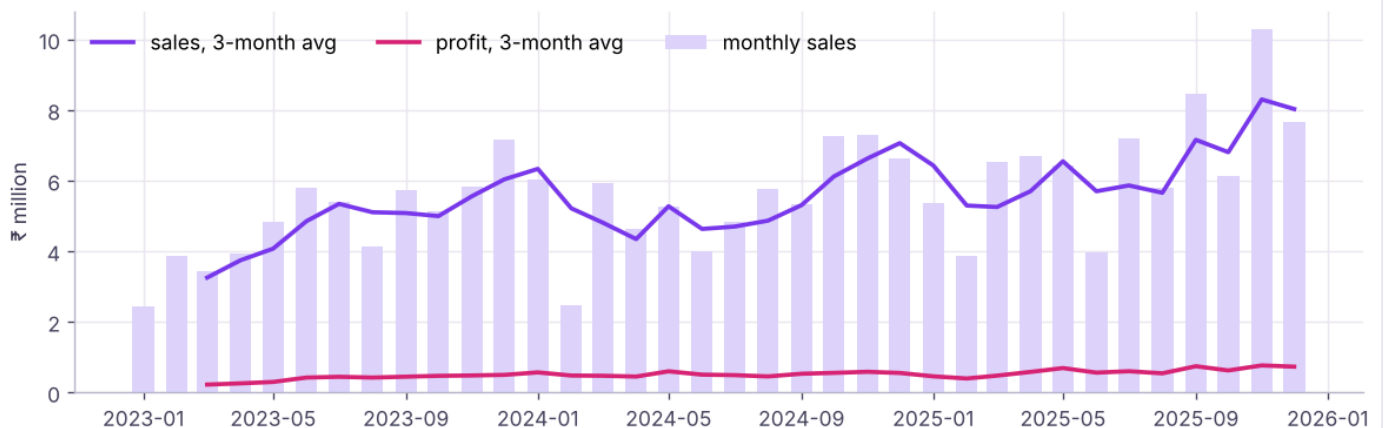
Python

```
def kpis(df):
    g = df.groupby("year").agg(sales=("sales", "sum"), profit=("profit", "sum"), orders=("order_id",
"nunique"), customers=("customer_id", "nunique"), units=("quantity", "sum"))
    g["margin_%"] = 100 * g.profit / g.sales
    g["aov"] = g.sales / g.orders
    g["sales_growth_%"] = 100 * g.sales.pct_change()
    g["profit_growth_%"] = 100 * g.profit.pct_change()
    return g
k = kpis(s)
view = k.assign(sales=k.sales / 1e6, profit=k.profit / 1e6, aov=k.aov / 1e3).round(1)
view.columns = ["sales ₹M", "profit ₹M", "orders", "customers", "units", "margin %", "AOV ₹K", "sales
growth %", "profit growth %"]
print(view)
```

OUTPUT

	sales ₹M	profit ₹M	orders	customers	units	margin %	AOV ₹K	sales growth %	profit
growth %									
year									
2023	57.6	4.7	883	540	4714	8.2	65.3		NaN
NaN									
2024	65.4	6.1	1085	590	5404	9.4	60.3	13.5	
29.9									
2025	78.4	7.5	1232	623	6266	9.6	63.6	19.8	
22.1									

Sales and profit both grew each year; profit grew faster



Monthly sales and profit, 2023–2025, with 3-month rolling averages.

4. Where we make and lose money

Python

```
sub = s[s.year == 2025].groupby(["category", "sub_category"]).agg(sales=("sales", "sum"), profit=
("profit", "sum"), avg_disc=("discount", "mean"))
sub["margin_%"] = (100 * sub.profit / sub.sales).round(1)
sub = sub.sort_values("profit")
print((sub.assign(sales=sub.sales / 1e6, profit=sub.profit / 1e6)).round(2))
```

