



Python Course 2025/26/2

Lecture 1: Welcome, Setup, and a Grand Tour of Python in 2026



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Today's path

- What this course is and how you will be assessed
- Why Python is the language of choice in 2026 (history, features, applications)
- Installing Python the modern way (`uv` / `pixi`, virtual envs, first program)
- Development environments: VSCode, PyCharm, JupyterLab, Cursor, Claude Code
- AI-assisted development: capabilities, limits, prompting, ethics, and our course AI policy
- A live **Grand Tour** previewing lectures 2-12

Learning objective

By the end of this lecture you can install Python, run your first program, choose an IDE, and explain how AI tools fit (and don't fit) into the way **you** learn to code.

How to read these slides

Green code blocks are **runnable**. Boxes with a green left bar are notes / fun facts. Each section ends with a quick checkpoint quiz.

Setup

Tools

AI policy

Course preview

Lecture 1 is the on-ramp. Everything we set up today, you will use across the next 11 lectures.



About This Course

12 weeks, one running to-do app, ending in real data & AI work



Syllabus: 12 Lectures, One Story

Foundations & software craft (L1-L8)

- **L1:** Setup, tools, AI policy, Grand Tour
- **L2:** Data types, variables, expressions
- **L3:** Flow control, strings, lists
- **L4:** Tuples, dicts, sets, functions
- **L5:** Errors, files, comprehensions
- **L6:** OOP, modules, packages
- **L7:** Advanced OOP, testing, debugging, TDD
- **L8:** Web APIs with FastAPI, DI, DTOs

Data, ML & Generative AI (L9-L12)

- **L9:** AI/ML/DS taxonomy, pandas, Polars
- **L10:** Visualization (Matplotlib, Seaborn, Plotly)
- **L11:** ML, computer vision, PyTorch
- **L12:** LLMs, Ollama, streaming, RAG

The red thread

Lectures 2-8 grow a tiny `to-do` app from 3 lines to a tested, packaged, web-served application. Lectures 9-12 then show what Python is uniquely good at: turning data into decisions and language into systems.



Semester Assessment

How your grade adds up

Component	Weight
Lab Warm-ups (Canvas)	20%
Midterm exam	20%
Endterm exam	20%
Programming assignment 1	20%
Programming assignment 2	20%

Attendance: max 3 absences allowed. Each lab has a Canvas warm-up exercise to complete.

Defending your assignments

- Both submitted assignments are **checked and defended** in person.
- You must be able to explain **every line** of code you submit.
- "I copied this from an AI and don't know what it does" is treated as a non-submission.

The contract: use any tool that helps you **learn** faster. Submit only what you **understand**. We come back to this in detail in the AI section of this lecture.



Checkpoint: About This Course

Q1

How many lab absences are allowed before it affects your grade?

A) Unlimited, as long as you watch recordings

B) 3

C) 0 - every lab is mandatory

D) Only the labs that fall on Tuesdays

Show Answer

Q2

What happens if you submit code you cannot explain during the assignment defense?

A) Nothing — submission alone is enough

B) You get half credit because at least it runs

C) The submission is treated as if you did not submit it

D) You get a small bonus for being honest about it

Show Answer



Why Python in 2026?

A short history, the language's character, and where it dominates today



A Short History

From a Christmas hobby project to the world's most-taught language

- **1989:** Guido van Rossum starts Python as a holiday project at CWI (Amsterdam).
- **1991:** Python 0.9.0 released to a small Usenet audience.
- **2000:** Python 2.0 (list comprehensions, garbage collection).
- **2008:** Python 3.0 - intentionally backward-incompatible cleanup.
- **2020:** Python 2 reaches end-of-life. Python 3 is the only Python.
- **2025:** Python 3.14 released (Oct) - free-threaded interpreter, experimental tail-call JIT.
- **2026:** Python tops virtually every "most popular" survey for the 4th year in a row.

Fun fact #1 - the name

Python is **not** named after the snake. Guido picked it while reading scripts of **Monty Python's Flying Circus** over Christmas 1989 - he wanted "something short, unique, and slightly mysterious." The snake imagery is a happy accident the community ran with.

Why CS students should care

Python is the **lingua franca** of modern data and AI. Almost every paper, library, and tutorial in lectures 9-12 ships Python first.



Main Features & the Zen of Python

The character of the language

- High-level, dynamically typed, **multi-paradigm** (imperative, functional, OOP)
- Interpreted (CPython), but executables and JIT exist
- **Indentation** defines blocks - no curly braces
- Case-sensitive; keywords are lowercase
- Newline ends a statement; `:` opens a block
- Cross-platform: Windows, macOS, Linux (and embedded)

```
>>> import this
```

The Zen of Python, by Tim Peters

Beautiful **is** better than ugly.

Explicit **is** better than implicit.

Simple **is** better than **complex**.

Complex **is** better than complicated.

Readability counts.

... (19 aphorisms **in** total)

Fun fact #2 - the Easter eggs

Type `import this` in a Python REPL and you get the 19-line Zen of Python by Tim Peters. Type `import antigravity` and Python opens an XKCD comic in your browser. Try both today.



Application Areas

Data Science

NumPy, pandas, Polars, Matplotlib, Seaborn, Plotly - covered in lectures 9-10.

ML & Deep Learning

scikit-learn, PyTorch, TensorFlow, JAX - lecture 11.

Generative AI

Transformers, Ollama, LangChain, vLLM - lecture 12.

Web & APIs

FastAPI, Flask, Django, httpx - lecture 8.

Scientific Research

SciPy, SymPy, Astropy, Biopython - powering everything from CERN to climate models.

Automation & DevOps

Glue scripts, CI/CD, Ansible, infrastructure as code.

Fun fact #3 - black holes run on Python

The first ever image of a black hole (M87, 2019) was assembled by the Event Horizon Telescope team using **Python** - NumPy, SciPy and Matplotlib were core to the imaging pipeline. The same stack you'll learn in lectures 9-10.



Why Python Won Data Science (in One Slide)

The four ingredients

1. **Notebooks (Jupyter):** read-eval-plot in seconds, perfect for exploration.
2. **NumPy / pandas / Polars:** fast vectorized arrays and dataframes that wrap C/Rust.
3. **A glue language:** wraps native libraries (BLAS, CUDA, ONNX, Arrow) without ceremony.
4. **A community network effect:** every new model, dataset, paper, and course ships Python bindings first.

"Glue language" in 2026 means

You write 20 lines of readable Python and let the libraries do the heavy lifting in C++ / Rust / CUDA. The productivity-to-performance ratio is unmatched for research and prototyping.

What it does **not** mean

Python is not the fastest language line-for-line. For tight CPU-bound loops, you reach for vectorized NumPy/Polars, Cython/Numba, Rust extensions, or move the hot path into a compiled library.



Checkpoint: Why Python

Q3

Where does the name "Python" come from?

- A) The snake species
- B) A Greek mythology creature Guido found in a CS textbook
- C) Monty Python's Flying Circus
- D) A typo of "Pylon" that nobody dared to fix

Show Answer

Q4

Why is Python so dominant in data science despite being slower than C++?

- A) Because it secretly compiles to C++ at runtime
- B) Because notebooks + vectorized libraries (NumPy, pandas, PyTorch) wrap fast native code, and the productivity beats the speed gap
- C) Because nobody has written equivalent libraries in any other language yet
- D) It is not - Excel is dominant; Python just has better PR

Show Answer



Installing Python the 2026 Way

From `python.org` to a clean virtual environment with `uv`

Get Python on Your Machine

The official path

- Download from `https://www.python.org/downloads/` for Windows / macOS / Linux.
- On macOS: also fine via `brew install python@3.13`.
- On Linux: distribution-managed Python is fine for system tools, but use a per-project venv for your own code.

Which version in 2026?

- **Python 2.x:** end-of-life since 2020 - never start a new project here.
- **Currently supported:** 3.12, 3.13, 3.14.
- **3.14 (Oct 2025):** free-threaded interpreter, experimental tail-call JIT, t-strings.
- For this course we target **3.12+** so all our demos run identically.

```
$ python3 --version  
Python 3.13.2
```

```
$ which python3  
/usr/local/bin/python3
```

The command may be `python` (Windows) or `python3` (macOS / Linux). When you read tutorials online, mentally translate.



Virtual Environments: One Per Project

```
# standard library (always available)
$ python3 -m venv .venv
$ source .venv/bin/activate          # macOS / Linux
$ .venv\Scripts\Activate.ps1        # Windows PowerShell

(.venv) $ python -m pip install requests
(.venv) $ python -c "import requests; print(requests.__version__)"
2.32.3
```

A virtual environment is just a folder of symlinks. Activate it for the shell, install packages into it, then deactivate. Never share envs across projects.

Why bother?

- Project A needs `numpy 2.x`, project B is pinned to `numpy 1.26` - no conflict.
- You can delete the env (`rm -rf .venv`) and rebuild from a lockfile - perfect for CI.
- The system Python stays clean; nothing you install can break `/usr/bin/python3`.

Course rule: every Python project in this course gets its own venv. We commit `pyproject.toml` + lockfile, never the `.venv/` folder.



Package Management: `pip`, `conda`, and Why We Use `uv`

```
# the historical default
$ pip install pandas matplotlib

# conda - reach for it only for system
# libraries (CUDA, GDAL, ...)
$ conda install -c conda-forge gdal

# the modern Rust-based default
$ uv venv # creates .venv
$ uv add pandas matplotlib # pyproject + lock
$ uv run python script.py # auto-activates
```

All three install packages. They differ in speed, ecosystem, and how strictly they pin versions.

Comparison

Tool	Use when
<code>pip</code>	One-off scripts, legacy projects
<code>conda</code> / <code>pixi</code>	CUDA, GIS, scientific stacks
<code>uv</code>	Default for this course

Fun fact #4 - speed of `uv`

`uv` (Astral, Rust) resolves & installs entire dependency trees in **under a second** - typically **10-100x faster than `pip`**. Same team makes the `ruff` linter (lecture 6).

Your First Python Program

```
# hello.py
x = 10
y = 20
print(f"x + y = {x + y}")
print("Welcome to the course!")
```

```
$ uv run python hello.py
x + y = 30
Welcome to the course!
```

Three ways to run it today

1. **The REPL:** `python3` → type the lines interactively.
2. **A file:** save as `hello.py` and run with `uv run python hello.py`.
3. **Online:** a browser-only sandbox (e.g. programiz.com/python-programming/online-compiler) when you have no local install.

Notice no semicolons, no `main()`, no `{ }`. Lecture 2 unpacks **why** Python looks like this and what the trade-offs are.



Checkpoint: Setup

Q5

You start a brand-new project that needs `pandas`. What is the recommended 2026 sequence in this course?

- A) `sudo pip install pandas` - fastest, gets it everywhere
- B) `uv venv` → `uv add pandas` → `uv run python ...`
- C) Copy-paste pandas source files into your project
- D) Email it to yourself as a zip every Friday

Show Answer

Q6

Which Python version is the minimum we target in this course's demos?

- A) Python 2.7 - it's still on macOS
- B) Python 3.6
- C) Python 3.12
- D) Whatever ChatGPT recommends today

Show Answer



Development Environments

From vi to AI-native IDEs - pick the one that helps you focus

Editors and IDEs



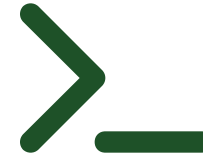
VS Code

Microsoft. Free. The most-used Python editor in the world; massive extension marketplace.



PyCharm

JetBrains. Free Community edition + paid Pro. Heaviest, but the deepest refactoring + debugger.



Vim / Neovim

Ancient, modal, terminal-native. With `pyright` + `ruff` LSPs it competes with the GUI giants.

For this course, any of these works. We'll show snippets in **Cursor** (which is a fork of VS Code) so VS Code users feel at home.

JupyterLab: The Notebook Workhorse



What it is

A browser-based environment that mixes Python **cells**, Markdown notes, and rich output (charts, tables, LaTeX) in one document (`.ipynb`).

Why we use it from lecture 9 onward

- Read CSV → chart → conclusion in seconds
- Cells let you re-run only the part you changed
- Notebooks render perfectly on GitHub for handing in

Browser-only options: Google Colab and CoCalc run JupyterLab in the cloud with free GPUs - useful when your laptop has no install access.



AI-Native IDEs: Cursor and Claude Code



Cursor

Fork of VS Code with an LLM in every keystroke.

- **Tab:** predicts your next edit, multi-line
- **Cmd-K:** inline edit with a natural-language prompt
- **Chat / Agent:** ask questions or let it run tools across files
- Free student plan via GitHub Education



Claude

Claude Code

Terminal-native AI coding agent from Anthropic.

- Lives in your terminal; works on any project tree
- Plans, edits, runs tests, and reads logs end-to-end
- Best when you describe a goal, not a snippet
- Pairs naturally with VS Code, vim, and JupyterLab

Both are **augment** tools. In this course we use Cursor for the slides and demos. You can use whatever you like at home, as long as you can defend your code (Q2 rule).



AI-Assisted Development

Honest expectations, sensible workflow, and good prompting



The 2026 Landscape: Tools, Modes, and Roles

Three families of tools

- **Inline assistants:** GitHub Copilot, Cursor Tab, Codeium - tab to complete, Cmd-K to edit
- **Agentic IDEs / CLIs:** Cursor Agent, Claude Code, Aider - plan + execute multi-file tasks
- **Chat assistants:** ChatGPT, Claude, Gemini - explain, debug, brainstorm in a side window

Most professional setups in 2026 mix **two**: an inline tool for typing speed and an agent for "do this whole task".

Common interaction modes

Mode	What it does
Ask	Q&A in the side panel - read-only
Edit	Modify selected code with a prompt
Plan	Sketch the approach first, no edits
Agent	Read, edit, run, and observe across files
Review	Critique a diff before commit

Rule of thumb: the more autonomy you grant, the more carefully you must read the diff before saving.



Limitations and Risks

Hallucinations

AI invents plausible-looking APIs, library names, and signatures - especially for newer or domain-specific code.

Security flaws

SQL injection, hardcoded secrets, missing input validation - AI happily produces working-but-vulnerable code.

Bias & antipatterns

Trained on public code (good and bad). It will faithfully reproduce a 12-year-old anti-pattern from Stack Overflow.

Skill atrophy

If you only ever accept suggestions, you never build the mental models that let you debug at 2 AM.

Privacy

Pasting credentials, internal data, or proprietary code into a hosted LLM is a leak. Don't.

Stale knowledge

Training cutoffs mean the model often does not know about your library's latest version (e.g. pandas 3.0, Pydantic v2).

Fun fact #5 - the real numbers

GitHub's 2025 developer surveys report that **~75%** of professional developers use an AI assistant daily, but only **~30%** of Copilot suggestions are accepted - and **most accepted lines are still edited** before the commit lands. AI is a strong first draft, not a finished product.

Effective Prompting for Code

Four prompting habits

1. **Be specific** about inputs, outputs, and constraints.
2. **Give context:** function signature, types, sample data, the file's purpose.
3. **Ask for explanations** alongside the code so you can verify the why.
4. **Iterate** - the second prompt is almost always better than the first.

Vague vs specific - same task

Vague:

```
Write a function to process data.
```

Specific:

```
Write a Python function that takes a list of dicts with 'name' and 'score' keys, filters scores > 80, and returns a sorted list of names. Add type hints and a docstring with one example.
```

Same model, same minute - the second prompt produces code you can drop straight into a test.



Live Demo: Recursive vs Iterative Fibonacci

Prompt to the assistant in class:

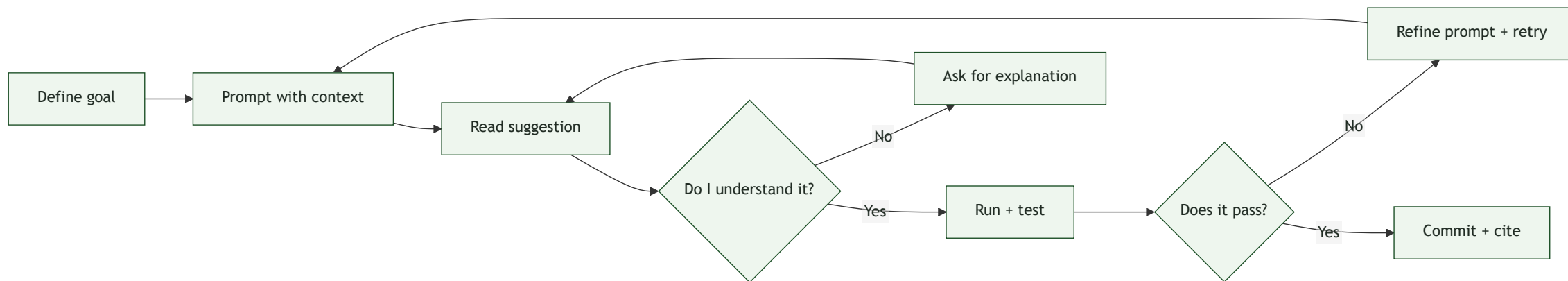
```
Write two Python functions:  
1) fib_recursive(n) - the textbook  
   recursive version  
2) fib_iterative(n) - an O(n) loop  
Add type hints, then time both for  
n = 30 with timeit.
```

```
from timeit import timeit  
  
def fib_recursive(n: int) → int:  
    if n < 2: return n  
    return fib_recursive(n - 1) + fib_recursive(n - 2)  
  
def fib_iterative(n: int) → int:  
    a, b = 0, 1  
    for _ in range(n):  
        a, b = b, a + b  
    return a  
  
print(timeit(lambda: fib_recursive(30), number=10))  
print(timeit(lambda: fib_iterative(30), number=10))
```

In-class prediction: by what factor is the iterative version faster for `n = 30`? (Hint: recursive is $\sim O(2^n)$, iterative is $O(n)$.)
We'll let the AI write it, then run it together and see if the explanation it gave us was honest.



The AI Pair-Programming Loop



The two checkpoints ("**do I understand**" + "**does it pass**") are what make this safe to use during your assignments. Skipping either is how you fail your defense.



Checkpoint: Working with AI

Q7

The AI tool autocompletes a function that calls `pandas.DataFrame.iteritems()`. You run it and get an `AttributeError`. The most likely cause?

- A) Your computer is broken
- B) The model's training data is older than the version of pandas you have installed (it was renamed/removed)
- C) Pandas only works on Tuesdays
- D) You forgot to `sudo` the import

Show Answer

Q8

Which prompt is more likely to produce code you can ship?

- A) "Make me a sorting thing"
- B) "Sort some data, you know which kind"
- C) "Write a Python function `sort_users(users: list[User]) → list[User]` that returns users sorted by `.signup_date` descending; include a docstring and one doctest"
- D) "Just guess what I want"

Show Answer



Ethical & Academic Use of AI

The course rules - read once, follow always



Ethics: Plagiarism, Quality, Responsibility

Avoiding plagiarism

- Models are trained on open-source code - large generated blocks may quote real projects without attribution.
- Strive to keep **your** reasoning visible: write the structure, ask the AI for the details.
- When you keep an AI snippet verbatim, **cite it** in a comment.

Quality & responsibility

- You sign off on every line you submit - the AI does not.
- Review, run, and test the code before it leaves your editor.
- Security and correctness are **your** job, not the model's.

Treat AI like a fast, opinionated junior pair-programmer: helpful, sometimes wrong, never accountable.

Course AI Policy



You may

- Use AI tools as learning aids, in any lecture or assignment
- Ask the AI to explain a concept, refactor your code, or generate test cases
- Use Cursor / Copilot / Claude during the labs

You must

- Understand and be able to explain every line you submit
- Disclose AI assistance in your assignment write-up
- Practice writing code without AI regularly to build core skills

You must not

- Submit AI-generated code as your own without disclosure - that is academic dishonesty
- Paste secrets, exam content, or other students' work into AI tools
- Bring AI assistants to the midterm / endterm exam

The rule of thumb

If you used AI, be transparent. If you cannot defend it, do not submit it. **AI should accelerate your learning, not replace it.**



Checkpoint: Ethics & Policy

Q9

You used Cursor to draft a parser for assignment 1, then heavily edited it. What does the course policy require?

- A) Nothing - if you edited it, it's yours
- B) Disclose AI assistance in the write-up **and** be ready to defend every line
- C) Re-write the whole thing without any tools to be safe
- D) Email the original prompt to the lecturer

Show Answer

Q10

Which of the following is **not** allowed?

- A) Asking ChatGPT to explain Python `decorators` to you
- B) Using Cursor Tab during a lab warm-up
- C) Bringing Claude Code to the midterm exam
- D) Citing a generated function with a comment in your assignment

Show Answer



Python's Grand Tour

A 4-slide preview of where the next 11 lectures take you



Foundations & Flow (Lectures 2-3)

What you will learn

- Data types, variables, expressions (L2)
- Booleans, comparison & logic (L3)
- Flow control: `if` / `while` / `for`
- Strings (immutable, sliceable) and lists (mutable, ordered)

The to-do app is born here: from `x = 10` to a list of tasks the user can iterate over.

```
tasks = ["learn syntax", "write loops", "ship a list"]
for i, t in enumerate(tasks, start=1):
    status = "DONE" if t.startswith("learn") else "TODO"
    print(f"{i:>2}. [{status}] {t}")
```

```
# Output:
# 1. [DONE] learn syntax
# 2. [TODO] write loops
# 3. [TODO] ship a list
```

10 lines that touch lists, strings, f-strings, conditionals, loops, and built-ins. Welcome to Python's density.



Structures, Errors, OOP & the Web (Lectures 4-8)

What you will learn

- Tuples, dicts, sets, functions & scope (L4)
- Errors, file I/O, comprehensions (L5)
- Classes, modules, packaging (L6)
- Inheritance, testing, debugging, TDD (L7)
- FastAPI, REST, dependency injection (L8)

The to-do app grows up: classes, persistence, tests, and finally a JSON HTTP API you can curl.

```
from fastapi import FastAPI
from pydantic import BaseModel

class Task(BaseModel):
    title: str
    done: bool = False

app = FastAPI()
TASKS: list[Task] = []

@app.post("/tasks")
def add_task(task: Task) → Task:
    TASKS.append(task)
    return task

@app.get("/tasks")
def list_tasks() → list[Task]:
    return TASKS
```

A complete REST API in 14 lines. Auto-generated OpenAPI docs included.



The Data Stack (Lectures 9-10)

What you will learn

- AI / ML / DS taxonomy and CRISP-DM
- JupyterLab, pandas 3.0, Polars (L9)
- Matplotlib, Seaborn, Plotly, MapLibre (L10)
- Loading, cleaning, joining, grouping, plotting

The point is not the libraries. It's the **workflow**: question
→ table → chart → decision.

```
import pandas as pd

missions = pd.read_csv("space_missions.csv")
top = (missions
       .query("destination == 'Moon'")
       .nlargest(5, "cost_million_usd")
       [["mission_name", "agency", "cost_million_usd"]])
print(top)
```

#	mission_name	agency	cost_million_usd
# 0	Apollo 11	NASA	25400
# 1	Apollo 13	NASA	25400
# 2	Artemis II	NASA	4500
# 3	Artemis I	NASA	4100
# 4	Apollo 8	NASA	2500

Six lines: load → filter → rank → report. This is what lecture 9 demonstrates live.



Machine Learning & Generative AI (Lectures 11-12)

What you will learn

- scikit-learn pipelines, KNN, K-Means (L11)
- Computer vision & PyTorch foundations (L11)
- LLMs: tokens, sampling, temperature (L12)
- Local inference with Ollama, streaming, RAG (L12)

Fun fact #6 - the closing loop

PyTorch (Python on top of C++/CUDA) is the framework behind almost every flagship open LLM since GPT-3 - Llama, Qwen, Mistral, Gemma. The same Python you write in lecture 2 powers the model you'll prompt in lecture 12.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

X, y = load_iris(return_X_y=True)
X_tr, X_te, y_tr, y_te = train_test_split(X, y, random_state=0)

model = KNeighborsClassifier(n_neighbors=3).fit(X_tr, y_tr)
print(f"accuracy = {model.score(X_te, y_te):.3f}")

# accuracy = 0.974
```

A complete ML pipeline in 7 lines: load, split, fit, score.
Lecture 11 shows what happens after this baseline.



Lecture 1 Summary

What we set up

- Python 3.12+ via `uv`
- One venv per project
- An IDE that works for you (VSCode / PyCharm / Cursor)
- JupyterLab for the data half of the course

What we agreed on

- AI tools are **welcome** as learning aids
- Disclose & defend every submitted line
- Build core skills **without** AI in parallel
- Exams remain AI-free

What's next

- **Lecture 2:** data types, variables, expressions, the to-do app starts
- Lab warm-up on Canvas this week
- Bring a laptop with Python + your favorite IDE

The real skill of this course is not memorizing syntax. It is building a **workflow**: install once, use a venv, write small testable code, ask AI well, defend everything you submit.



Thank You for Your Attention!

"Programs must be written for people to read, and only incidentally for machines to execute." — Harold Abelson

Setup

Tooling

AI policy

Grand Tour

See you in Lecture 2.

Python Course 2025/26/2

Lecture 2: Python Basics



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Agenda

- What a computer does, and how algorithms become programs
- How Python executes source code in 2026 (CPython, bytecode, PVM)
- Objects, types, and identity: the working mental model
- Operators, precedence, and everyday expression gotchas
- Variables as bindings, plus strings, `f" ... "`, and basic I/O
- Live coding: a small to-do app grown in three iterations

Learning objective

Move from "I have heard of Python" to writing small, correct Python programs with a clear mental model of what runs, what objects exist, and what a name actually refers to.

By the end, you can

Explain how CPython runs your code, reason about types and expressions, read and write readable Python assignments and f-strings, and grow a short script feature by feature.

CPython

types

expressions

variables

strings

live coding

Lecture 2 is the bridge from "what is Python?" to writing Python confidently.



Programming Fundamentals

What computers do, and how we describe solutions

What Does a Computer Do?

Learning objective

See the big picture before the syntax: a computer is a fast, precise instruction follower, and a program is just a careful sequence of those instructions.

By the end of this section, you can

- Explain the difference between an algorithm and a program
- Describe what happens at machine level vs high-level code
- Translate a short algorithm into Python statements

🔧 Core capabilities

- Executes instructions **extremely** fast
- Stores intermediate and final results in memory
- Repeats operations with near-perfect precision
- Only does **exactly** what we say, nothing more

</> Instruction levels

- **Machine code** fixed CPU operations (add, load, jump, ...)
- **High-level code** what we write in Python



Algorithm vs Program

Algorithm	Program
Language-independent recipe for solving a problem	Concrete implementation in a programming language
Answers WHAT to do	Answers HOW to do it in code
Independent of Python, C, or JavaScript	Depends on language, runtime, operating system

Algorithm = the cooking recipe. Program = the dish cooked by a specific kitchen using specific ingredients. Same recipe, many possible programs.

💡 **Fun fact** Guido van Rossum named the language "Python" in 1989 — not after the snake, but after **Monty Python's Flying Circus**. That is also why many Python docs use `spam` and `eggs` as example values instead of `foo` and `bar`.

Example: Calculate an Average

Algorithm steps

1. Read two numbers
2. Add them together
3. Divide the sum by 2
4. Display the result

Any language can implement this. The steps do not change.

Python program

```
number1 = 33
number2 = 55
sum_total = number1 + number2
average = sum_total / 2
print(f"Average = {average}")
```

Four named steps, one line each — algorithm and program line up 1:1.



Checkpoint: Programming Fundamentals

Q1

Which statement is correct?

- A) An algorithm is the same as one Python file
- B) A program is one concrete implementation of an algorithm
- C) Algorithms require a compiler
- D) Algorithms execute themselves

Show Answer

Q2

In the average example, which Python line implements step 3: "Divide the sum by 2"?

- A) `number1 = 33`
- B) `number2 = 55`
- C) `sum_total = number1 + number2`
- D) `average = sum_total / 2`

Show Answer



How Python Runs + Data Types

From source code to objects in memory



Compiler vs Interpreter

Learning objective

Understand how Python source text becomes running behavior, and why Python is called "interpreted" even though CPython compiles in the background.

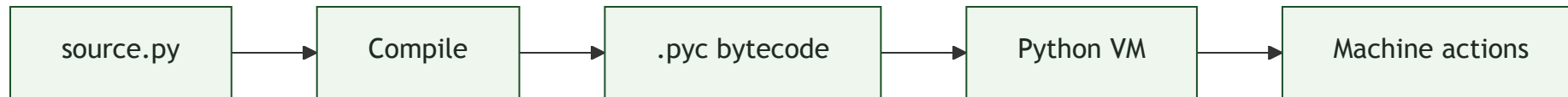
By the end of this section, you can

- Describe the CPython source → bytecode → PVM pipeline
- Tell scalar from non-scalar objects
- Use `type()` and `id()` to inspect values
- Cast between `int`, `float`, `str`, `bool` safely

Compiler path	Interpreter path
Translate the whole program before execution	Translate and execute progressively
Often produces a standalone executable	Runs source or bytecode at runtime
Examples: C, C++, Rust, Go	Examples: Python, Ruby, JavaScript

Python is classified as interpreted, but **CPython still compiles** your `.py` file to bytecode first. The line is blurrier than textbooks suggest.

Python Execution Flow



On repeated runs, cached bytecode in `__pycache__` is reused when the source has not changed — this is why re-running a script feels instant.



CPython in 5 Steps

1

Write `.py` source

2

Compile to bytecode

3

PVM reads
instructions

4

Runtime performs
operations

5

CPU executes
machine-level work

CPython is the reference implementation in C. Alternatives like `PyPy` (JIT), `Pyodide` (browser), and `MicroPython` (microcontrollers) all follow the same basic idea: compile, then run.

Data Objects in Python

Python Data Object Hierarchy

object (everything inherits here)

Scalar

Numbers
int, float,
complex

Boolean
bool

None
NoneType

Non-scalar

Sequences
str, list,
tuple,
bytes

Mappings
dict

Sets
set,
frozenset

Key ideas

- Every value in Python is an **object**
- `type(obj)` tells you what operations are valid
- **Scalar** = one atomic value (e.g., `42`)
- **Non-scalar** = collection/sequence of items (e.g., `[1, 2, 3]`)

```
type(42)           # <class 'int'>
type(3.14)         # <class 'float'>
type("hi")         # <class 'str'>
type([1, 2, 3])    # <class 'list'>
```

Scalar Types and Casting

Core scalar types

- `int`, `float`, `complex`
- `bool` (subclass of `int`)
- `NoneType` with the single value `None`

`True + True == 2` — booleans literally are integers.

Casting examples

```
float(3)      # 3.0
int(3.9)      # 3 (truncates toward zero)
str(42)       # '42'
bool(0)       # False
bool("")      # False
5 + 3.2       # 8.2 (implicit int → float)
```

`int()` truncates; for rounding use `round()` or `math.floor()`.

∞ **Fun fact** Python's `int` has **arbitrary precision** — there is no 32- or 64-bit limit. Try `2 ** 1000` in a cell and you'll get the exact 302-digit integer with no overflow. Most languages would silently wrap or crash.



Checkpoint: Execution and Types

Q3

Which statement about Python execution is correct?

- A) CPython compiles to bytecode, then the PVM executes it
- B) Python always produces a native executable first
- C) Bytecode exists only in Java
- D) Code runs by telepathy

Show Answer

Q4

What does `True + 2` evaluate to?

- A) `0`
- B) `1`
- C) `3`
- D) A philosophical warning

Show Answer



Expressions, Variables, and Strings

From operators to practical Python syntax



Operators on `int` and `float`

Learning objective

Read and write common Python expressions confidently — pick the right operator, anticipate the result type, and spot floating-point gotchas before they bite you in production.

By the end of this section, you can

- Use arithmetic and string operators correctly
- Predict operator precedence and add parentheses when needed
- Bind values to variables and re-bind safely
- Format output with `f" ... "` strings

Operator	Meaning	Example
<code>+</code> , <code>-</code>	Add / subtract	<code>7 - 2 → 5</code>
<code>*</code>	Multiply	<code>3 * 5 → 15</code>
<code>/</code>	Division (always <code>float</code>)	<code>5 / 2 → 2.5</code>
<code>//</code>	Floor division	<code>5 // 2 → 2</code>
<code>%</code>	Remainder (modulo)	<code>5 % 2 → 1</code>
<code>**</code>	Exponentiation	<code>2 ** 3 → 8</code>

⚠ Floating-point surprise `0.1 + 0.2 == 0.3` returns `False` in Python — and in almost every language that uses IEEE-754 floats. The real sum is `0.30000000000000004`. Compare floats with `math.isclose(a, b)`, never `==`.

Expressions Always Produce a Value

Expression pattern

`<object> <operator> <object>`

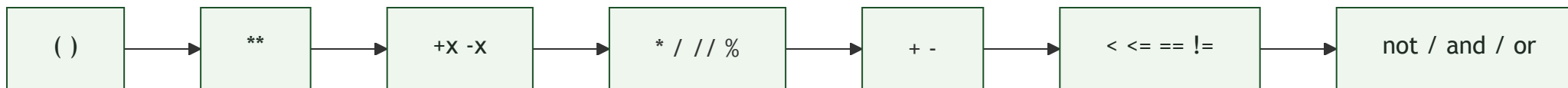
- `3 + 5` → `8`
- `3.14 * 9` → `28.26`
- `10 / 4` → `2.5`
- `"ab" + "cd"` → `'abcd'`
- `"ab" * 3` → `'ababab'`

Result type rule of thumb

- Both operands `int` → usually `int`
- Any `float` involved → result is `float`
- `/` in Python 3 **always** returns `float`
- Operators on strings behave completely differently from numbers



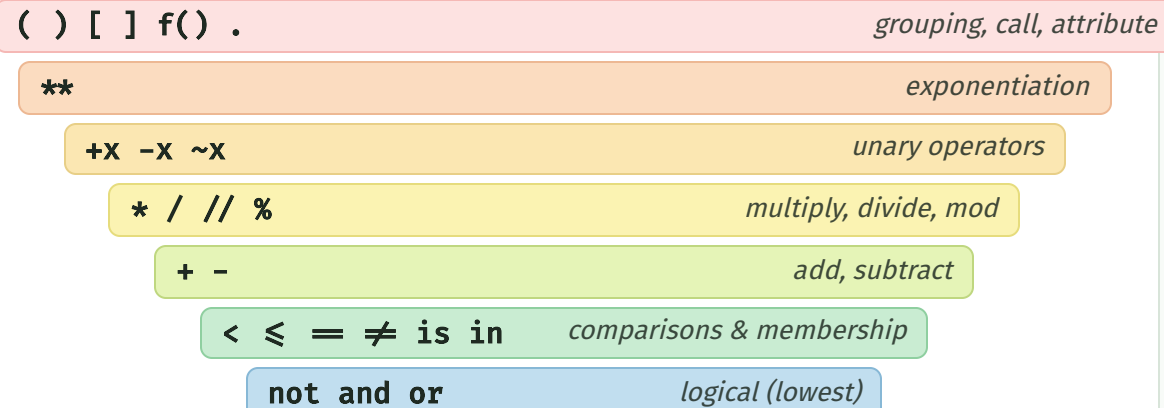
Operator Precedence



Python evaluates higher-precedence operators first. When in doubt, **add parentheses**: readability beats memorizing the table.

Precedence Pyramid and Edge Cases

Precedence (top = evaluated first)



Common gotcha

```
print(-2**2)    # -4  (** before unary -)
print((-2)**2)  # 4
print(2 + 3 * 4) # 14, not 20
print((2 + 3) * 4) # 20
```

Tip

When mixing arithmetic and comparisons, parenthesize. Future-you reading the code in a month will be grateful.

Variables Are Bindings

Core idea

- A **name** refers to an object in memory
- Assignment creates or updates that reference
- Inspect with `type()` and `id()`
- Two names can refer to the **same** object

```
a = 13234
print(type(a))  # <class 'int'>
print(id(a))    # object identity

b = a
print(a is b)   # True (same object)
```

`id()` is implementation-specific, but stable while the object lives.

🔄 **Fun fact** CPython pre-creates the small integers from `-5` to `256` at startup and reuses them. That is why `a = 100; b = 100; a is b` is `True`, but `a = 1000; b = 1000; a is b` is usually `False`. Moral: use `=` for equality, `is` only for identity.



Variable Naming and Assignment Patterns

Naming rules

- Letters, digits, and underscore only
- Cannot start with a digit
- Case-sensitive (Romeo $\neq$ romeo)
- Cannot be a keyword (if, while, class, ...)
- **PEP 8:** use snake_case for variables and functions

Useful patterns

```
x, y = 2, 3          # multiple assignment
x, y = y, x          # swap without a temp
total += 5           # augmented assignment
r = 2.5
area = 3.14159 * r**2
```

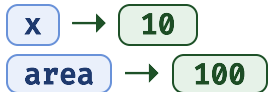
Prefer meaningful names: area > a, user_count > uc.

Re-binding in Memory



1 Initial binding

```
x = 10  
area = x * x # 100
```



2 Re-binding x

```
x = 20 # x points elsewhere
```



3 Derived value is stale

```
print(area) # 100, NOT 400
```

Reassigning `x` does **not** recompute `area`.
You must run the formula again to update derived values.

Strings, Dynamic Typing, and Basic I/O

Strings and f-strings

```
hi = "hello there"
name = "Anna"
greeting = f"{hi}, {name}!"
print(len(greeting))      # 17
print("line1\nline2")     # two lines
print(f"pi = {3.14159:.2f}") # pi = 3.14
```

Strings are immutable — methods like `upper()` return a new string.

Input and type conversion

```
raw = input("Age: ")      # always returns str
age = int(raw)             # convert explicitly
print(f"You are {age} years old")
```

- `input()` always returns `str`
- Convert with `int()` / `float()` at the boundary
- Comments explain why, not what the syntax does

⚡ **Fun fact** f-strings arrived in Python 3.6 (2016, PEP 498) and quickly became both the **fastest** and most readable way to format strings. **PEP 701** (Python 3.12, 2023) went further: you can now freely nest quotes and write multi-line expressions inside an f-string — so `f"{data["key"].upper()}"` finally just works.

Checkpoint: Expressions, Variables, Strings



Q5

Which pair is correct?

A) `-2**2` → -4, `(-2)**2` → 4

B) `-2**2` → 4, `(-2)**2` → 4

C) Both produce -4

D) Both depend on the weather forecast

Show Answer

Q6

What does this program print for the last line?