OUTPUT

		sales	profit	avg_disc	margin_%
category	sub_category				
Furniture	Tables	5.30	-0.29	0.17	-5.4
	Bookcases	3.92	-0.17	0.19	-4.3
Office Supplies	Art	0.10	0.02	0.08	15.3
	Binders	0.16	0.02	0.08	14.2
Furniture	Furnishings	0.49	0.03	0.08	5.2
Office Supplies	Paper	0.19	0.03	0.07	15.8
Furniture	Chairs	3.05	0.13	0.08	4.3
Office Supplies	Storage	0.98	0.14	0.08	14.1
Technology	Accessories	1.22	0.15	0.07	11.9
Office Supplies	Appliances	2.54	0.38	0.07	14.9
Technology	Phones	12.94	1.71	0.06	13.2
	Machines	16.50	2.00	0.08	12.1
	Copiers	30.98	3.35	0.08	10.8

Python

```
db = s.groupby("discount_band", observed=True).agg(lines=("row_id", "size"), sales=("sales", "sum"),
profit=("profit", "sum"))
db["margin_%"] = (100 * db.profit / db.sales).round(1); db["share_of_lines_%"] = (100 * db.lines /
db.lines.sum()).round(1)
print(db.assign(sales=(db.sales / 1e6).round(1), profit=(db.profit / 1e6).round(2)))
```

OUTPUT

	lines	sales	profit	margin_%	share_of_lines_%
discount_band					
0%	3480	123.9	21.72	17.5	55.0
1-10%	726	21.9	1.29	5.9	11.5
11-20%	1217	35.9	-0.59	-1.6	19.2
>20%	900	19.7	-4.06	-20.6	14.2



Margin by discount band: deep discounts turn profitable lines into losses.

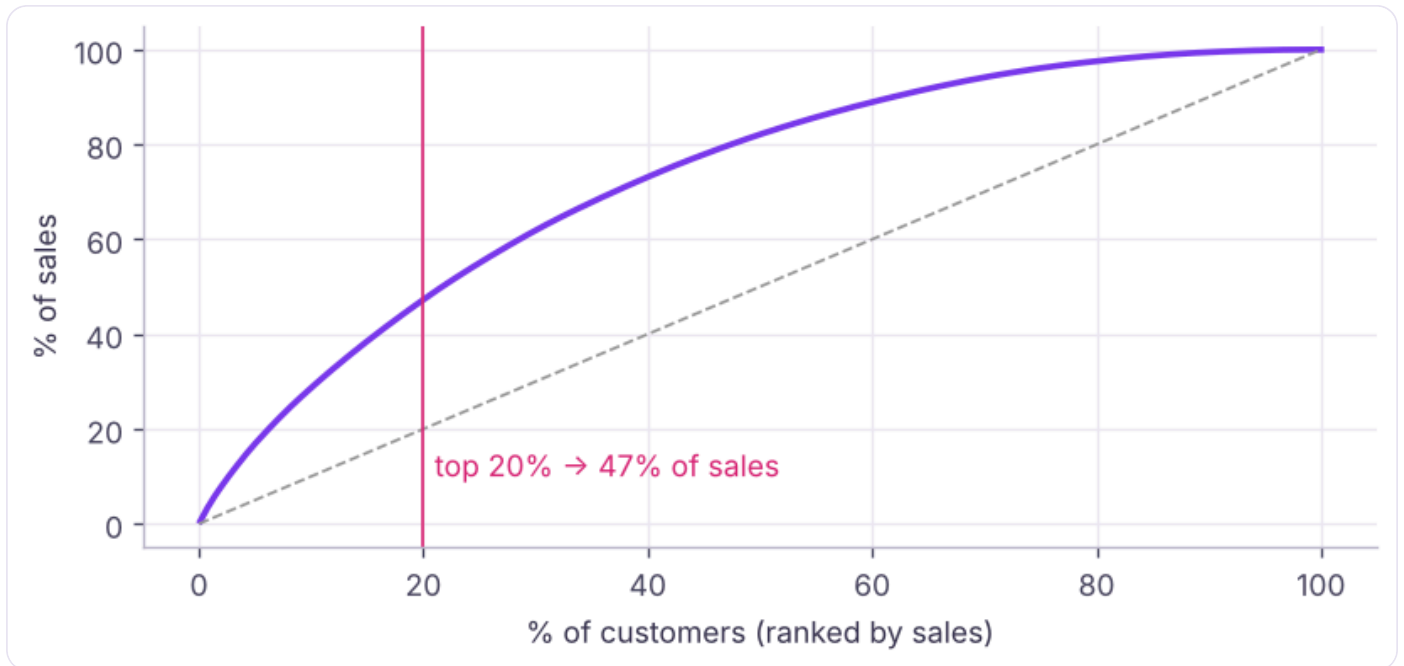
5. Customer concentration (Pareto)