```
x = 10
area = x * x
x = 20
print(area)
```

A) 20

B) 100

C) 400

D) Python recomputes `area` automatically

Show Answer



Live Coding: To-Do App

From hardcoded data to interactive input, one iteration at a time



Demo Kickoff: To-Do App

Learning objective

Grow a working Python script in small, verifiable steps — each iteration stays runnable and teaches one new idea (types, averaging, user input).

By the end of this demo, you can

- Store typed data in meaningful variables
- Compute ratios and averages with the right operator
- Accept and convert user input safely
- Format readable output with f-strings

Goal

Build a tiny CLI that helps a student estimate task duration and check how accurate the estimate was.

- Track task descriptions
- Track planned vs actual durations
- Compute estimation accuracy

Mindset

Iterate feature by feature. Each step must run before we move to the next one.

Iteration 1: One Task

Feature goals

- Store one task description
- Store number of tasks (starts at 1)
- Store planned and actual duration in minutes
- Compute estimation accuracy

Starter code

```
task = "Study Python basics"
number_of_tasks = 1
planned = 60
actual = 50
accuracy = planned / actual    # 1.2 means finished 20% faster
print(f"Task: {task}")
print(f"Accuracy: {accuracy:.2f}")
```



Iteration 2: Two Tasks and Average

Feature goal

Extend the app to two tasks and calculate their average duration.

- Add a second description and duration
- Compute `avg_duration`
- Notice how the code starts to repeat — foreshadowing lists & loops in Lecture 3

Mini implementation

```
dur1 = 50
dur2 = 30
avg_duration = (dur1 + dur2) / 2
print(f"Average: {avg_duration} minutes")
```

Parentheses are required here: without them, `dur1 + dur2 / 2` would compute `dur1 + (dur2 / 2)`.



Iteration 3: User Input and Summary

Feature goals

- Replace hardcoded values with `input()`
- Convert input with `int()` immediately
- Print a task-list summary
- Calculate average estimation accuracy

Key pattern: convert at the boundary

```
actual = int(input("Actual minutes: "))
planned = int(input("Planned minutes: "))
accuracy = planned / actual
print(f"Accuracy: {accuracy:.0%}")
```

Data enters as `str`; we immediately turn it into typed values we can compute with.

☠ **Security history fact** In Python 2, `input()` literally called `eval()` on whatever the user typed — so a classroom demo like this could execute arbitrary code from the prompt. Python 3 quietly fixed that: `input()` now always returns a `str`, and Python 2 `raw_input()` became the new default.



Checkpoint: To-Do App Iterations

Q7

What should we do with values coming from `input()` before using them in arithmetic?

- A) Convert them immediately with `int()` or `float()`
- B) Hope for the best and try `str + int`
- C) Pass them unchanged through Slack first
- D) Rename the variable

Show Answer

Q8

Why is dividing two `input()` results directly usually broken?

- A) `/` on two strings raises `TypeError`
- B) Python silently rounds everything to `0`
- C) Division only works on weekends
- D) It depends on CPU brand

Show Answer



Lecture 2 Summary

Execution

- `.py` → bytecode → PVM
- CPython is interpreted and compiled
- `__pycache__` makes reruns fast
- Every value is an `object`

Core Syntax

- Expressions produce typed values
- Precedence matters (`-2**2` → `-4`)
- Variables are **bindings** to objects
- Compare floats with `math.isclose()`, not `=`

Practical Skills

- Use meaningful `snake_case` names
- Convert `input()` immediately
- Prefer f-strings for output
- Grow scripts iteration by iteration

Next step (Lecture 3): these foundations become **control flow, strings, and lists** — the bread and butter of every Python program.



Thank You for Your Attention!

"There are only two hard things in Computer Science: cache invalidation and naming things."

— Phil Karlton

algorithms

execution model

types

expressions

variables

strings

Try `import this` in a Python REPL for the Zen of Python — it's been built in since 2004.

Python Course 2025/26/2

Lecture 3: Flow, Strings, and Lists



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Topics

- Logic operators and comparison
- Flow control with `if / elif / else`
- Loops with `while` and `for`
- Strings: indexing, slicing, immutability
- Lists: operations and practical usage

Session Format

Concepts

Examples

Live Coding

We will build from core language building blocks to a mini command-based To-do app.



Logic and Comparison

Boolean thinking for decisions in code



Logic Operators on Booleans

A and B are Boolean values.

- not A: True if A is False
- A and B: True only if both are True
- A or B: True if at least one is True

Operator precedence: not > and > or.

Truth Table

A	B	A and B	A or B	not A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False



Truthy, Falsy, and Short-Circuiting

Falsy values (common)

False, 0, 0.0, "", [], None

Everything else is usually truthy.

```
items = []
if items:
    print("has items")
else:
    print("empty")
```

```
name = ""
```

```
# short-circuit: second part is skipped
if name and name[0] == "A":
    print("Starts with A")
```

Fun Python fact

and and or return one of the original operands.

```
0 or 42 # 42
"hello" and "world" # "world"
```



Comparison Operators

Comparison expressions evaluate to `True` or `False`.

Operator	Meaning	Example	Result
<code>=</code>	Equal to	<code>5 = 5</code>	<code>True</code>
<code>≠</code>	Not equal to	<code>True ≠ False</code>	<code>True</code>
<code>></code>	Greater than	<code>7 > 10</code>	<code>False</code>
<code><</code>	Less than	<code>"Ann" < "Bob"</code>	<code>True</code>
<code>≥</code>	Greater or equal	<code>5.0 ≥ 5.0</code>	<code>True</code>
<code>≤</code>	Less or equal	<code>4 ≤ 2</code>	<code>False</code>
<code>in</code>	Membership test	<code>"py" in "python"</code>	<code>True</code>
<code>is</code>	Identity (same object)	<code>x is None</code>	<code>True/False</code>

Strings compare lexicographically. Cross-type equality usually returns `False` (for example: `'3' = 3`).
Use `is` for identity checks (especially `is None`), and `=` for value equality. Chained comparisons are valid: `0 < x < 10` .



Checkpoint: Logic and Comparison

Q1

`not not not True` = ?

A) True

B) False

C) SyntaxError

D) Existential crisis

Show Answer

Q2

`True + True` = ?

A) True

B) 2

C) TypeError

D) TrueTrue

Show Answer

Q3

`"banana" > "apple"` = ?

A) True

B) False

C) TypeError

D) Depends on ripeness

Show Answer



Flow Control

From straight-line code to decisions and repetition

Why Flow Control?

Straight-line execution

- Runs top to bottom in fixed order
- Always follows the same path
- Limited for real-world problem solving

```
x = 5
y = x + 2
print(y) # always prints 7
```

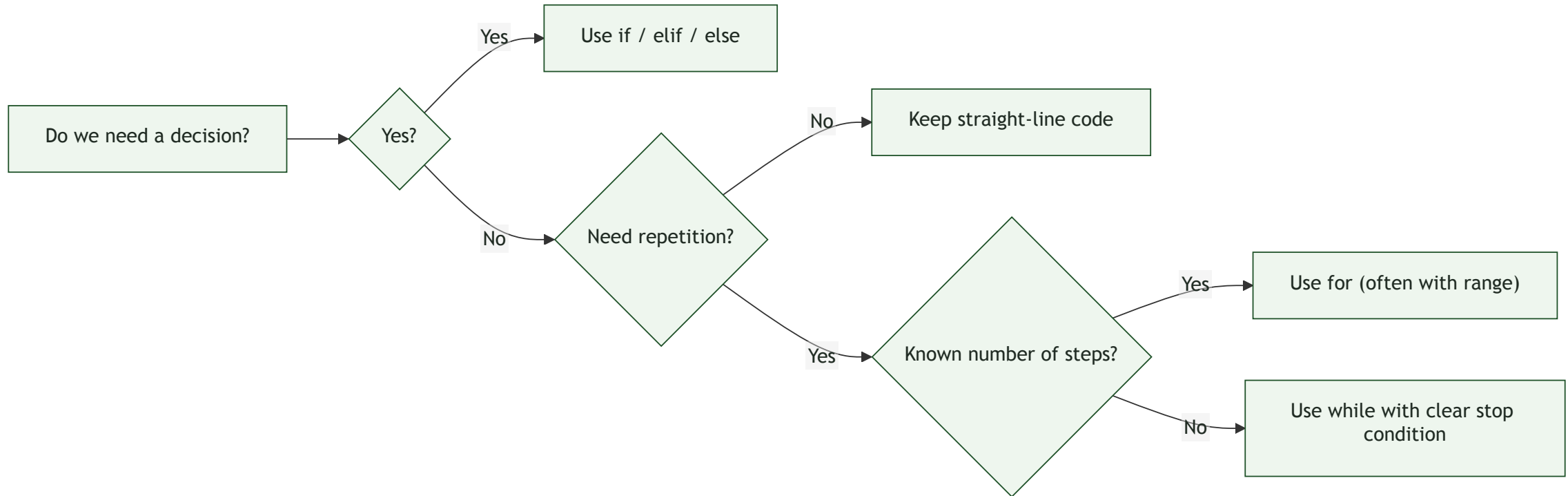
Real programs need:

Decisions

Repetition



Flow Control Decision Guide



Quick rule: choose `for` when you iterate over known data, and `while` when logic decides when to stop.

Branching with if

Use `if` to execute a block only when a condition is true.

```
x = 10

if x > 5:
    print("x is large")
```

Pattern

```
if <condition>:
```

Indented block executes when condition is True.



If...else

Add `else` to define the "otherwise" path.

```
age = 16

if age ≥ 18:
    print("Adult")
else:
    print("Minor")
```



If...elif...else

Handle multiple mutually exclusive conditions.

```
score = 85

if score ≥ 90:
    grade = "A"
elif score ≥ 75:
    grade = "B"
else:
    grade = "C"
```

`elif` is evaluated only if the previous conditions were False.

Importance of Indentation

```
age = 16

if age ≥ 18:
    print("Adult")
    student = False
else:
    print("Minor")
```

Why it matters

- Defines code blocks in Python
- Keeps conditional logic readable
- Incorrect indentation causes syntax errors
- Use 4 spaces (PEP 8); avoid mixing tabs/spaces

Indentation vs Scoping



Java: braces create block scope

```
int age = 20;

if (age ≥ 18) {
    String status = "adult";
}

System.out.println(status); // compile-time error
```

Variables declared inside `{ }` are not visible outside that block.

Python: indentation defines blocks

```
age = 20

if age ≥ 18:
    status = "adult"

print(status) # "adult"
```

Indentation groups statements, but `if` does not create a new scope.

In Python, `if`, `for`, and `while` blocks do not create new scopes. New scopes are introduced by functions, classes, and modules.

Checkpoint: Flow Control



Q4

How many times does this print "hi" ?

```
while False:  
    print("hi")
```

- A) 0
- B) 1
- C) Infinite
- D) Depends on mood

Show Answer

Q5

What happens if you indent with 3 spaces?

- A) SyntaxError
- B) It works
- C) Always IndentationError
- D) TabError

Show Answer



Loops

Repeat work cleanly and safely

Loops in Python



Main loop types

- `while`: repeat while condition is true
- `for`: iterate over sequence/iterable

Why loops?

- Avoid copy-paste repetition
- Process data item by item
- Retry/search/countdown patterns



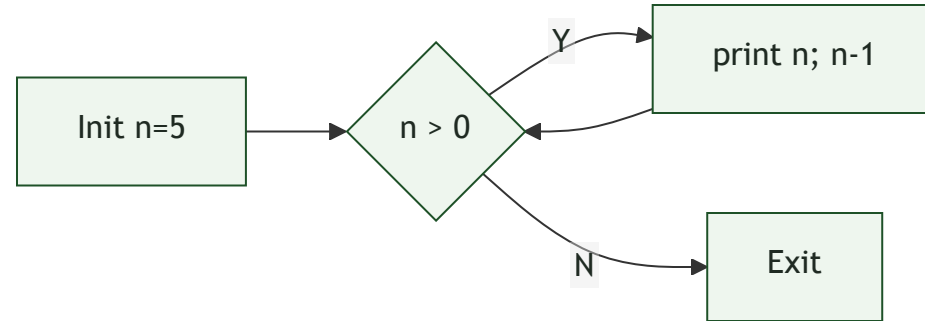
While Loop

`while` runs its block as long as the condition remains true.

```
n = 5
while n > 0:
    print(n)
    n -= 1
```

Always make progress toward ending the condition.

While Loop Flow



The condition is rechecked after each iteration until it becomes false.



Infinite Loop

```
while True:  
    print("I'm stuck in an infinite loop")
```

```
# Stop with Ctrl + C
```

Useful only when you include a clear exit strategy (break, event, signal).



For Loop

`for` iterates over iterable values such as strings, lists, and tuples.

```
for letter in "Python":  
    print(letter)  
print("Done")
```

While and For with Numbers



While version

```
n = 0
while n < 5:
    print(n)
    n += 1
```

For version

```
for n in range(5):
    print(n)
```

Use `for` when iteration count is known.

Range, Break, and Continue

range()

range(start, stop, step)

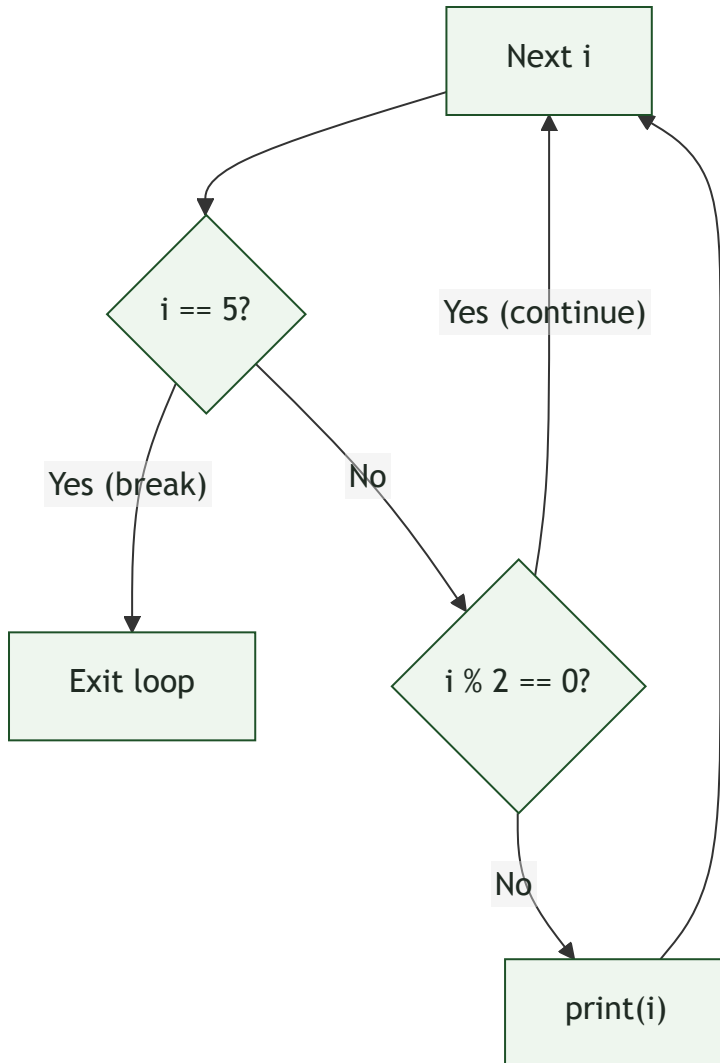
```
print(list(range(5)))      # 0..4
print(list(range(2, 8)))   # 2..7
print(list(range(0, 20, 3))) # 0,3,6, ...
```

Loop control

```
for i in range(10):
    if i == 5:
        break
    if i % 2 == 0:
        continue
    print(i)
```

for ... else exists too: else runs only if no break happened.

Break and Continue



```
for i in range(10):  
    if i == 5:  
        break  
    if i % 2 == 0:  
        continue  
    print(i)
```

Output: 1, 3

Loop Variable Scoping

Java: loop var stays in loop scope

```
for (int i = 0; i < 3; i++) {  
    System.out.println(i);  
}  
System.out.println(i); // compile-time error
```

`i` is declared in the `for` header and is not visible after the loop.

Python: loop var remains accessible

```
for i in range(3):  
    print(i)  
  
print(i) # 2
```

`i` stays in the enclosing scope and keeps the last iterated value.

Python loop variables behave like other names in the surrounding scope. This is why `print(i)` after the loop works in Python.



Flow Control: Summary

Branching

`if / elif / else` picks one path based on conditions.

Repetition

`while` for condition-driven loops,
`for` for sequence-driven loops.

Control

`range()`, `break`, and `continue` fine-tune behavior.

Core idea:

Flow control defines how your program thinks, decides, and repeats.

Checkpoint: Loops



Q6

What does `range(10, 0)` produce?

A) 10..1

B) Empty

C) Error

D) [10, 0]

Show Answer

Q7

What is printed?

```
for i in range(3):  
    pass  
print(i)
```

A) 0

B) 2

C) 3

D) NameError

Show Answer



Strings

Immutable sequences of characters

Strings in Python

Core properties

- Sequence of case-sensitive characters
- Supports comparison operators
- Supports membership checks via `in`
- Length available via `len()`

```
s = "abc"  
print(len(s)) # 3
```



String Indexing

Use square brackets to access characters at positions.

```
s = "abc"

print(s[0])    # "a"
print(s[1])    # "b"
print(s[2])    # "c"
print(s[-1])   # "c"
print(s[-2])   # "b"
```

Accessing out-of-range indices raises an error.

String Slicing

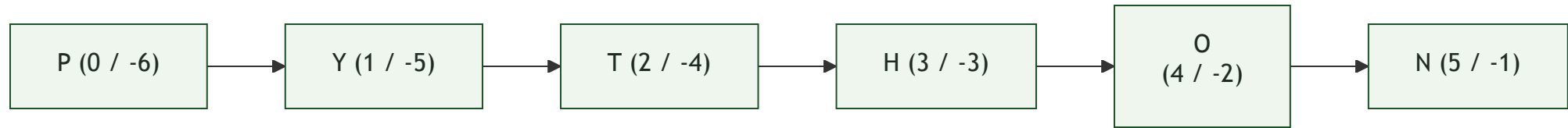


Slicing syntax: `[start:stop:step]`

```
s = "abcdefgh"

print(s[3:6])      # "def"
print(s[3:6:2])    # "df"
print(s[::])       # "abcdefgh"
print(s[::-1])     # "hgfedcba"
print(s[4:1:-2])   # "ec"
```

String Indexing Map



Read it left to right

`s[0]` is first char, `s[-1]` is last char.

Slice examples

`s[1:4]` - from index 1 up to 4 (excluded)

`s[::-1]` - full reverse

String Immutability



```
s = "hello"  
s[0] = "y"      # TypeError
```

```
s = "hello"  
s = "y" + s[1:]  # "yello"
```

Strings cannot be modified in-place. You create a new string instead.



Common String Methods

- `.lower()`, `.upper()`
- `.strip()` remove edge whitespace
- `.split()` and `"sep".join( ... )`
- `.replace(old, new)`
- `.find()`, `.count()`

```
command = "  Add  "
clean = command.strip().lower() # "add"

words = "python is fun".split()
slug = "-".join(words)
print(slug.replace("fun", "great"))
```

String methods return a new string; they do not mutate the original.

f-Strings (Recommended Formatting)

```
name = "Ada"
score = 97.5
print(f"{name} scored {score:.1f}%")

for i, task in enumerate(tasks, start=1):
    print(f"{i}. {task}")
```

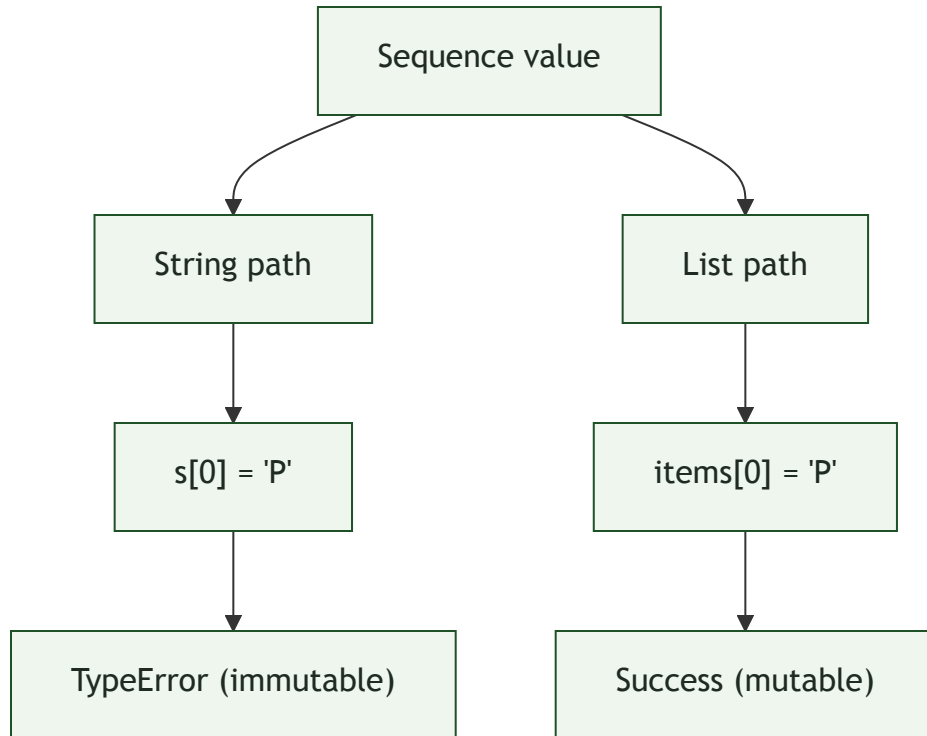
Prefix with `f`, then put expressions inside `{}`.

Why use f-strings?

- Most readable string formatting style
- Fast and explicit for mixed text + values
- Supports formatting options like `:.2f`

```
pi = 3.14159
print(f"{pi:.2f}") # 3.14
```

Mutable vs Immutable



```
s = "python"
# s[0] = "P" # TypeError

items = ["python", "java"]
items[0] = "Python" # works
```

Rule of thumb: if you need frequent edits, lists are the better container.

Checkpoint: Strings



Q8

`"hello"[::-1] = ?`

A) hello

B) olleh

C) h

D) Error

Show Answer

Q9

`"ha" * 3 = ?`

A) ha3

B) hahaha

C) Error

D) hhhaaa

Show Answer

Q10

Can we do `s[0] = "X"` on a string?

A) Yes

B) No

C) Only Tuesdays

D) With admin rights

Show Answer



Lists

Mutable collections for real-world data

Lists in Python

Why lists?

- Store collections in one variable
- Can contain mixed data types
- Mutable: editable after creation

```
my_list = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "cherry"]
mixed = [1, "apple", True]
empty = []
```



Accessing List Elements

- Indexing: `list[index]`
- Negative indexing: `list[-1]` for last item
- Slicing: `list[start:stop:step]`
- Membership: `value in list`

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])      # "apple"
print(fruits[-1])     # "cherry"
print(fruits[1:])     # ["banana", "cherry"]
print(fruits[::-1])   # reversed
print("apple" in fruits) # True
```

Modifying List Elements

Add or replace

```
tasks = ["study", "exercise"]  
tasks[0] = "review"  
tasks.append("sleep")  
tasks.insert(1, "code")
```

Remove

```
tasks.remove("exercise")  
del tasks[0]  
done = tasks.pop() # removes + returns last item
```



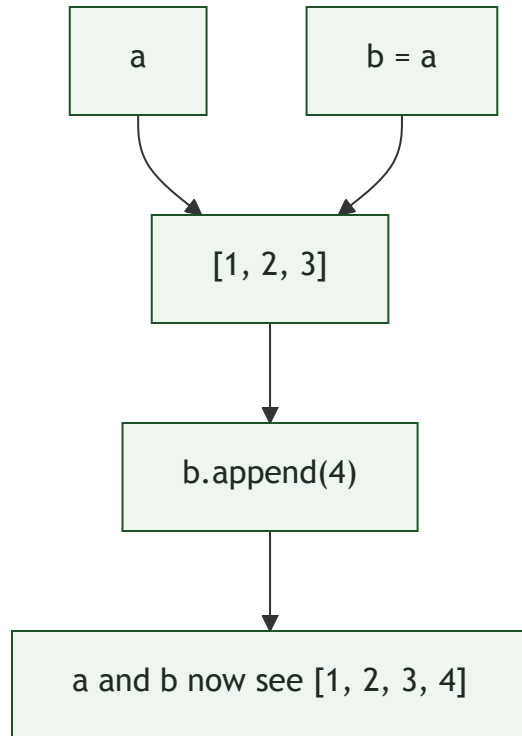

Useful List Functions

- `len(list)`: length
- `sorted(list)`: new sorted list
- `.sort()`: sort in place (returns `None`)
- `.reverse()`: reverse in place
- `.extend(items)` vs `.append(item)`

```
nums = [3, 1, 2]
print(sorted(nums)) # [1, 2, 3]
print(nums)         # [3, 1, 2]
nums.sort()         # nums changed in place

a = [1, 2]
a.append([3, 4])     # [1, 2, [3, 4]]
a.extend([5, 6])     # [1, 2, [3, 4], 5, 6]
```

List Aliasing Trap



```
a = [1, 2, 3]
b = a
b.append(4)
print(a)  # [1, 2, 3, 4]

# safe copy
c = a.copy()  # or a[:]
```

Assignment shares the same list object. Copy when you need independence.

Iterating over Lists with for

Direct iteration

Use `for item in my_list` when you only need each value.

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

Index-based iteration

Use `range(len( ... ))` when you need positions.

```
for i in range(len(fruits)):
    print(i, fruits[i])
```

enumerate(): Index + Value

Recommended pattern

```
fruits = ["apple", "banana", "cherry"]  
  
for idx, fruit in enumerate(fruits):  
    print(idx, fruit)
```

enumerate() returns pairs: (index, value).

Why use enumerate?

- Cleaner than `range(len( ... ))`
- Less error-prone index handling
- Easy to start counting from another value

```
for n, fruit in enumerate(fruits, start=1):  
    print(n, fruit)
```

List Operators

Concatenation

+ combines lists

Repetition

* repeats a list

```
fruits = ["apple", "banana"]
berries = ["blueberry", "raspberry"]

print(fruits + berries)
print(fruits * 3)
```





Why Lists Matter

Collect Data

Store many related values together.

Enable Loops

Iterate through user input and file data.

Foundation

Essential for data processing and analysis.



Lists: Summary

Data

- Mutable, ordered collections
- Can store mixed data types
- Use indexing and slicing to access values

Operations

- Add/update: `append()`, `insert()`, indexing
- Remove: `remove()`, `del`, `pop()`
- Reorder/compose: `sorted()`, `sort()`, `reverse()`, `+`, `*`

Iteration

- `for item in items` for values
- `range(len(items))` when index is needed
- `enumerate(items)` for index + value pairs
- Copy when needed: `items.copy()`

Lists are central for collecting, transforming, and iterating over program data.

Checkpoint: Lists



Q11

`[1, 2, 3] * 0` returns?

A) `[0, 0, 0]`

B) `[]`

C) Error

D) `[1, 2, 3]`

Show Answer

Q12

What is printed?

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)
```

A) `[1, 2, 3]`

B) `[1, 2, 3, 4]`

C) Error

D) `[4]`

Show Answer



To-do App Demo

Putting flow control, strings, and lists together



Iteration 4

Feature 4.1

Manage **3 hardcoded tasks**. Store and print their descriptions using lists and loops.

Feature 4.2

Ask the user for one extra task, append it to the list, then print all tasks with numbering.



Iteration 5

Implement a main command loop with this interface:

```
while True:
    command = input("Enter command (add/view/exit): ").strip().lower()

    if command == "add":
        ...
    elif command == "view":
        ...
    elif command == "exit":
        break
```

add

view

exit



Lecture Summary

Flow Control

- Use `if / elif / else` for branching
- Use `while` and `for` for repetition
- Use `range()`, `break`, `continue` to guide loops

Strings and Lists

- Strings are immutable but sliceable/indexable
- Lists are mutable and ideal for collections
- Together they power practical mini apps

Bonus: Bitwise Operators

Operator Cheat Sheet

Operator	Meaning
&	bitwise AND
	bitwise OR
^	bitwise XOR
~	bitwise NOT
<<	left shift
>>	right shift

Tip: do not confuse bitwise operators with logical `and` / `or` / `not`.

Simple Examples

```
5  # 0101
3  # 0011

5 & 3  # 0001 = 1
5 | 3  # 0111 = 7
5 ^ 3  # 0110 = 6
1 << 3  # 1000 = 8
8 >> 1  # 0100 = 4
```

Where bitwise operators are useful

- Flags and masks (permissions, status bits)
- Binary and networking tasks
- Performance-minded low-level operations



Thank You for Your Attention!

"Talk is cheap. Show me the code."

-- Linux Torvalds

if/elif/else

loops

strings

lists

Python Course 2025/26/2

Lecture 4: Tuples, Dictionaries, Functions



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Topics

- Quick recap: lists and mutability
- Tuples and immutability
- Dictionaries and key-value modeling
- Sets and membership operations
- Functions and variable scope
- Live coding: To-do app refactor

Learning objective

Use the right data structure for the right task, then wrap logic into reusable functions.

By the end, you can

Choose between `list`, `tuple`, `dict`, `set`, and structure small programs with clean function boundaries.



Tuples

Immutable sequences and why that matters

Lists: Quick Refresher

List properties

- **Mutable** items can change in place
- **Ordered** index-based access
- **Duplicates** allowed
- **Mixed types** are valid

Great when data changes over time.

```
tasks = ["email", "laundry"]
tasks.append("dishes")
tasks[1] = "groceries"
tasks.remove("email")
```

```
for task in tasks:
    print(task)
```

Lists are your default dynamic collection.



Lists vs Tuples

Feature	List	Tuple
Mutability	Mutable	Immutable
Syntax	[1, 2, 3]	(1, 2, 3)
Typical use	Data that changes	Fixed records and safe keys

Choose `list` for evolving state, choose `tuple` for stable structure.



Tuples in Python

Core properties

- Immutable after creation
- Ordered and indexable
- Duplicates allowed
- Can hold mixed data types

Useful for coordinates, configs, and return bundles.

```
coords = (10.0, 20.0)
```

```
# These fail on purpose:  
# coords.append(100.0)  
# coords[1] = 10.5  
# coords.remove(10.0)
```

Common misconception

Tuple methods for resizing do not exist. If structure must change, use a list.



Tuples in Action: Packing and Unpacking

```
my_tuple = (1, 2, 3, 4, 5)
fruits = ("apple", "banana", "cherry")
mixed = (1, "apple", True)
empty = ()

coordinates = (10, 20)    # packing
x, y = coordinates        # unpacking

a, b = 3, 9
a, b = b, a                # classic tuple swap
```

Fun Python fact: `(1)` is just an `int`, while `(1,)` is a one-element tuple.

Accessing Tuple Elements

```
dice_rolls = (1, 2, 2, 3, 4, 5, 2)

print(dice_rolls[0])      # 1
print(dice_rolls[-1])     # 2
print(dice_rolls[1:3])    # (2, 2)
print(dice_rolls[::-1])   # reversed tuple
```

Indexing patterns

- `[i]` single element
- `[-1]` last element
- `[a:b]` slicing
- `[::-1]` reverse copy



Tuple Methods and Operations

```
dice_rolls = (1, 2, 2, 3, 4, 5, 2)

print(len(dice_rolls))
print(dice_rolls.count(2))
print(dice_rolls.index(3))

fake_dice_rolls = (6, 6, 6)
all_dice_rolls = dice_rolls + fake_dice_rolls
print(all_dice_rolls)
```

Membership checks

```
6 in dice_rolls
```

```
6 not in dice_rolls
```

Tuples support the same membership operators as lists and strings.



Why Immutability Matters

Data integrity

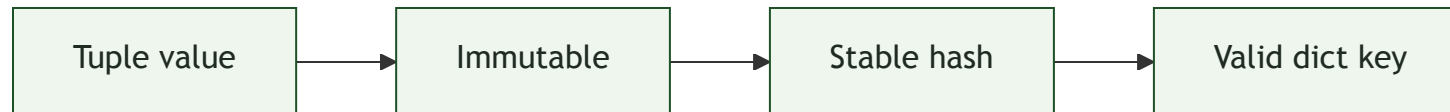
Critical records cannot be changed by accident.

Performance

Fixed-size memory layout is efficient to store and iterate.

Hashability

Stable values can become dictionary keys.



Immutability connects tuples directly to dictionary design because keys must remain stable.

Checkpoint: Tuples



Q1

What is the type of `(1)`?

- A) tuple
- B) int
- C) list
- D) Depends on moon phase

Show Answer

Q2

What happens with `coords[1] = 99` if `coords` is a tuple?

- A) Works and updates index 1
- B) TypeError
- C) KeyError
- D) Only works on Tuesdays

Show Answer

Q3

Which object can be a dictionary key?

- A) `[1, 2]`
- B) `(1, 2)`
- C) `{"x": 1}`
- D) `([1, 2], 1)`

Show Answer



Dictionaries

Flexible key-value data modeling

Dictionarys in Python

Dictionary basics

- Stores `key: value` pairs
- Keys must be unique and immutable
- Values can be any type and can repeat
- Fast lookup by key

```
student_grades = {  
    "Alice": 95,  
    "Bob": 87,  
    "Charlie": 92  
}  
  
student = {"name": "Alice", "age": 20}  
empty = {}
```

Real-world hook: JSON payloads map directly to dictionaries.

Key-Value Mental Model



Keys

"name"

"age"



Values

"Alice"

20

```
student = {  
  "name": "Alice",  
  "age": 20  
}
```



Keys Must Be Immutable and Unique

```
valid_dict = {  
    "name": "Alice",  
    100: "ID Number",  
    (10, 20): "coordinate",  
    "name": "Alicia"    # overwrites previous "name"  
}  
print(valid_dict)  
  
# invalid_dict = {  
#     ["a", "b"]: "list key"    # TypeError  
# }
```

Gotcha: duplicate keys are not stored twice, the last value wins.

Accessing Dictionary Values

```
student = {"name": "Alice", "age": 20}
```

```
student["name"]      # "Alice"  
student["grade"]     # KeyError
```

```
student.get("age")   # 20  
student.get("grade") # None  
student.get("grade", "N/A")
```

Method	Missing key behavior
<code>dict[key]</code>	Raises <code>KeyError</code>
<code>dict.get(key)</code>	Returns <code>None</code> or provided default



Modifying Dictionaries

```
student = {"name": "Alice", "age": 20}

# add
student["grade"] = "A"

# update
student["age"] = 21

# remove
del student["age"]          # or student.pop("age")

# batch update
student.update({"city": "Budapest"})
```

Use `pop()` when you want both deletion and the removed value.



Iterating Through Dictionaries

```
person = {"name": "Bob", "age": 30, "city": "New York"}

for key in person:
    print(f"Key: {key}, Value: {person[key]}")

for value in person.values():
    print(f"Value: {value}")

for key, value in person.items():
    print(f"{key} → {value}")
```

Choose `items()` when you need both key and value.

Dictionary Methods

```
student = {  
    "name": "Alice",  
    "age": 20,  
    "grade": "A"  
}
```

```
student.keys()  
student.values()  
student.items()  
student.clear()
```

Notes

- These methods return view objects
- Convert with `list( ... )` when needed
- Views update if dictionary changes

Fun Python fact: dict preserves insertion order (Python 3.7+).

Membership Operator: `in` and `not in`

Works with many iterables

`list` `tuple` `string` `dict` `set`

For dictionaries, membership checks keys by default.

```
text = "Gandalf the Grey"
print("h" in text)

db_config = {"user": "admin", "password": "012345678"}
print("password" in db_config)           # key lookup
print("admin" in db_config.values())     # value lookup
```

Checkpoint: Dictionaries



Q4

Which call avoids `KeyError` for missing keys?

- A) `d["x"]`
- B) `d.get("x", 0)`
- C) `d.keys("x")`
- D) `d.unlock("x")`

Show Answer

Q5

What is printed?

```
d = {"name": "Alice", "name": "Alicia"}  
print(d["name"])
```

- A) Alice
- B) Alicia
- C) ["Alice", "Alicia"]
- D) Identity crisis

Show Answer

Q6

Which key is valid in a dictionary?

- A) `["a", "b"]`
- B) `{"x": 1}`
- C) `(10, 20)`
- D) `([10], 20)`

Show Answer



Data Types and Sets

Choosing the right container



Non-Scalar Data Types

☰ List

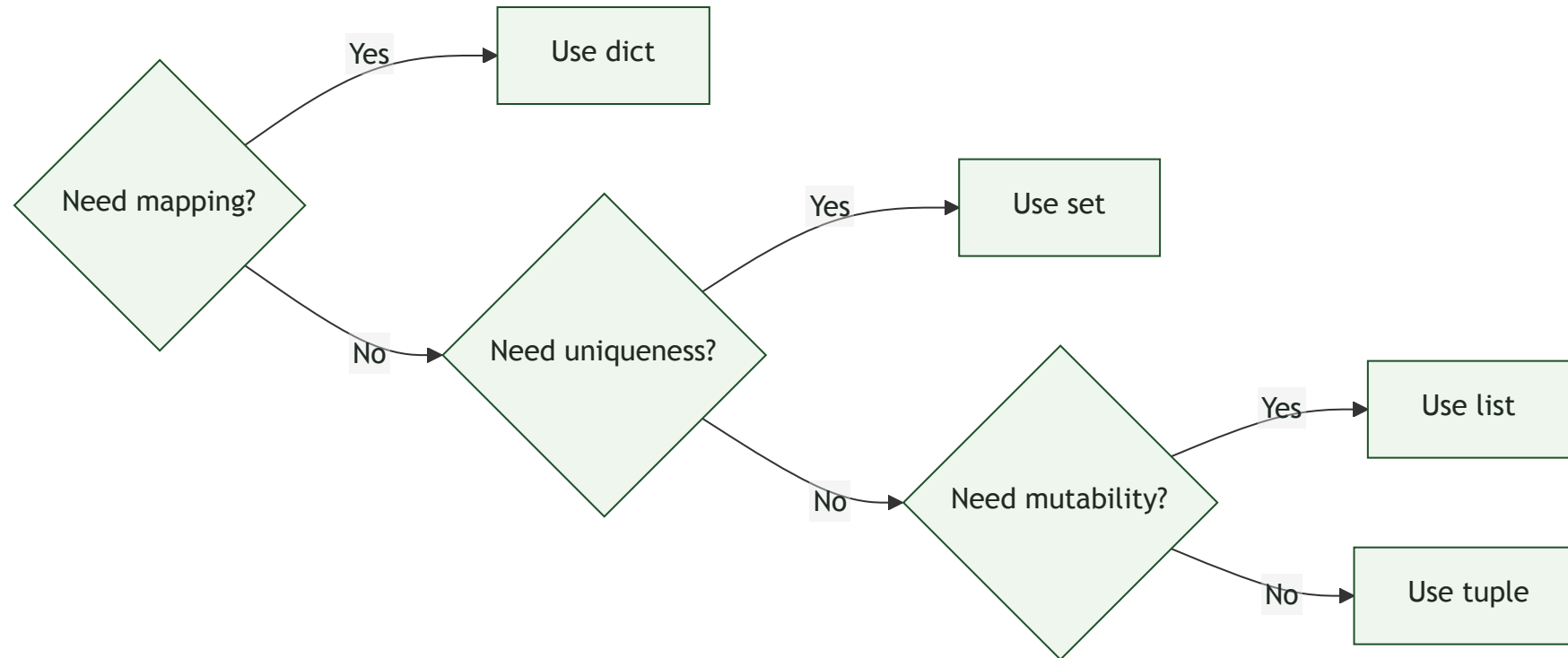
Ordered, mutable sequence

🔗 Tuple

Ordered, immutable sequence

☐ Dict

Key-value mapping



Data structure choice should be intentional, not accidental.

Sets in Python

Set properties

- Unordered and unique elements
- Mutable collection
- No indexing or slicing
- Excellent for fast membership checks

Gotcha: empty set is `set()`, not `{}`.

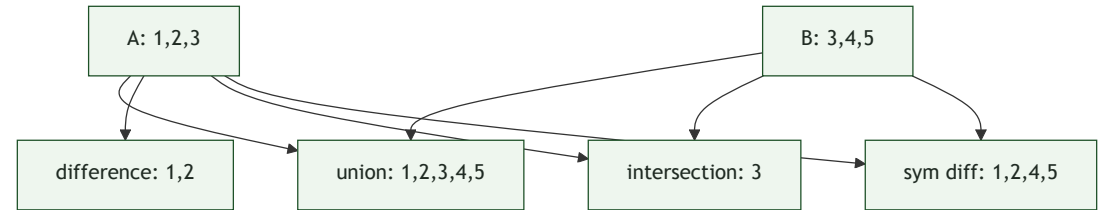
```
my_set = {1, 2, 3, 4}
print(my_set)
```

```
empty_set = set()
my_set.add(5)
my_set.remove(3)
```

Set Operations

```
a = {1, 2, 3}
b = {3, 4, 5}

print(a.union(b))           # {1,2,3,4,5}
print(a.intersection(b))    # {3}
print(a.difference(b))      # {1,2}
print(a.symmetric_difference(b)) # {1,2,4,5}
```



Checkpoint: Sets



Q7

What does this print?