Python

```
cust = s.groupby("customer_id")["sales"].sum().sort_values(ascending=False)
cum = cust.cumsum() / cust.sum()
top20 = int(len(cust) * 0.2)
print(f"{len(cust)} customers | top 20% ({top20}) bring {cum.iloc[top20 - 1]:.0%} of sales | customers  
needed for 50% of sales: {(cum < 0.5).sum() + 1}")
```

OUTPUT

785 customers | top 20% (157) bring 47% of sales | customers needed for 50% of sales: 171



Pareto curve of customer sales.

6. Cohort retention

A **cohort** groups customers by the month (here: quarter) of their first purchase; retention tracks how many buy again later.

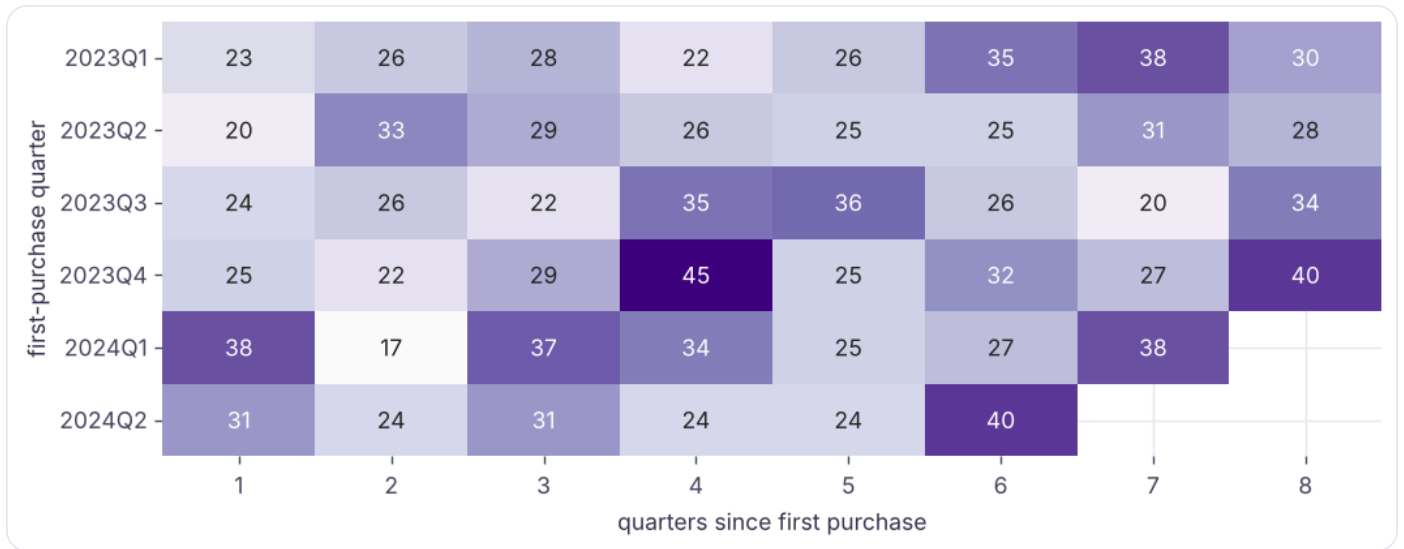
Python

```
s["quarter"] = s.order_date.dt.to_period("Q")
first = s.groupby("customer_id")["quarter"].min().rename("cohort")
c = s.join(first, on="customer_id")
c["age_q"] = (c.quarter - c.cohort).apply(lambda d: d.n)
active = c.groupby(["cohort", "age_q"])["customer_id"].nunique().unstack()
retention = (active.div(active[0], axis=0) * 100).round(0)
print(active[0].rename("cohort size").head(6).to_dict())
print(retention.iloc[:6, :8])
```