```
a = {1, 2, 3}
b = {3, 4}
print(a.intersection(b))
```

A) {1, 2, 3, 4}

B) {3}

C) [3]

D) The set is shy and prints nothing

Show Answer

Q8

How do you create an empty set?

A) {}

B) []

C) set()

D) summon_set()

Show Answer



Functions and Scope

Reusable logic and variable visibility

Functions in Python

Why functions?

- Follow **DRY** principle
- Improve modularity and readability
- Enable reuse and testing

```
def function_name(parameter1, parameter2):  
    # do something  
    return result
```

Anatomy: `def` + name + parameters + indented body + optional return.



Calling a Function

```
# 1) define
def greet():
    print("Hello, world!")
    print("Welcome to the world of functions.")

# 2) call
greet()
greet()
```

Define once, call many times.



Parameters vs Arguments

Definitions

- **Parameter:** variable in function definition
- **Argument:** concrete value passed on call

```
def greet_person(name):    # parameter
    print(f"Hello, {name}!")
```

```
greet_person("Gandalf")    # argument
greet_person("Luke")
```



Return Values

```
def add(a, b):  
    result = a + b  
    return result  
  
sum_result = add(5, 3)  
print(sum_result) # 8
```

No return

Implicitly returns `None`

Multiple values

Returned as a tuple

Early return

Exits function immediately



Positional and Keyword Arguments

```
def describe_pet(pet_name, animal_type):  
    print(f"I have a {animal_type} named {pet_name}.")  
  
describe_pet("Willie", "hamster")           # positional  
describe_pet(animal_type="cat", pet_name="Milo") # keyword
```

Keyword arguments improve readability when function signatures grow.



Default Arguments

```
def describe_pet(pet_name, animal_type="dog"):
    print(f"I have a {animal_type} named {pet_name}.")
```

```
describe_pet("Fido")
```

Gotcha: avoid mutable defaults.

```
# bad
```

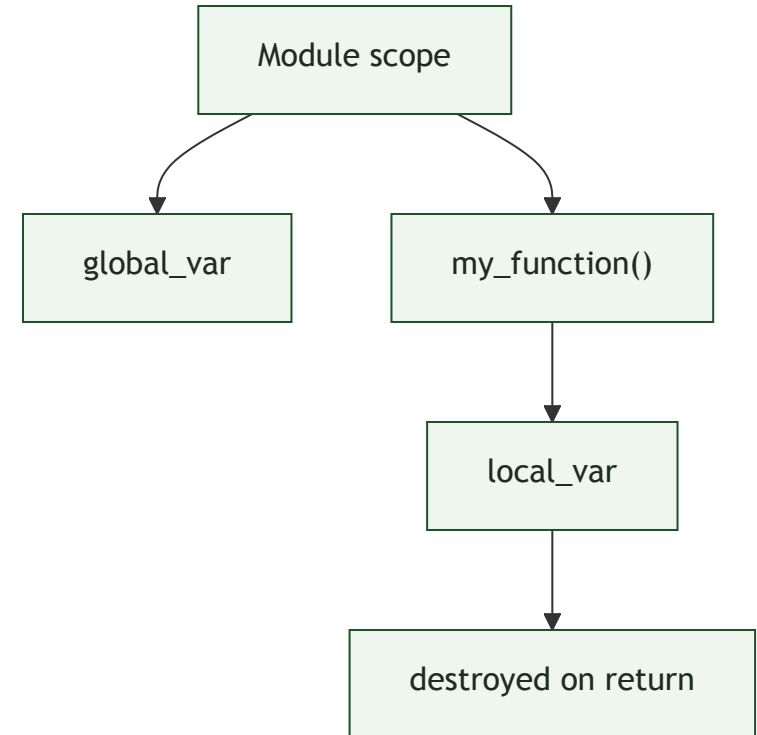
```
def add_item(item, items=[]):
    items.append(item)
    return items
```

```
# better
```

```
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
```

Variable Scope: Local vs Global

- **Local scope:** inside a function
- **Global scope:** module-level variables
- Functions can read globals
- Locals are created/destroyed on call



Scope Example

```
global_var = "I am global"

def my_function():
    local_var = "I am local"
    print(global_var)
    print(local_var)

my_function()
print(global_var)
# print(local_var) # NameError
```

Local names are invisible outside the function.



No Block Scope in Python

```
for i in range(3):  
    loop_var = 100  
    print(f"Inside loop: i={i}, loop_var={loop_var}")  
  
print(i)          # 2  
print(loop_var)   # 100
```

Unlike Java/C++, loop variables remain available after the loop ends.



Checkpoint: Functions and Scope

Q9

What is printed?

```
def f(x=5):  
    return x * 2  
print(f())
```

- A) 5
- B) 10
- C) None
- D) Server overheats

Show Answer

Q10

Why can mutable default arguments be risky?

- A) They are recreated on every call
- B) They are shared across calls
- C) Python forbids them
- D) They require admin rights

Show Answer



Live Coding: To-do App

Refactoring with dictionaries and functions

Iteration Goal

From Iteration 5

Simple loop with list-based tasks and basic commands.

Now in Iteration 6/7

Add richer task structure (`id` , `status`) and split logic into clean functions.





Feature 6.2: Task Metadata

```
task = {  
    "id": 101,  
    "title": "finish lab",  
    "status": "pending"  
}  
  
ALLOWED_STATUS = {"pending", "completed"}
```

Store tasks as dictionaries so fields stay explicit and extensible.



Refactor Principles

Separate I/O

Logic functions return data, main loop handles input/output.

Single responsibility

One function, one job: add, find, complete, render.

Validate input

Never trust user input; check IDs and statuses.

Return data structures, then render messages in one small formatting layer.

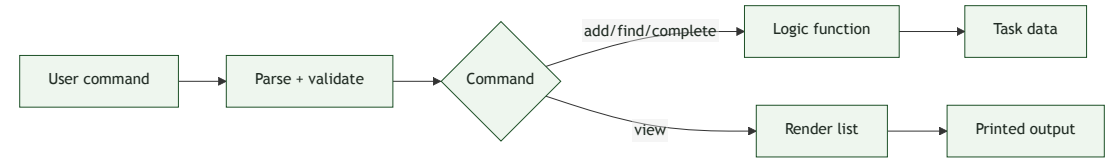
Iteration 7: New Commands

Feature 7.1

`find <id>` searches by task ID.

Feature 7.2

`complete <id>` updates task status.





Summary

Tuples

- Immutable sequence
- Good for fixed records
- Useful as dictionary keys

Dictionaries and Sets

- Dict: key-value lookup
- Set: uniqueness + operations
- Membership checks are central

Functions and Scope

- Reusable logic blocks
- Parameter/argument discipline
- Understand local vs global names

Next step: combine data structures + functions to build maintainable mini applications.



Thank You for Your Attention!

"Simple is better than complex."

-- The Zen of Python

tuples

dicts

sets

functions

Python Course 2025/26/2

Lecture 5: Error Handling, Files, Comprehensions



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Agenda

- Recap: functions and to-do app assumptions
- Error types and resilient mindset
- `try/except/else/finally` and `raise`
- File I/O: modes, context managers, common failures
- JSON persistence for structured data
- Comprehensions and generator expressions

Learning objective

Write Python programs that fail gracefully, keep data persistent, and stay concise without losing readability.

By the end, you can

Handle user and file errors, choose correct file modes, save state with `json`, and use comprehensions appropriately.



Recap

Functions and the to-do app baseline



Functions: Quick Refresher

Core terms

- Function: reusable block of logic
- Parameters: placeholders in definition
- Arguments: values at call time
- Return value: result sent back to caller
- Local scope: names created inside function

```
def calculate_area(length, width):  
    """Calculate rectangle area."""  
    area = length * width  
    return area  
  
result = calculate_area(5, 3)  
print(result) # 15
```

Functions improve reuse, testing, and readability.



Recap: To-Do App State

```
tasks = {
    1: {"description": "Learn Python basics", "status": "complete"},
    2: {"description": "Implement to-do app", "status": "pending"},
}

def add_task(description, status="pending"):
    task_id = len(tasks) + 1
    tasks[task_id] = {"description": description, "status": status}
    return task_id, tasks[task_id]
```

Assumptions we make

- User always enters valid IDs
- Keys always exist in dictionaries
- Program never restarts unexpectedly

Reality check: in-memory data disappears when the process ends. We need resilience and persistence.



Error Handling

From crashes to graceful behavior



What Could Go Wrong?

Invalid key access

```
tasks[99] # KeyError
```

Happens when users request non-existing task IDs.

Invalid numeric input

```
task_id = int(input("Enter Task ID: "))  
# "three" → ValueError
```

Common in CLI apps where input arrives as strings.

Engineering mindset: robust software starts with proactive pessimism: "What happens when assumptions fail?"

Resiliency



Errors

Problems that prevent intended behavior.

Exceptions

Runtime events that disrupt normal flow.

Resilient code

Continues safely or fails with clear user feedback.

Goal of this section: anticipate, catch, and handle exceptions without crashing the whole program.



Errors vs Exceptions vs Logic Bugs

```
# Syntax error (detected before execution)
# print("Hello"

# Runtime exception
print(10 / 0) # ZeroDivisionError

# Logic bug (runs, wrong result)
def add_numbers(a, b):
    return a - b
```

Quick distinction

- **SyntaxError** parser rejects invalid Python
- **Exception** code is valid but runtime context fails
- **Logic error** no crash, but wrong output

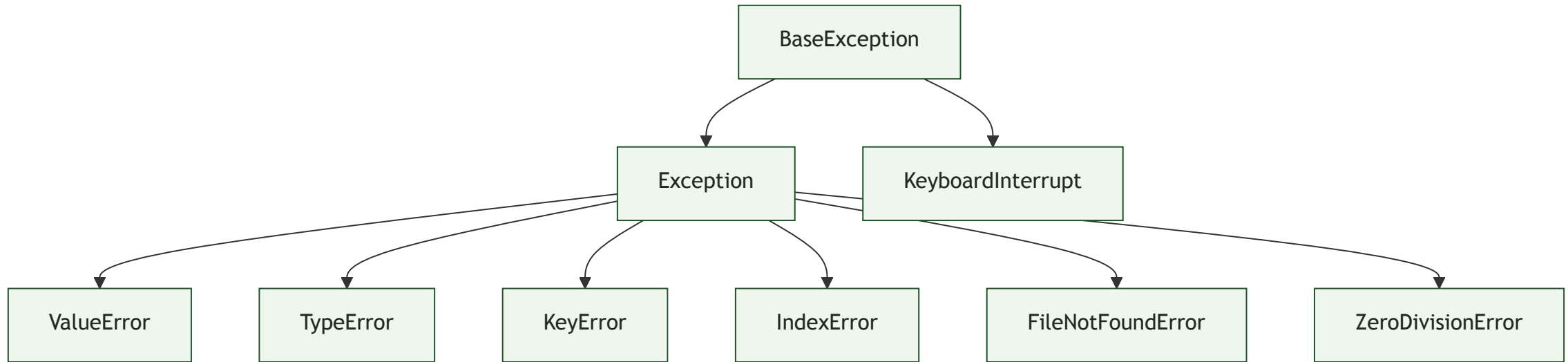


Exceptions We Already Met

Exception	Typical trigger	Example
<code>NameError</code>	Undefined variable in scope	<code>print(total_price)</code>
<code>TypeError</code>	Operation on incompatible types	<code>"3" + 5</code>
<code>ValueError</code>	Right type, wrong value	<code>int("abc")</code>
<code>IndexError</code> / <code>KeyError</code>	Missing position or key	<code>my_list[99]</code> , <code>my_dict["x"]</code>

Different exception types communicate different failure causes. Catching specifically makes fixes clearer.

Exception Hierarchy (Big Picture)



Catch the most specific exception you expect. Broad catches should be rare and intentional.



Handling Exceptions with try/except

```
try:
    x = int(input("Enter an integer: "))
    result = 10 / x
    print("10 /", x, "=", result)
except ValueError:
    print("Please enter a valid number!")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Execution model

- Run risky code in `try`
- Jump to first matching `except`
- Skip other code in `try` after an exception
- If no exception: `except` blocks are skipped



else and finally

else

Runs only when the `try` block succeeds without any exception.

Good for post-processing successful results.

finally

Runs no matter what: success, handled failure, or re-raised exception.

Use for cleanup (files, sockets, temporary resources).

```
try:
    risky_operation()
except SomeError:
    handle_error()
else:
    process_success()
finally:
    cleanup_resources()
```

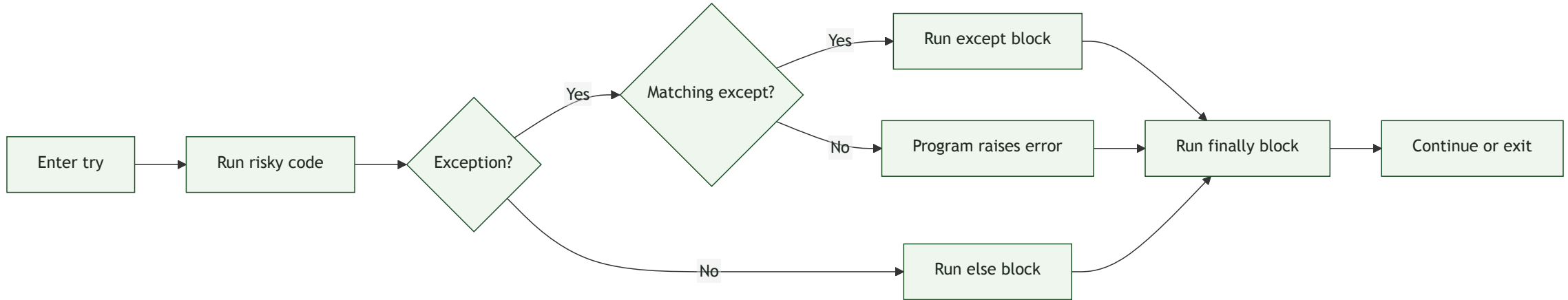
Example: safe_division()



```
def safe_division(numerator, denominator):  
    try:  
        num = float(numerator)  
        den = float(denominator)  
        result = num / den  
    except ValueError:  
        print("Invalid input: use numbers.")  
    except ZeroDivisionError:  
        print("Cannot divide by zero.")  
    else:  
        print(f"Success! Result: {result}")  
    finally:  
        print("Division attempt completed.")
```

Pattern: validate and execute in `try`, communicate failures in `except`, keep success-only output in `else`.

try/except/else/finally Flow



`finally` always runs, even when an exception is not handled locally.



raise: Signal Invalid States

```
def calculate_birth_year(age, current_year):  
    if age < 0:  
        raise ValueError("Age cannot be negative.")  
    return current_year - age  
  
try:  
    year = calculate_birth_year(-5, 2024)  
except ValueError as err:  
    print("Calculation failed:", err)
```

Why raise manually?

- Enforce function preconditions
- Fail early with clear messages
- Let calling code decide recovery strategy



Best Practices

Be specific

Catch known exceptions instead of broad `Exception`.

Handle meaningfully

Provide user-friendly messages and useful fallback behavior.

Do not overuse

Not every line needs a `try` block.

Common gotcha: bare `except:` catches almost everything, including `KeyboardInterrupt`.



Checkpoint: Error Handling

Q1

What is printed by this code: `try: int("abc") ... except ValueError: print("bad input")`?

- A) bad input
- B) ok
- C) no output
- D) The laptop enters meditation mode

Show Answer

Q2

What is risky about using bare `except:` ?

- A) It only catches ValueError
- B) It catches many exceptions, including interrupts you often want to propagate
- C) It works only on Tuesdays
- D) Python automatically ignores it

Show Answer



File I/O

Persistence and safe file access

Introduction to File I/O



Why files matter

- Persist data across program restarts
- Share data between tools
- Record logs and configuration

Common categories

- Text files: `.txt`, `.csv`, `.json`
- Binary files: `.jpg`, `.pdf`, `.exe`
- Core operations: open, read/write, close



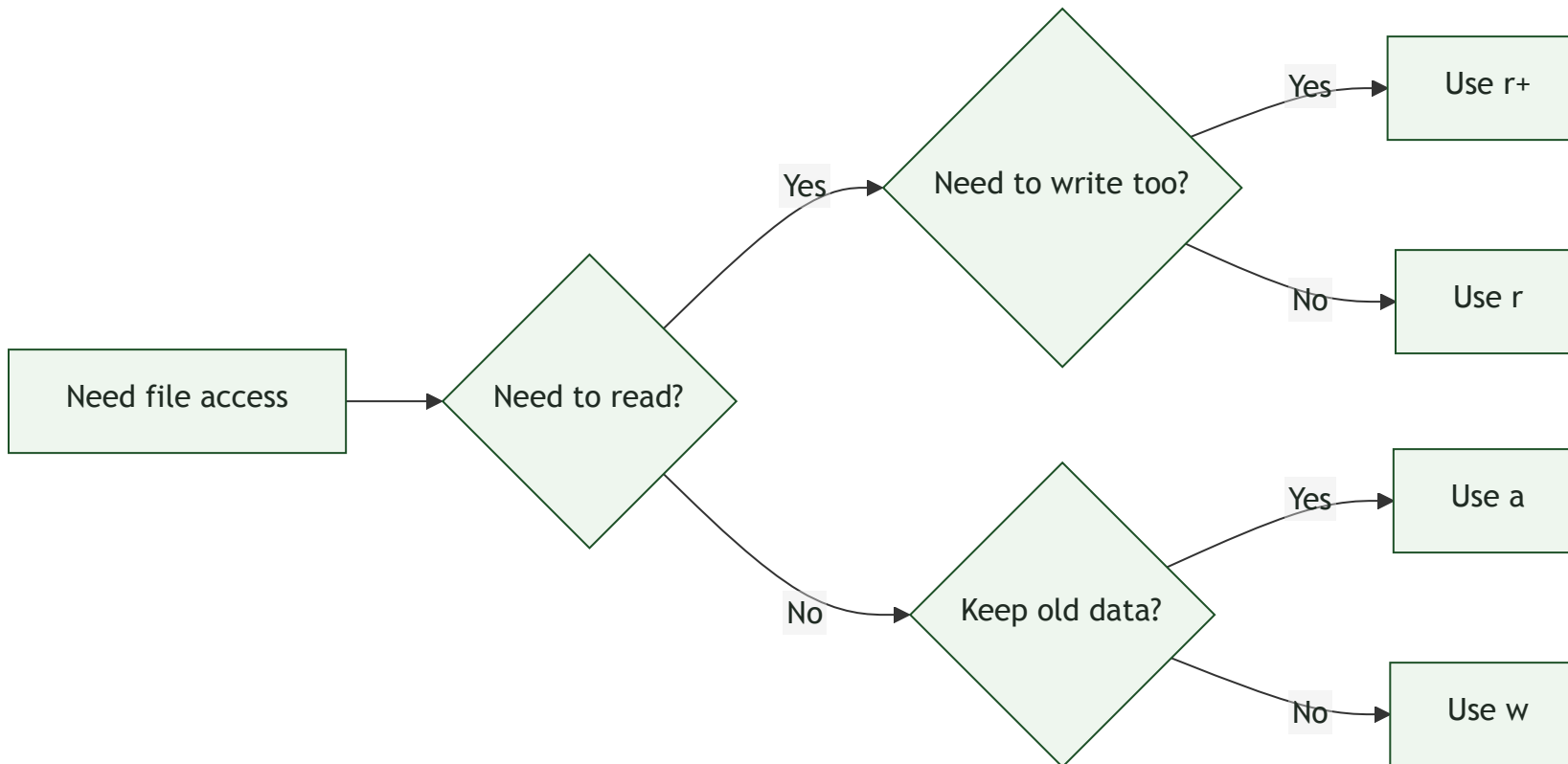
File Modes at a Glance

Mode	Purpose	If file exists	If file missing
r	Read only	Read from start	Raises FileNotFoundError
w	Write (truncate)	Overwrites all content	Creates new file
a	Append	Adds to end	Creates new file
r+	Read + write	Starts at beginning	Raises FileNotFoundError

Gotcha: `w` truncates immediately.



Which Mode Should You Use?



Start from intent: read, append, or replace. The mode should document your intent clearly.

Opening Files Safely with with

```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)

with open("story.txt", "a") as file:
    file.write("New line\n")
```

Why context managers?

- Close files automatically
- Safer when exceptions occur
- Less boilerplate than manual `close()`

Fun fact: `with` uses `__enter__` and `__exit__` under the hood.

Reading and Writing Patterns



Reading

```
with open("story.txt", "r") as file:
    for line in file:
        print(line.strip())
```

Line-by-line loops scale better for large files.

Writing

```
lines = ["Line 1\n", "Line 2\n"]
with open("story.txt", "a") as file:
    file.writelines(lines)
```

Use explicit newlines when writing line-wise output.

Handling File Errors (EAFP)

```
try:
    with open("config.txt", "r") as cfg:
        settings = cfg.read()
except FileNotFoundError:
    print("Config not found, using defaults.")
```

Typical file exceptions

- `FileNotFoundError`
- `PermissionError`
- `IsADirectoryError`

EAFP: try the operation, then handle failure if it happens.

EAFP keeps code direct and avoids race conditions common in pre-check patterns.



Structured Persistence with JSON

Why JSON?

- Human-readable text format
- Maps directly to Python lists/dicts
- Ideal for app state and settings

Core API

`json.dump(data, file)`: Python object -> JSON text

`json.load(file)`: JSON text -> Python object



JSON Example

```
import json

student_data = {
    "name": "Alice",
    "id": 12345,
    "courses": ["History", "Math", "Physics"]
}

with open("data.json", "w") as f:
    json.dump(student_data, f, indent=4)

with open("data.json", "r") as f:
    loaded_data = json.load(f)
```

JSON avoids brittle manual parsing for nested structures.



Checkpoint: File I/O

Q3

What happens when you open an existing file with mode `"w"` ?

- A) Existing content is preserved
- B) Existing content is deleted, then writing starts
- C) Python asks for admin approval first
- D) It switches automatically to append mode

Show Answer

Q4

Why is `with open( ... )` preferred over manual `open()` / `close()` ?

- A) It can auto-close files even when exceptions occur
- B) It doubles read speed by default
- C) It works only for JSON files
- D) It unlocks quantum storage mode

Show Answer



Comprehensions

Compact, expressive transformations

List Comprehensions

```
# Classic loop
squares = []
for x in range(10):
    squares.append(x ** 2)

# List comprehension
squares = [x ** 2 for x in range(10)]
even_squares = [x ** 2 for x in range(10) if x % 2 == 0]
```

When to use

- Simple transform/filter in one expression
- Readability still remains high
- Avoid deep nesting inside one line



Dictionary Comprehensions

```
# Number → square mapping
square_dict = {x: x ** 2 for x in range(5)}

fruits = {"apple": 1, "banana": 2, "cherry": 3}
inverted_fruits = {value: key for key, value in fruits.items()}

inventory = {"apples": 30, "bananas": 50, "oranges": 10}
low_stock = {item: count for item, count in inventory.items() if count < 20}
```

Great for remapping and filtering dictionary data in one pass.



Generator Expressions and Memory

```
large_list = [x ** 2 for x in range(1_000_000)]  
large_generator = (x ** 2 for x in range(1_000_000))  
  
total = sum(large_generator)
```

Generators produce values lazily, one at a time.

Comprehension vs generator

List comprehension

Stores full result in memory.

Generator expression

Streams values on demand.

Use generators for large pipelines where results are consumed once.



Gotcha: List Multiplication Aliasing

✂ The trap

```
grid = [[0, 0, 0]] * 3
print(grid) # [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
grid[0][0] = 9
print(grid)
# [[9, 0, 0], [9, 0, 0], [9, 0, 0]] # all rows changed
```

✓ The fix

```
grid = [[0, 0, 0] for _ in range(3)]
grid[0][0] = 9
print(grid)
# [[9, 0, 0], [0, 0, 0], [0, 0, 0]] # only row 0 changed
```

Why it happens

`[[0, 0, 0]] * 3` copies the **reference** three times, so every slot points to the same inner list. A comprehension creates a **new list object** on each iteration.

Rule of thumb: avoid using `*` to duplicate mutable nested objects. Use a comprehension instead.



Checkpoint: Comprehensions

Q5

What does this expression produce: `[x * 2 for x in range(5) if x % 2 == 0]`?

- A) `[0, 2, 4, 6, 8]`
- B) `[0, 4, 8]`
- C) `[2, 6]`
- D) It prints only on leap years

Show Answer

Q6

Which statement about generator expressions is true?

- A) They always create a list first
- B) They generate values lazily on demand
- C) They cannot be used with `sum( )`
- D) They require a GPU

Show Answer



Live Coding: To-Do App

Iteration 8 and 9



Iteration Roadmap

Iteration 8

Handle non-numeric task IDs and avoid crashes.

Iteration 9.1

Persist tasks to `tasks.txt`.

Iteration 9.2

Persist structured tasks to `tasks.json`.

Goal: upgrade from a fragile in-memory script to a durable CLI mini-application.



Iteration 8: Robust Input Handling

```
def ask_task_id():  
    while True:  
        raw = input("Enter task ID: ")  
        try:  
            return int(raw)  
        except ValueError:  
            print("Please enter a numeric ID.")
```

User-facing errors should be clear and actionable, not raw tracebacks.



Iteration 9.1: Persist to tasks.txt

```
def load_tasks_txt(path="tasks.txt"):
    try:
        with open(path, "r") as f:
            return [line.strip() for line in f if line.strip()]
    except FileNotFoundError:
        return []

def save_tasks_txt(tasks, path="tasks.txt"):
    with open(path, "w") as f:
        for task in tasks:
            f.write(task + "\n")
```

Simple but limited for multi-field task structures.



Iteration 9.2: Persist to tasks.json

```
import json

def load_tasks_json(path="tasks.json"):
    try:
        with open(path, "r") as f:
            return json.load(f)
    except FileNotFoundError:
        return {}

def save_tasks_json(tasks, path="tasks.json"):
    with open(path, "w") as f:
        json.dump(tasks, f, indent=2)
```

Text vs JSON

- **TXT** quick for plain lines
- **JSON** better for nested metadata
- JSON reduces manual parsing errors

For a real to-do app with IDs, status, and descriptions, JSON is the safer long-term choice.



Summary

Error Handling

- Classify failure types
- Use specific exceptions
- Prefer graceful recovery

File I/O and JSON

- Choose modes intentionally
- Use `with open( ... )`
- Persist structured data with JSON

Comprehensions

- Express simple transforms compactly
- Use generators for large streams
- Protect readability over cleverness

Resilient programs are not error-free; they are error-aware and user-respectful.



Thank You for Your Attention!

"Errors should never pass silently. Unless explicitly silenced."

-- The Zen of Python

exceptions

files

json

comprehensions

Python Course 2025/26/2

Lecture 6: OOP, Packages, and Project Structure



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Agenda

- OOP foundations: classes, objects, state, inheritance
- Modules, packages, and project structure
- Dependency management with `pip`, `requirements.txt`, virtual environments, `uv`, and `pixi`
- Type hints, PEP 8, and docstrings
- Command-line apps and the to-do package architecture

Learning objective

Design small object models, split them into packages, isolate dependencies safely, and wire them into a maintainable command-line application.

By the end, you can

Model simple domains with classes, organize code into modules and packages, manage dependencies with modern Python tools, and structure a small CLI project for clean growth.



OOP Foundations

From functions and data to cooperating objects

Python Is Multi-Paradigm

Python supports many styles

- Imperative: step-by-step instructions
- Functional: functions as values, mapping, filtering
- Declarative: comprehensions and generators
- Event-driven: common in GUIs and web frameworks
- Object-oriented: data and behavior live together

Why focus on OOP today?

Small scripts can stay procedural, but bigger programs need a cleaner way to model state, behavior, and collaboration.

Today's mental shift

Instead of asking only "Which function runs next?", we also ask "Which object owns this data, and who should change it?"

Why OOP When Programs Grow?

Procedural code is great when

- The task is small and linear
- State is simple and short-lived
- Only a few functions need to cooperate

But bigger projects hurt

Games, web apps, and data tools accumulate state. When data and behavior are scattered across many functions, code becomes harder to extend and debug.

Procedural Programming (Recipe)



- Focuses on steps/sequences
- Data and functions separate
- Example: "Step 1: Mix flour & sugar. Step 2: Bake at 350°F."

Object-Oriented Programming (Kitchen Station)



- Focuses on objects/classes
- Data and methods encapsulated
- Example: "BakeryStation" class with mix(), bake(), plate() methods.



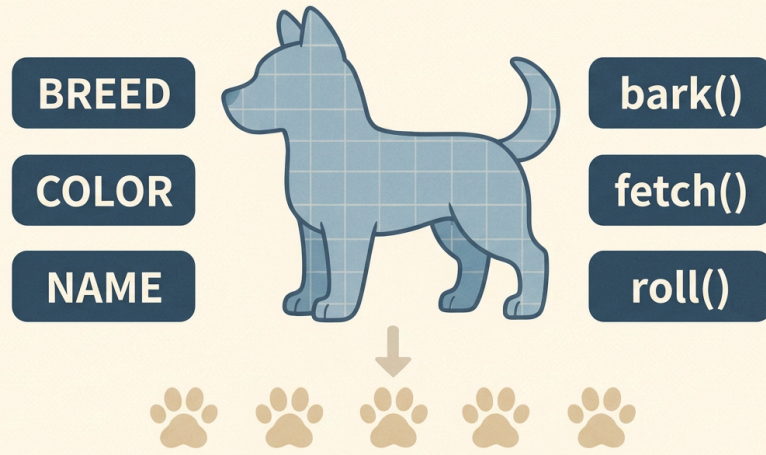
Key Differences

- Procedural: Linear flow; OOP: Modular reuse

Class vs Object

Class = blueprint

Dog Class (Template)



A class describes what data and behavior instances should have.

Object = one concrete instance

Dog Object (Instance)



An object stores actual values like a specific name, age, or breed.

Common confusion: a class is not "one special object". It is the reusable definition used to create many objects.

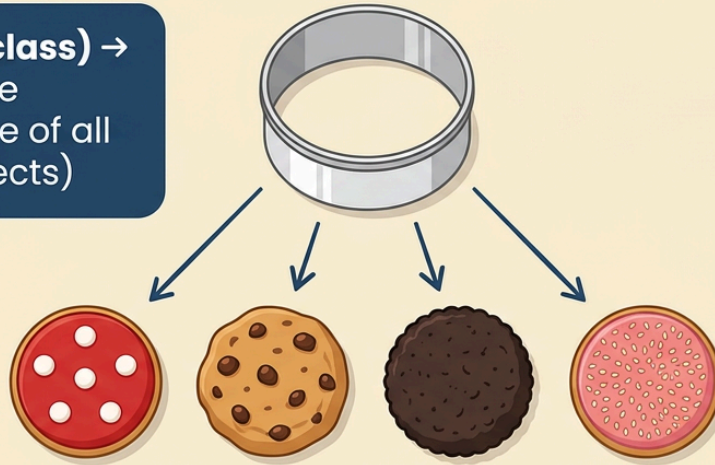
Fun Python fact: classes are objects too, so you can store a class in a variable, pass it to a function, or inspect it with `type()`.

Analogy: Cookie Cutter and Cookies



What Are Classes & Objects In Python?

Cookie cutter (class) →
Defines the
blueprint/shape of all
cookies (objects)



Different cookies (objects) → Have same blueprint but vary in flavor, toppings,

The cutter defines the shared shape; each cookie is a separate object with its own toppings, color, or flavor.

Anatomy of a Python Class

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("Woof!")

    def birthday(self):
        self.age += 1
```

Methods are functions inside the class. Their first parameter refers to the current object.

Four core pieces

- `class Dog:` creates a new type
- `__init__` runs when an object is created
- `self.name`, `self.age` are attributes
- `bark()`, `birthday()` are behaviors

Misconception to avoid: `self` is not a keyword like `if`. It is the reference to the current instance by convention.



Objects Remember Their Own State

```
mochi = Dog("Mochi", 2)
print(mochi.status())

mochi.walk(2)
mochi.feed(150)
mochi.learn("roll over")

print(mochi.status())
```

Calls on the same object update the same stored attributes.

Encapsulation

- Data lives inside the object
- Methods are the approved ways to change that data
- The object's state evolves over time

OOP feels natural when state and behavior belong together: the dog walks, the dog gets hungry, the dog learns tricks.



Objects Can Collaborate

```
adopted = shelter.adopt(alice, dogs[0])  
print(adopted.name, "→", alice.name)  
print([d.name for d in shelter.dogs])
```

Dog

Stores data about one dog.

Shelter

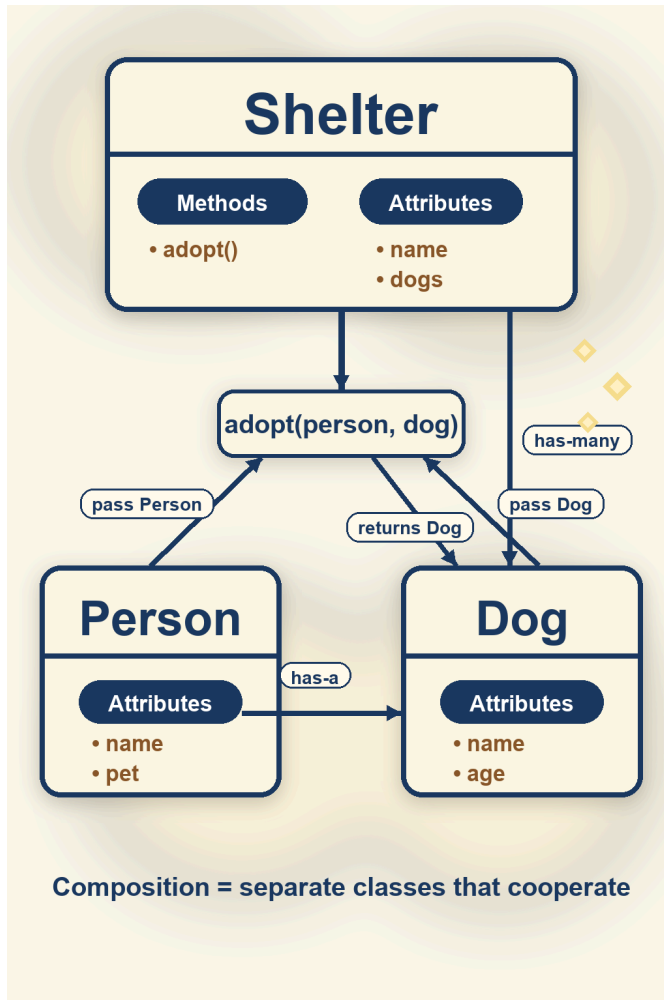
Owns a collection of dogs and manages adoption.

Person

May contain a reference to a pet.

Objects can be passed into methods, stored inside other objects, and returned from method calls. That is how small models become systems.

Composition: A Relationship Map



Four collaboration patterns

- **has-a** : one object keeps a reference to another
- **has-many** : one object manages a collection of others
- **Method parameter**: a method receives other objects so it can coordinate them
- **Method return**: a call can hand back the object it just worked with

Composition keeps classes independent. A shelter is not a dog and a person is not a dog, but they can still cooperate by storing references and calling methods.

Inheritance: Reuse With Care

When it helps

- A child class is a more specific version of the parent
- The child should reuse behavior and data layout
- The relationship is truly `is-a`

Good example: `Dog` is an `Animal`. Bad example: a shelter is not an animal.

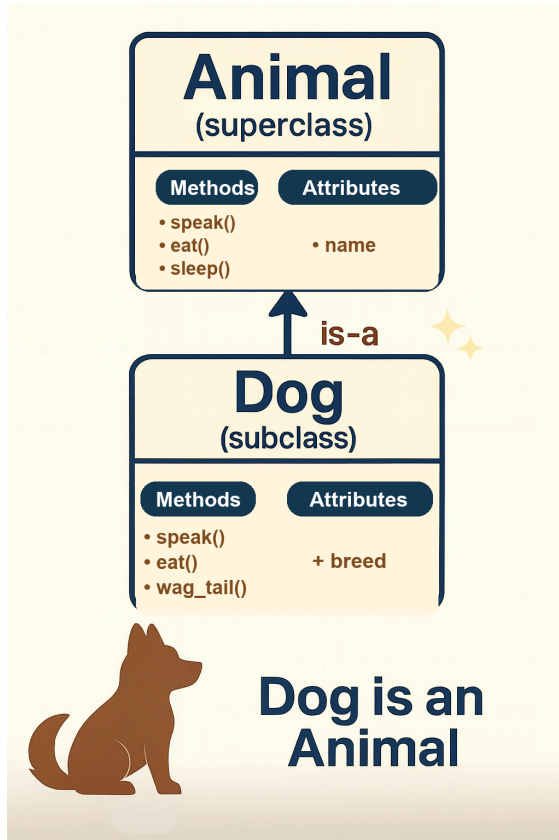
```
class Animal:
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def speak(self):
        print("Woof!")
```

`super()` lets the child reuse the parent's setup logic.

What Changes in the Child Class?



Four common cases

- Inherited untouched: the child keeps a parent behavior as-is
- Overridden: the child replaces a parent behavior with a more specific version
- Extended: the child starts with the parent behavior, then adds extra work
- New child-only method: the child adds behavior the parent never had

Inheritance is about specialization. If you only want one object to contain another, composition is often the better tool.

Checkpoint: OOP Foundations



Q1

What is printed?