OUTPUT

```
{Period('2023Q1', 'Q-DEC'): 151.0, Period('2023Q2', 'Q-DEC'): 142.0, Period('2023Q3', 'Q-DEC'): 127.0, Period('2023Q4', 'Q-DEC'): 120.0, Period('2024Q1', 'Q-DEC'): 71.0, Period('2024Q2', 'Q-DEC'): 42.0}
```

age_q	0	1	2	3	4	5	6	7
cohort								
2023Q1	100.0	23.0	26.0	28.0	22.0	26.0	35.0	38.0
2023Q2	100.0	20.0	33.0	29.0	26.0	25.0	25.0	31.0
2023Q3	100.0	24.0	26.0	22.0	35.0	36.0	26.0	20.0
2023Q4	100.0	25.0	22.0	29.0	45.0	25.0	32.0	27.0
2024Q1	100.0	38.0	17.0	37.0	34.0	25.0	27.0	38.0
2024Q2	100.0	31.0	24.0	31.0	24.0	24.0	40.0	NaN



Quarterly cohort retention (% of each cohort buying in later quarters).

7. Operations: shipping

Python

```
ship = s.groupby("ship_mode").agg(lines=("row_id", "size"), avg_days=("ship_days", "mean"), sales=
("sales", "sum"))
ship["share_%"] = (100 * ship.lines / ship.lines.sum()).round(1)
print(ship.round(1).sort_values("avg_days"))
reg = s[s.year == 2025].groupby("region").agg(sales=("sales", "sum"), profit=("profit", "sum"))
reg["margin_%"] = (100 * reg.profit / reg.sales).round(1); print((reg.assign(sales=reg.sales / 1e6,
profit=reg.profit / 1e6)).round(2))
```

OUTPUT

	lines	avg_days	sales	share_%
ship_mode				
Same Day	321	0.0	10082429.5	5.1
First Class	948	1.5	31455543.0	15.0
Second Class	1249	3.0	40858316.0	19.8
Standard Class	3805	5.6	119017280.5	60.2
	sales	profit	margin_%	
region				
East	18.91	2.07	10.9	
North	16.43	1.56	9.5	
South	20.80	1.77	8.5	
West	22.23	2.10	9.4	

8. Excel report for the business

Python

```
from pathlib import Path
out = Path("reports"); out.mkdir(exist_ok=True)
with pd.ExcelWriter(out / "sales_review_2025.xlsx", engine="openpyxl") as xw:
    view.to_excel(xw, sheet_name="KPIs")
    sub.reset_index().to_excel(xw, sheet_name="Sub-category 2025", index=False)
    db.reset_index().to_excel(xw, sheet_name="Discount bands", index=False)
    retention.reset_index().astype(str).to_excel(xw, sheet_name="Cohort retention", index=False)
    s[s.year == 2025].drop(columns=["month", "quarter"]).to_excel(xw, sheet_name="Data 2025",
index=False)
import openpyxl
wb = openpyxl.load_workbook(out / "sales_review_2025.xlsx")
for ws in wb.worksheets:
    ws.freeze_panes = "B2"; ws.auto_filter.ref = ws.dimensions
    for col in ws.columns: ws.column_dimensions[col[0].column_letter].width = 16
wb.save(out / "sales_review_2025.xlsx")
print(wb.sheetnames, f"{{(out / 'sales_review_2025.xlsx').stat().st_size / 1e6:.1f}} MB")
import shutil; shutil.rmtree(out)
```

OUTPUT

```
['KPIs', 'Sub-category 2025', 'Discount bands', 'Cohort retention', 'Data 2025'] 0.3 MB
```

9. Executive summary (template)

Write the summary **from the numbers above**; every claim should point to a table or chart.

KEY TAKEAWAYS

Fill each line from your outputs: 1. **Growth**: 2025 sales of ₹M (+% on 2024), profit ₹M (+%), margin __%. 2. **Profit engines**: __ and __ deliver __% of profit; __ and **lose mon.** 3. ****Discounts****: lines discounted above 20% he a __% mgin, against __% at no discount. 4. ****Customers****: the top 20% of customers brin__% of **sales, an**__% of a cohort is still buying a year later. 5. **Actions for 2026**: cap discounts on __; review pricing of __; a loyalty programme for the top customers; a retention push in the first quarter after a customer's first purchase.