```
ozzy = Dog("Ozzy", 2)
ozzy.birthday()
print(ozzy.age)
```

- A) 2
- B) 3
- C) "Happy Birthday"
- D) Dog years only

Show Answer

Q2

If `Dog` inherits from `Animal` and only overrides `speak()`, what happens when `my_dog.sleep()` runs?

- A) It uses the inherited `sleep()` method
- B) Python raises `MissingDogNapError`
- C) The child class forgets all parent methods
- D) It only works on Tuesdays

Show Answer



Checkpoint: OOP Foundations II

Q3

Which statement is the best description of a class?

- A) One specific dog named Mochi
- B) A blueprint used to create objects with shared structure and behavior
- C) A list that happens to bark
- D) A cookie with admin rights

Show Answer

Q4

In the shelter example, what kind of relationship is shown when `person.pet = dog` stores a dog inside a person object?

- A) Inheritance
- B) Composition / has-a relationship
- C) SyntaxError
- D) Bark-driven development

Show Answer



OOP: Summary

Model

- Class = blueprint
- Object = one instance
- Attributes store state

Behavior

- Methods act on the current object
- `self` refers to that object
- State changes stay bundled with data

Relationships

- Objects can collaborate
- Composition models "has-a"
- Inheritance models "is-a"

When you identify the right nouns first, the verbs often become obvious. That is the practical value of OOP.



Modules, Packages, and Environments

How Python projects stay organized and reproducible

A File Is a Module

Module basics

- Any `.py` file can be imported as a module
- Import executes the module once, then exposes its names
- The module name becomes a namespace
- Namespaces reduce accidental name collisions

```
# my_math.py
PI = 3.14159

def area_of_circle(radius):
    return PI * (radius ** 2)

# main.py
import my_math
print(my_math.area_of_circle(5))
```

The prefix tells readers exactly where a function or constant came from.

Fun Python fact: imported modules are cached in `sys.modules`, so importing the same module again usually reuses it instead of rerunning all its top-level code.



Three Import Patterns

Pattern	Example	Best when	Trade-off
<code>import module</code>	<code>import math</code>	Clarity matters most	More typing at call sites
<code>from module import name</code>	<code>from math import sqrt</code>	You want short local names	Origin becomes less visible
<code>import module as alias</code>	<code>import numpy as np</code>	The module name is long or widely aliased	Still need to learn the alias convention

Default to explicit imports first. Reach for shorter styles when the codebase has a clear convention.



A Folder Can Be a Package

```
todo-app-example/  
├── main.py  
└── todo/  
    ├── __init__.py  
    ├── cli.py  
    ├── manager.py  
    └── model.py
```

Packages help us move from "one big script" to clearly separated responsibilities.

Why package code?

- Related modules live together
- Imports become predictable
- Large apps stay navigable
- `__init__.py` marks package boundaries in beginner-friendly examples

`from todo.cli import main` is clearer than burying CLI, storage, and models in one file.

The Entry Point Guard

```
from todo.cli import main

if __name__ == "__main__":
    main()
```

The file stays tiny: it imports the real CLI entry function, then decides whether to run it.

What those two lines do

- When you run this file directly, `__name__` becomes `"__main__"`
- The condition becomes true, so `main()` starts the app
- If another module imports this file, the condition is false
- That prevents accidental side effects during import

Imports should usually expose names, not unexpectedly run the whole program. The `__main__` guard keeps startup logic explicit.



Three Sources of Python Code

Standard library

`json`, `sys`, `pathlib`

Ships with Python itself.

Third-party packages

`requests`, `numpy`, `rich`

Installed separately from package indexes.

Local application code

`todo.model`, `todo.manager`

Written by your team inside the project.

Important distinction: a Python package in your source tree and a downloadable package on PyPI are related ideas, but they are not the same thing.



Classic Workflow: pip and requirements.txt

```
pip install requests
pip install rich

pip freeze > requirements.txt
pip install -r requirements.txt
```

pip installs packages. requirements.txt records which versions you want to recreate.

What this gives you

- Repeatable installs on another machine
- A shareable dependency list
- A baseline for collaboration and deployment

Limit: requirements.txt does not isolate dependencies by itself. It should be paired with a virtual environment.



Virtual Environments Prevent Dependency Collisions

Why they exist

- Project A might need one package version
- Project B might need another
- Global installs mix everything together
- A virtual environment gives one project its own Python world

```
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt

# later
deactivate
```

Think of a virtual environment as a per-project backpack: interpreter, installed packages, and scripts stay local to that project.

Fun environment fact: a virtual environment is mostly just a folder on disk, which is why removing `.venv` cleanly removes a project's local packages without touching other projects.



Modern Workflow: uv and pixi

⚡ uv

- Great default for Python-first projects
- Creates environments quickly
- Handles install, sync, and run workflows

```
uv venv
uv add requests
uv run python app.py
uv sync
```

📦 pixi

- Useful when conda-level or compiled system dependencies matter
- Works well for scientific or mixed-language stacks
- Can manage both conda and PyPI packages

```
pixi init
pixi add numpy
pixi add --pypi rich
pixi run python app.py
```

Course recommendation: prefer `uv` by default, and reach for `pixi` only when you need non-PyPI or system-level dependencies.



Checkpoint: Modules and Environments

Q5

Which import keeps the origin visible at every use site?

- A) `import math` then `math.sqrt(16)`
- B) `from math import sqrt` then `sqrt(16)`
- C) `from math import *`
- D) Whisper the function name and hope Python hears you

Show Answer

Q6

What problem does a virtual environment primarily solve?

- A) It makes Python run in virtual reality
- B) It isolates project-specific packages from other projects
- C) It replaces `requirements.txt` as documentation
- D) It automatically writes all your docstrings

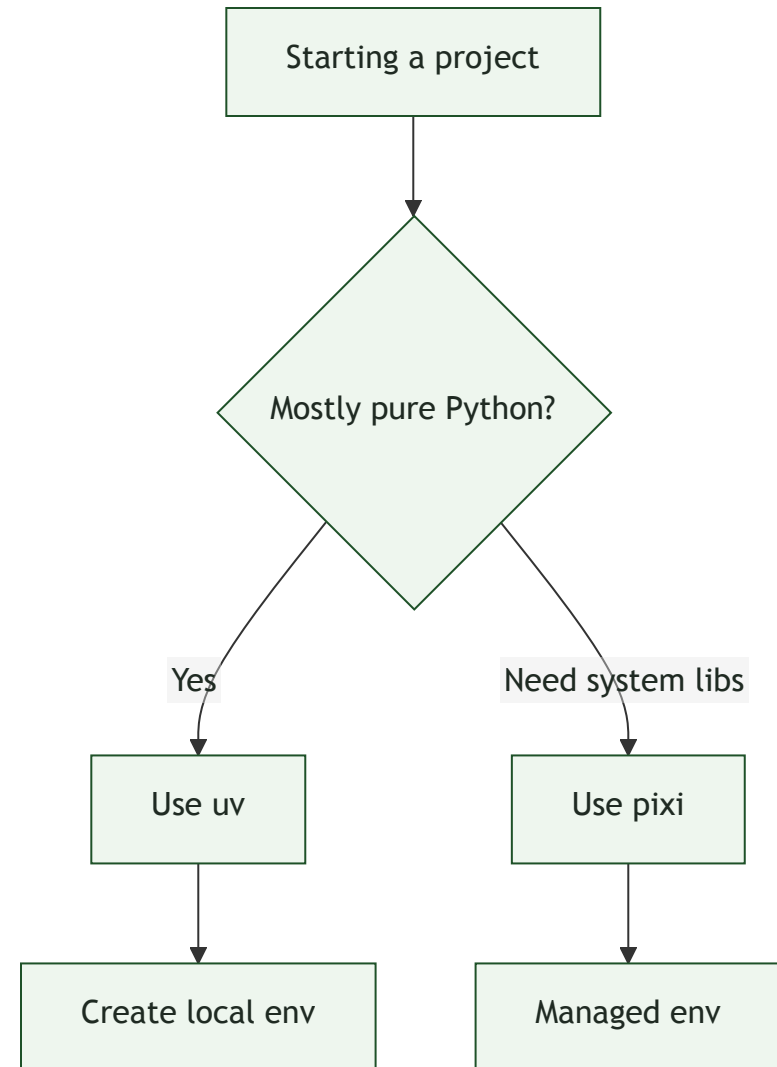
Show Answer



Packages and Environments: Summary

Keep these layers separate

- Module: one Python file
- Package: a folder of related modules
- Installer + environment solve different problems





Type Hints, Style, and Docs

Making code easier for humans and tools to understand

Why Add Type Hints?

```
def add(a, b):  
    return a + b  
  
add(5, 10)  
add("hello", "world")  
add(5, "world")
```

All three calls are syntactically allowed, but not all three make semantic sense.

Hints answer three questions

- What kind of data does this function expect?
- What kind of result should it return?
- What mistakes can tools catch before runtime?

Type hints are a communication tool for humans, editors, and static checkers.

Modern Hint Syntax

```
def process_names(names: list[str]) → None:
    for name in names:
        print(name.upper())

def print_prices(prices: dict[str, float]) → None:
    for item, price in prices.items():
        print(f"{item}: ${price:.2f}")

def greeting(name: str) → str:
    return "Hello, " + name
```

What to notice

- Parameter hint: `name: str`
- Return hint: `→ str`
- Containers can also be typed: `list[str]`

You will still meet older container hints like `List[str]` and `Dict[str, float]` in tutorials and existing code. Modern Python often prefers built-in generics like `list[str]` and `dict[str, float]`, so it is worth recognizing both styles.



Hints Help Tools, Not the Interpreter

```
def add(a: int, b: int) → int:  
    return a + b  
  
result = add(5, "world")
```

Python accepts the function definition. The failure comes later when incompatible values are actually combined.

Where hints shine

- Editor autocomplete and hover help
- Static checking with tools like mypy or pyright
- Clearer APIs for teammates and future you

Misconception: type hints do not make Python "strictly typed". They make code easier to reason about before bugs reach runtime.

Fun Python fact: type hints are available at runtime through `__annotations__`, which is why editors, checkers, and some frameworks can inspect them.



PEP 8 Makes Code Easier to Scan

High-value rules

- 4 spaces for indentation
- Spaces around operators and after commas
- `lowercase_with_underscores` for functions and variables
- `PascalCase` for classes
- Imports at the top, grouped by origin

```
# harder to scan
def CalculateAverage(NUMBERS):
    Total=0
    for n in NUMBERS: Total+=n
    return Total/len(NUMBERS)

# clearer
def calculate_average(numbers):
    total = 0
    for number in numbers:
        total += number
    return total / len(numbers)
```

Docstrings Explain What and Why

```
def calculate_average(numbers: list[float]) → float:
    """Calculate the average of a list of numbers.

    Args:
        numbers: Input values.

    Returns:
        The arithmetic mean.
    """
    return sum(numbers) / len(numbers)
```

Comments vs docstrings

- Comments help developers follow tricky implementation details
- Docstrings describe the public contract of a module, class, or function
- Docstrings are available through `help()` and `__doc__`

A good docstring starts with a one-line summary, then expands only if readers need more context.



Checkpoint: Type Hints, Style, and Docs

Q7

Which statement about type hints is correct?

- A) They are always enforced by the Python interpreter
- B) They help static tools and readers understand intent
- C) They replace unit tests completely
- D) They work only after sacrificing a rubber duck

Show Answer

Q8

Where should a function docstring go?

- A) As the first string inside the function body
- B) At the end of the file, next to the imports
- C) Only in a separate README
- D) Hidden inside a stack trace for dramatic effect

Show Answer



Command-Line Apps

Turning Python scripts into small developer tools

Reading sys.argv

```
import sys

print(sys.argv)

# python calc.py 4 + 5
# ['calc.py', '4', '+', '5']
```

Key facts

- `sys.argv[0]` is the script name/path
- All later items are command-line arguments
- Every argument arrives as a string
- You still need validation and conversion

`sys.argv` is low-level but important. It shows what command-line tools really receive from the shell.

Fun shell fact: quotes are handled before Python starts, so `"buy milk"` can arrive as one argument instead of two.

Real Example: calc.py

```
if len(sys.argv) != 4:
    print("Usage: python3 calc.py <num1> <op> <num2>")
    sys.exit(1)

try:
    num1 = float(sys.argv[1])
    num2 = float(sys.argv[3])
except ValueError:
    print("Error: Both numbers must be numeric.")
    sys.exit(1)
```

What it already handles

- Wrong number of arguments
- Non-numeric input
- Unknown operators

Good review question: what still fails? Division by zero is still possible when the operator is `/`.



When argparse Beats Manual Parsing

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("num1", type=float)
parser.add_argument("op", choices=["+", "-", "*", "/"])
parser.add_argument("num2", type=float)
args = parser.parse_args()
```

Why upgrade?

- Automatic `--help` output
- Built-in type conversion
- Validation rules like `choices=`
- Cleaner code once commands and flags multiply

Learn `sys.argv` first so you understand the raw input. Then use `argparse` when real tools need safer parsing.



Checkpoint: Command-Line Apps

Q9

What does `sys.argv[0]` usually contain?

- A) The first user-supplied argument
- B) The script name or path
- C) The operating system version
- D) The shell's emotional state

Show Answer

Q10

Which bug can still happen in the provided `calc.py` after numeric conversion succeeds?

- A) ZeroDivisionError when dividing by zero
- B) Python forgets what numbers are
- C) The plus sign becomes a string only on leap years
- D) The shell silently rewrites the source file

Show Answer



To-Do App Integration

Putting OOP, packaging, CLI design, and persistence together



Model the Problem With a Task Class

```
class Task:
    def __init__(self, description):
        self.description = description
        self.completed = False

    def complete(self):
        self.completed = True
```

Why start here?

- A task is the core noun in the app
- The object owns its own state
- Behavior like `complete()` belongs next to that state

This is OOP paying off: the app stops being "a pile of dicts" and starts having explicit domain objects.



Persistence Lives in the Manager

```
class TodoManager:
    def add_task(self, description):
        task = Task(description)
        self.tasks.append(task)
        self.save_to_file()

    def load_from_file(self):
        if os.path.exists(self.file_path):
            data = json.load(f)
            self.tasks = [Task.from_dict(item) for item in data]
```

Manager responsibilities

- Own the task collection
- Save and load JSON
- Validate task indices
- Return clear success/error messages

Notice how persistence and validation are centralized. CLI code stays simpler because storage logic is not copied into every command branch.

Fun data fact: JSON knows strings, numbers, booleans, lists, and dictionaries, but not your custom Python classes directly, so apps usually convert objects before saving.



The CLI Is Just the Front Door

```
command = argv[1].lower()

if command == "add":
    self.manager.add_task(description)
elif command == "list":
    tasks = self.manager.list_tasks()
elif command == "complete":
    success, message = self.manager.complete_task(task_index)
```

Separation of concerns

- CLI parses arguments and prints messages
- Manager owns business logic
- Model represents one task

Packages make this architecture easy to read: each module has a job, and imports show who depends on whom.



Package Structure Makes the App Scalable

```
src/todo-app-example/  
├── main.py  
└── todo/  
    ├── __init__.py  
    ├── cli.py  
    ├── manager.py  
    └── model.py
```

Import flow

```
main.py imports todo.cli.  
todo.cli imports TodoManager.  
todo.manager imports Task.
```

That structure mirrors the architecture in your head, which is exactly what good packaging should do.



Checkpoint: To-Do App Integration

Q11

Which class should be responsible for saving tasks to disk?

- A) `Task` , because every task deserves a hard drive
- B) `TodoManager` , because it manages the collection and persistence
- C) `TodoCLI` , because printing text and file I/O are the same thing
- D) `__init__.py` , because it sounds official

Show Answer

Q12

Why is it useful that `main.py` only imports and calls `todo.cli.main()`?

- A) It keeps the entry point tiny and pushes details into the package
- B) It makes Python execute twice as fast instantly
- C) It prevents all future bugs forever
- D) It unlocks admin mode

Show Answer



Lecture 6: Big Picture

Model

- Use classes to represent important nouns
- Let methods own behavior that changes state

Organize

- Split code into modules and packages
- Use isolated environments for reproducible dependencies

Communicate

- Type hints clarify intent
- PEP 8 and docstrings improve maintainability

The to-do app is the bridge between all these ideas: OOP gives structure, packages give shape, and CLI plus persistence make the program actually useful.



Thank You for Your Attention!

"The real question is not whether machines think but whether men do."

-- B. F. Skinner

oop

packages

type hints

cli apps



Python Course 2025/26/2

Lecture 7: Advanced OOP, Testing, and Debugging



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Agenda

- Pythonic class design with special methods and method types
- Multiple inheritance, MRO, ABCs, and where Protocols fit
- Object identity, mutation, argument passing, and CPython memory behavior
- Debugging with tracebacks, breakpoints, pytest, and TDD
- Architecture patterns that keep testing fast and practical

Learning objective

Move from writing classes that merely work to writing classes that are easier to inspect, reason about, test, and extend.

By the end, you can

Choose method and abstraction patterns intentionally, explain Python's object model and argument behavior, debug with evidence, and run a modern quality loop for safer refactors.

Concepts

Code reading

Debugging

Testing

This lecture turns the to-do app from a small program into a maintainable codebase.



Advanced Class Design

Make objects feel natural to use and easier to inspect

Special Methods: `__str__` VS `__repr__`

Two audiences, one object

- `__str__` is for a human-friendly display.
- `__repr__` is for debugging and precise inspection.
- Good `repr` values reduce guesswork in logs, shells, and tracebacks.

Fun Python fact

If `__str__` is missing, `print(obj)` falls back to `__repr__`.

```
class Task:
    def __repr__(self):
        return (
            f"Task(description='{self.description}', "
            f"completed={self.completed})"
        )

    def __str__(self):
        status = "[x]" if self.completed else "[ ]"
        return f"{status} {self.description}"
```

The same object can expose a neat user view and a richer developer view.



Same Object, Two Different Displays

```
task = Task("Buy groceries", completed=True)

print(task)
print(repr(task))
```

Expected output

```
[x] Buy groceries
Task(description='Buy groceries',
completed=True)
```

When each one helps

- User-facing CLI output: `__str__`
- Debugging sessions and logs: `__repr__`
- Shell inspection: typing the variable name often shows `repr(obj)`

Rule of thumb: always implement a useful `__repr__`; add `__str__` when users need a cleaner message.



Class Variables vs Instance Variables

```
class Task:
    task_count = 0

    def __init__(self, description):
        self.description = description
        self.completed = False
        Task.task_count += 1
```

Every task has its own description, but all tasks share one class-level counter.

How to tell them apart

- Instance variable: stored on one object via `self`
- Class variable: stored once on the class itself
- Instances can read class variables, but that does not make them instance-specific

Common mistake: using a mutable class variable for per-object data makes instances accidentally share state.

Class methods: `@classmethod`

```
class Task:
    def __init__(self, description, completed=False):
        self.description = description
        self.completed = completed

    @classmethod
    def from_line(cls, line):
        """Parse '[x] ... ' or '[ ] ... ' into a Task."""
        line = line.strip()
        if line.startswith("[x]"):
            return cls(line[3:].strip(), completed=True)
        if line.startswith("[ ]"):
            return cls(line[3:].strip(), completed=False)
        return cls(line, completed=False)
```

The first parameter is conventionally `cls` (the class object). Use `cls( ... )` so subclasses get the right type.

Why use them?

- **Alternative constructors** (factories): same class, different input shapes.
- Logic that needs the class (hooks, registries, defaults on the class) but not a particular instance yet.
- Call style: `Task.from_line("[x] Done")` reads clearly at the call site.

Contrast: an instance method uses `self` because it works with one object that already exists.



Static methods: `@staticmethod`

```
class Task:
    # ... __init__ and other methods ...

    @staticmethod
    def normalize_description(text):
        """Pure helper: collapse whitespace; no access to class or instance."""
        return " ".join(text.split())

# Call through the class or an instance – same effect
Task.normalize_description("  buy    milk  ")
t = Task("demo")
t.normalize_description("  too  many  spaces  ")

```

Python does not pass `self` or `cls`. The method is just a namespaced function living on the class.

When they fit

- Small utilities that conceptually belong with the type but need no instance or class state.
- Often a module-level function is enough; use `@staticmethod` when grouping under the class helps discovery (API surface).



Choose the Right Method Type

Instance

`self`

Use it when behavior reads or changes one object's state.

Class

`cls`

Use it for factory methods and class-wide behavior.

Static

No implicit first argument
Use it for utility logic tied to the class concept but not to class or instance state.

Kind	First argument	Typical use
Instance	<code>self</code>	Per-object behavior and data
Class	<code>cls</code>	Factories, class-level setup
Static	(none)	Stateless helper on the namespace

Quick rule: state-aware logic belongs in instance or class methods. If a helper needs no state, consider a module-level function first; keep it static only when class-local discoverability helps.



Checkpoint: Advanced Class Design

Q1

If a class defines `__repr__` but not `__str__`, what does `print(obj)` use?

- A) It raises an error immediately
- B) It falls back to `__repr__`
- C) It always prints an empty string
- D) It ignores both methods and prints nothing useful

Show Answer

Q2

After creating two `Task` objects, which value should a shared class variable like `Task.task_count` hold?

- A) One separate count per instance
- B) 2, because the class-level counter is shared
- C) 0, because only instances can change it
- D) A random value based on instance creation order

Show Answer

Q3

Which statement is true about `@classmethod` and `@staticmethod`?

- A) Both pass `self` as the first argument
- B) `@classmethod` passes the class as the first argument; `@staticmethod` passes neither `self` nor `cls`
- C) `@staticmethod` passes `cls`; `@classmethod` does not
- D) You may define only one of the two per class

Show Answer



Inheritance and Abstraction

Reuse behavior carefully and define clear contracts



Multiple Inheritance: Mix Capabilities Carefully

```
class Phone:
    def make_call(self):
        print("Calling... ")

class Camera:
    def take_picture(self):
        print("Snap!")

class Smartphone(Phone, Camera):
    pass
```

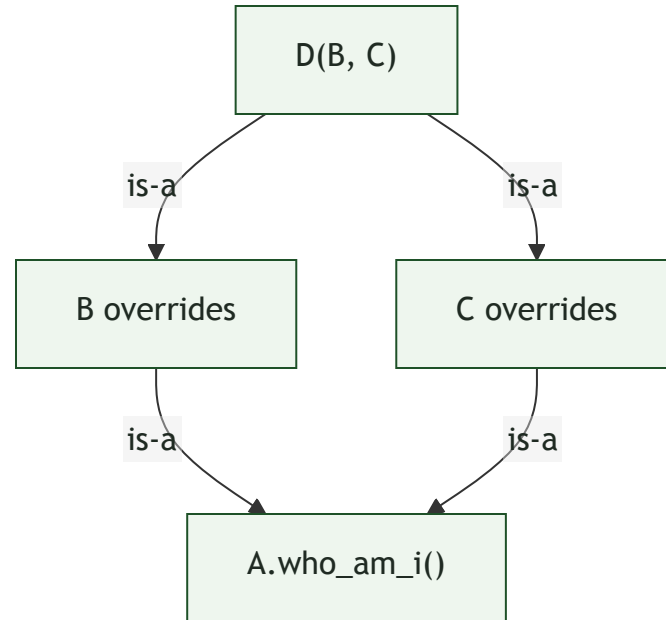
One child class can reuse behavior from more than one parent.