10. README for your portfolio

Text

```
# Superstore India: 2025 Sales Review (Python)
**Question:** How did 2025 perform, where is profit made and lost, how loyal are customers?
**Data:** 6,323 order lines, 2023-2025 (synthetic, modelled on the Superstore dataset).
**Tools:** pandas, Matplotlib, Seaborn, openpyxl.
**Method:** validation checks → enrichment → KPIs, Pareto, cohort retention, discount analysis.
**Key findings:** (3-5 bullets with numbers)
**Recommendations:** (3 bullets)
**How to run:** `pip install -r requirements.txt && python src/run_report.py`
**Repo:** data/ · src/ · notebooks/ · reports/ · README.md
```

Summary

KEY TAKEAWAYS

- A project is a pipeline: load → validate (assertions) → enrich → analyse → visualise → report.
- KPI tables with growth, sub-category profitability, discount bands, Pareto concentration and cohort retention answer most retail questions.
- `pd.ExcelWriter` + `openpyxl` create business-ready, filterable workbooks.
- Finish with an executive summary tied to numbers and a portfolio README.

Quick quiz

6 questions · tap an answer to check it

1. Why put validation checks at the start of the pipeline?

A To stop early if the data breaks assumptions, rather than producing wrong results

B To make the code longer

C Because pandas requires it

D To speed up loading

Answer: A. Assertions make failures loud.

2. What does a Pareto analysis of customers show?

- A Average sales
- B How concentrated sales are among the top customers
- C Retention over time
- D Discount impact

Answer: B. Often a minority of customers brings most of the revenue.

3. What is a cohort in retention analysis?

- A Customers grouped by the period of their first purchase
- B Customers in the same city
- C All customers
- D Products in a category

Answer: A. Retention follows each cohort over time.

4. Which tool writes several sheets to one Excel file?

- A `df.to_csv`
- B `pd.ExcelWriter`
- C `openpyxl.load_workbook` only
- D `df.to_json`

Answer: B. Then format with openpyxl.

5. What belongs in an executive summary?

- A All code
- B Key numbers, findings and recommended actions**
- C Raw data
- D Library versions

Answer: B. Lead with the answer.

6. Why write analysis steps as functions?

- A They can be tested, re-run on new data and reused**
- B They run faster in all cases
- C Notebooks forbid loops
- D To avoid variables

Answer: A. Reproducibility matters.

EXERCISE Extend the capstone

(1) Add product-level ABC analysis (A = top 80% of sales). (2) Compute RFM scores for 2025 customers (see ML track lesson on segmentation). (3) Turn the chart cells into a function that saves PNGs to `reports/`. (4) Publish the project on GitHub with the README template and one chart in the README.

Show solution (worked answer)

Python

```
prod = s.groupby("product_name")["sales"].sum().sort_values(ascending=False)
share = prod.cumsum() / prod.sum()
abc = pd.cut(share, [0, .8, .95, 1], labels=["A", "B", "C"]); print(abc.value_counts())
snap = s.order_date.max() + pd.Timedelta(days=1)
rfm = s[s.year == 2025].groupby("customer_id").agg(R=("order_date", lambda d: (snap - d.max()).days), F=("order_id", "nunique"), M=("sales", "sum"))
rfm["score"] = pd.qcut(-rfm.R, 4, labels=False) + pd.qcut(rfm.F.rank(method="first"), 4, labels=False) + pd.qcut(rfm.M, 4, labels=False)
print(rfm.sort_values("score", ascending=False).head())
```

INTERVIEW CORNER**Questions you may be asked**

Q: Walk me through a project you are proud of. Use STAR: the business situation and question, your task, the actions (data, cleaning, analysis, tools), and the result with numbers and what changed. Keep it to two minutes and be ready to go deeper on any step.

Q: How do you make sure your numbers are right? Validation checks, reconciling totals with a trusted source, spot-checking individual records, peer review, and versioned, re-runnable code.

Q: What would you do next with more time? Automate the refresh (schedule + Parquet/database), add a dashboard, build a churn or demand-forecast model, and run an experiment on discount policy.

TRACK 7