Why people use it

- To mix independent capabilities into one object
- To avoid duplicating helper behavior
- To express "is both X and Y" relationships

Risk: if two parents define the same method, the child needs a predictable rule for deciding which implementation runs.

The Diamond Problem



If `B` and `C` both override the same method from `A`, which one should `D` use? Python solves that with the MRO.



Method Resolution Order (MRO)

```
class D(B, C):  
    pass  
  
print([cls.__name__ for cls in D.__mro__])  
# ['D', 'B', 'C', 'A', 'object']
```

`super()` means "call the next method in this order", not simply "call my parent".

What to remember

- Lookup order is predictable and inspectable
- Parents listed first matter
- Cooperative multiple inheritance depends on each class using `super()` consistently

Fun Python fact

Python computes the MRO with `C3 linearization`, which preserves local left-to-right priority while keeping the order consistent across the hierarchy.



Abstract Base Classes (ABCs) Define Contracts

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass
```

A subclass must implement every abstract method before you can instantiate it.

Why ABCs help

- They document expected behavior as a clear contract
- They fail fast at runtime when implementations are incomplete
- They make dependency injection and test doubles safer to use

In modern codebases, ABCs are often paired with type hints; `typing.Protocol` can complement them when structural typing is enough.



Real Example: Storage ABC in the To-Do App

```
class Storage(ABC):
    @abstractmethod
    def load(self) → list[dict[str, object]]:
        raise NotImplementedError

    @abstractmethod
    def save(self, tasks: list[dict[str, object]]) → None:
        raise NotImplementedError
```

Both `JsonFileStorage` and `MemoryStorage` promise the same interface contract.

Why this matters later

- The manager can work with file storage or memory storage
- Tests become faster because they can avoid real file I/O
- Adding a new backend does not require rewriting the manager

Bridge to testing: a clean interface is what makes dependency injection and isolated unit tests practical.



`typing.Protocol`: Structural (“Duck”) Typing

```
from typing import Protocol, runtime_checkable

class Drawable(Protocol):
    def draw(self) → None: ...

def render(obj: Drawable) → None:
    obj.draw()
```

A type checker treats any object with a matching `draw` signature as a `Drawable`, no need to inherit from the protocol class.

What Protocol is for

- Documents “I need these methods/attributes” for static analysis
- Fits third-party or legacy classes you cannot subclass
- Keeps APIs flexible: implement the shape, not the name on the inheritance line

Runtime: protocols are mostly for type checkers. Add `@runtime_checkable` if you need `isinstance()` checks against the protocol.



ABC vs `Protocol`

	Abstract Base Class (<code>ABC</code>)	<code>typing.Protocol</code>
Kind of contract	Nominal — types must subclass the ABC (or register) to be “officially” part of the family	Structural — if the object has the right methods/attributes, it matches
Who enforces it	Python at instantiation : missing abstract methods → <code>TypeError</code>	Primarily static type checkers (<code>mypy</code> , <code>Pyright</code> , ...); optional <code>isinstance</code> with <code>@runtime_checkable</code>
Typical use	Shared implementation, strict hierarchy, “this is a <code>Shape</code> ”	Callbacks, pluggable dependencies, “this behaves like something drawable”

They solve overlapping problems: ABCs lean on inheritance and runtime safety; protocols lean on shape and checker-time clarity. Many projects use both.



Checkpoint: Inheritance and Abstraction

Q4

In `class D(B, C)`, which parent wins first when both define the same method?

- A) `B`, because it appears first in the child definition
- B) `C`, because it is alphabetically later
- C) `A`, because the ancestor is always safer
- D) Both parent methods run in parallel

Show Answer

Q5

What does `super()` mean in cooperative multiple inheritance?

- A) Call the direct parent by hardcoded class name
- B) Call the next method in the MRO
- C) Skip every parent and call the child again
- D) Jump straight to `object`

Show Answer

Q6

What happens when you instantiate a subclass of `ABC` that still leaves an `@abstractmethod` unimplemented?

- A) Python raises `TypeError` at instantiation time
- B) Python raises `SyntaxError` when the subclass body is parsed
- C) Missing abstract methods silently use the parent's `pass` stubs
- D) It only fails if you run a static type checker

Show Answer



Python Under the Hood

Mental models for objects, references, and memory



Core Philosophy: Everything Is an Object

Values are objects

1 is an int.
"hello" is a str.
[1, 2, 3] is a list.

Functions are objects

```
def foo():  
    pass  
  
print(type(foo))  
# function
```

Classes are objects

```
class Task:  
    pass  
  
print(type(Task))  
# type
```

Fun Python fact

Functions and classes are first-class objects, so you can store them in variables, pass them to functions, and inspect them at runtime.

Identity, Type, and Value

Three things every object has

- **Identity:** which object it is
- **Type:** what operations it supports
- **Value:** the current data it holds

Use `id()` for identity and `type()` for type.

```
name = "Ada"
numbers = [1, 2, 3]

print(type(name))
print(type(numbers))
print(id(numbers))
```

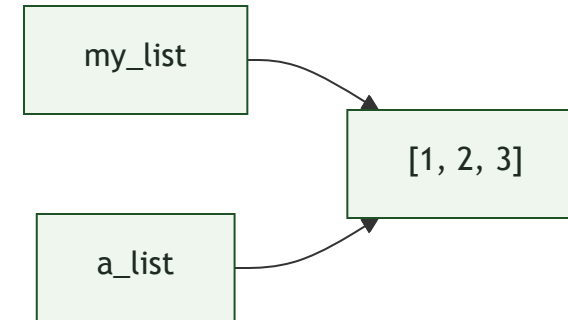
Mutation reminder: immutable objects change by creating a new object; mutable objects can change in place.



How Python Passes Arguments

The right mental model

- Python binds function parameters to existing objects at call time.
- Mutating a shared mutable object is visible to the caller.
- Rebinding a local name points that name to a different object.



The names are different, but at call time they can point to the same mutable object.

Mutation vs Rebinding



Immutable example

```
def add_one_to_num(a_num):  
    a_num = a_num + 1  
  
my_num = 10  
add_one_to_num(my_num)  
print(my_num)  # 10
```

Rebinding `a_num` does not change the caller's original integer.

Mutable example

```
def add_to_list(a_list):  
    a_list.append(4)  
  
my_list = [1, 2, 3]  
add_to_list(my_list)  
print(my_list)  # [1, 2, 3, 4]
```

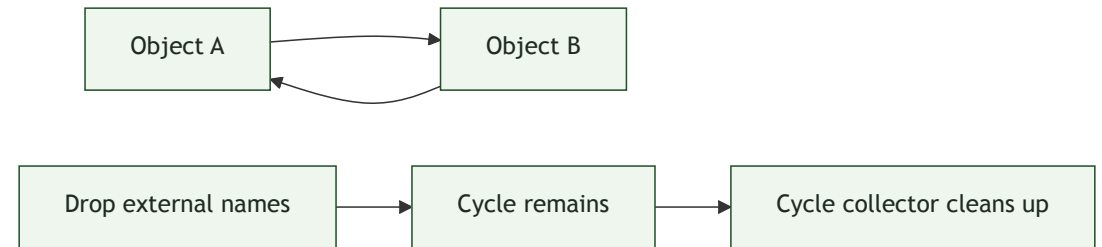
Mutating the shared list changes what the caller sees too.

Key distinction: mutating an object and rebinding a local variable are not the same operation.

Automatic Memory Management (CPython Focus)

Reference counting

- Create `x = [1, 2]` -> one live reference
- Assign `y = x` -> another reference to the same object
- Delete or rebind names -> reference count can decrease
- When references drop to zero, CPython usually reclaims the object quickly



Fun Python fact

CPython combines refcounting with a cyclic garbage collector. Finalization timing can differ across runtimes, so use context managers for critical resource cleanup instead of relying on `__del__`.



Checkpoint: Python Under the Hood

Q7

After calling a function that runs `a_list.append(4)`, what does the caller see?

- A) The original list unchanged
- B) The list with 4 added
- C) A traceback every time
- D) A new tuple with 4 added

Show Answer

Q8

What is the type of a user-defined class object like `Task`?

- A) function
- B) object
- C) type
- D) module

Show Answer



Debugging, Testing, and TDD

Make invisible bugs visible before they escape



Debugging Is a Method, Not a Panic Button

What not to do

- Panic and start changing random lines
- Guess without gathering evidence
- Depend only on endless `print()` calls

What to do instead

- Reproduce the bug reliably
- Shrink to a minimal reproducible case
- Pause execution near suspicious state with breakpoints
- Inspect values and step through the exact code path
- Confirm the fix by rerunning

A debugger turns "I think" into "I can see the actual state now." Use IDE breakpoints or `breakpoint()` when you need a quick in-code pause.



Read Tracebacks from the Bottom Up

Traceback (most recent call last):

```
File "main.py", line 12, in <module>  
    run()
```

```
File "app.py", line 8, in run  
    total = add("2", 3)
```

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

How to parse it

- **Bottom line:** what failed
- **Lines above:** where the failing call came from
- Use the stack to reconstruct the path to the bug
- Python 3.11+ often highlights the exact failing expression

If you only read the first line, you usually miss the actual cause. Start at the bottom, then walk upward through frames.



Bug Hunt: String Key vs Integer Key

```
TASKS_DB = {  
    1: {"desc": "Buy milk"},  
    2: {"desc": "Do laundry"},  
}  
  
def find_task_by_id(task_id):  
    if task_id in TASKS_DB:  
        return TASKS_DB[task_id]  
    return None
```

The user input is a string like `"2"`, but the dictionary keys are integers.

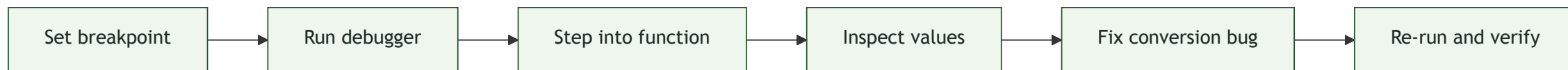
Why the bug happens

- `"2"` and `2` differ in both type and value semantics
- The membership test therefore returns `False`
- Fix near the boundary: convert and validate user input before lookup

Type mismatches are easier to see when you inspect runtime values. A robust fix is `int(task_id)` with `ValueError` handling.



Debugger Workflow in VS Code



Breakpoints are valuable because they freeze the program exactly at the moment where your mental model and reality start to differ.

Why Do We Write Tests?

Tests give us

- Confidence to refactor without fear
- Early bug detection before users hit the bug
- Executable examples of expected behavior
- Pressure toward smaller, more modular code

Tests are not extra paperwork. They are a feedback tool that makes changes safer.

Good design side effect

If something is painful to test, that often reveals a coupling or responsibility problem in the design.



Pytest: Low Ceremony, Fast Feedback

```
from tasks import add

def test_add_positive_numbers():
    assert add(2, 3) == 5

def test_add_negative_numbers():
    assert add(-1, -1) == -2
```

Simple file names and function names are enough for pytest discovery.

Why pytest stays the default

- Plain `assert` statements
- No test class boilerplate required
- Helpful failure messages
- Course workflow: `uv add --dev pytest`, then `uv run pytest -q`

Fun Python fact

`assert` statements can disappear under `python -O`, so use them for tests and invariants, not for user-facing validation logic.



Fixtures and `tmp_path` Keep Tests Clean

```
def test_save_tasks_to_file(tmp_path):  
    test_file = tmp_path / "test_tasks.json"  
    storage = JsonFileStorage(str(test_file))  
    storage.save([{"description": "Learn", "completed": False}])  
    assert test_file.exists()
```

Each test gets its own temporary directory and fresh file path.

Why `tmp_path` is so useful

- Isolation: tests do not pollute real files
- Cleanup: pytest removes temporary directories automatically
- Portability: works across macOS, Linux, and Windows
- Modern style: `tmp_path` is a `pathlib.Path` object

Defining fixtures: `@pytest.fixture`

```
import pytest

@pytest.fixture
def empty_task_list():
    return []

def test_len_empty(empty_task_list):
    assert len(empty_task_list) == 0
```

The parameter name must match the fixture function name; pytest wires it in before the test runs.

What fixtures are for

- Reuse setup that many tests need (data, clients, config)
- Keep each test short: arrange complex state once, inject it
- One fixture, one responsibility—compose small fixtures instead of one giant setup

Teardown

Use `yield` in the fixture body: code after `yield` runs after the test, even if the test fails.



`conftest.py` and fixture scope

```
# tests/conftest.py – discovered automatically
import pytest

@pytest.fixture(scope="module")
def shared_config():
    return {"debug": True}
```

Place shared fixtures in `conftest.py` under `tests/` (or a package) so every test module in that tree can request them by name.

Scopes (default: `function`)

Scope	Created
<code>function</code>	Once per test (safest default)
<code>class</code> / <code>module</code>	Shared across tests in that class or file
<code>session</code>	Once for the whole run, watch mutable shared state

Good default practice

- Start with `scope="function"` ; widen only when setup is expensive and shared state is clearly safe.
- Wider scopes + mutable objects: one test can affect another via the same fixture instance.
- Pytest's fixture model scales across scopes, but explicit, easy-to-read fixture wiring is usually better than clever scope tricks.



Fixture best practices

Do

- Name fixtures after the resource they provide (`db_connection`, not `setup`)
- Prefer `tmp_path` for files; use `monkeypatch` for env vars and small fakes
- Build bigger setups from smaller fixtures (pytest resolves the dependency graph)
- Use `params` on a fixture when every test needs the same matrix of inputs

Avoid

- `autouse=True` unless every test in the file truly needs it, hidden dependencies are hard to read
- Session-scoped fixtures that mutate global state without teardown
- Fixtures that assert or “test” things; keep assertions inside tests only
- Over-nesting: if setup is unreadable, extract a helper or factory function

Rule of thumb

A reader should see every dependency in the test signature. Explicit fixture parameters beat magic global setup.



What a Good Test Looks Like

Arrange

Create the objects, input data, and environment for the test.

Act

Run the one behavior you actually want to verify.

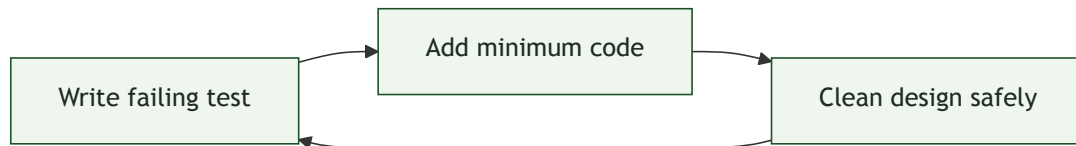
Assert

Check observable outcomes with clear expectations.

Quality checklist: keep tests deterministic (no hidden time/network dependencies), focus each test on one behavior, and use parametrization when only inputs vary.



TDD: Red - Green - Refactor



Why teams still use it

- Forces you to define behavior before implementation
- Builds a safety net before refactoring
- Encourages smaller steps and tighter feedback loops
- Works best for design-heavy or bug-prone behavior, not every trivial line

TDD is a tool, not a religion. Use it deliberately where fast feedback improves design decisions.



TDD Example: Add Priority to Task

```
def test_task_creation_with_priority():  
    task = Task("Buy milk", priority=1)  
    assert task.description == "Buy milk"  
    assert task.priority == 1
```

The first run fails because the feature does not exist yet.

Then we improve it in stages

- **Red:** failing test proves the gap
- **Green:** add the minimal constructor change
- **Refactor:** set default priority and validate allowed range

Notice how the test turns a vague idea like "support priority" into a precise, checkable contract.



Checkpoint: Debugging, Testing, and TDD

Q9

When you read a traceback, where should you usually start?

- A) The bottom line with the exception and message
- B) The first blank line you can find
- C) Somewhere in the middle for suspense
- D) The very first frame only

Show Answer

Q10

Why is `tmp_path` such a good fit for file-based tests?

- A) It gives each test a clean temporary directory
- B) It avoids collisions with real project files
- C) It disables all file errors forever
- D) It makes every OS use the same path format

Show Answer



Project-Level Craftsmanship

Architecture choices that make real testing possible



Why Better Architecture Helps Testing

Before refactoring

- Task logic and file I/O lived too close together
- Side effects leaked into places that should stay pure
- Changing storage details risked breaking unrelated behavior

After refactoring

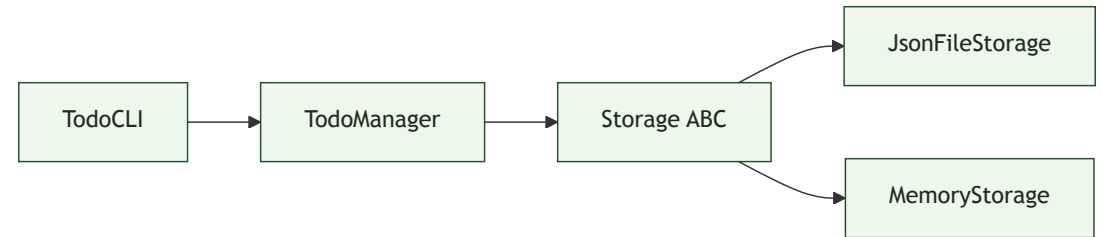
- `TodoManager` focuses on task behavior
- Storage implementations own persistence side effects
- Alternative backends become easy to swap in for tests

Good architecture is not abstract decoration. It directly lowers the cost of testing and change.

Storage, Manager, and CLI Fit Together

```
class TodoManager:
    def __init__(self, storage: Storage):
        self.storage = storage
        self.tasks = []
        self._load_tasks()
```

The manager receives a storage dependency instead of creating one internally.



Dependency injection lets the same manager work with production storage and test-friendly storage.



Testing File Storage Safely

```
def test_save_tasks_to_file(tmp_path):  
    test_file = tmp_path / "test_tasks.json"  
    storage = JsonFileStorage(str(test_file))  
    storage.save([  
        {"description": "Learn Python", "completed": False}  
    ])  
    assert test_file.exists()
```

What this test suite covers

- Basic save/load behavior
- Non-existent files
- Round-trip integrity
- Corrupted JSON and bad data shapes
- Formatting and independence of storage instances
- A foundation for parametrized edge-case tests



Live Coding and Next Iterations

Iteration 4 focus

- Create the `Storage` ABC
- Extract `JsonFileStorage`
- Add `MemoryStorage`
- Inject storage into the manager

Iteration 5 focus

- Write TDD-style tests for task rename behavior
- Write storage tests with `tmp_path`
- Use tests to protect further refactoring

Iteration 6 quality loop

- `uv sync`
- `uv run ruff check .`
- `uv run pyright`
- `uv run pytest -q`

The app is now big enough that quality tools are part of daily development, not a final cleanup step.



Lecture 7: Big Picture

Pythonic Classes

- Use special methods to improve readability
- Separate instance state from class-wide state
- Choose the right method type intentionally

Mental Models

- Everything is an object
- Arguments pass object references, not magical copies
- Memory management mixes refcounts and cycle cleanup

Craftsmanship

- Use debuggers to inspect evidence
- Keep tests deterministic and fast
- Automate lint, type, and test checks in CI

Today's core message: good OOP design, clear mental models, and disciplined testing reinforce each other. Better structure makes bugs easier to find and safer to prevent.



Thank You for Your Attention!

*"Programs must be written for people to read, and only
incidentally for machines to execute."*

Carry the debugger, the tests, and the design mindset into the next version of
your app.



Python Course 2025/26/2

Lecture 8: Web APIs, REST, DTOs, and Dependency Injection



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Agenda

- How a web app differs from a one-shot CLI program
- HTTP requests, responses, and REST-style API design
- Why we separate domain models, DTOs, services, and routes
- FastAPI, Pydantic, OpenAPI docs, and modern dependency patterns
- Testing APIs with `pytest`, `TestClient`, and dependency overrides

Learning objective

Move from "I can write Python code" to "I can expose that logic as a clean, testable web API with explicit contracts."

By the end, you can

Explain the request-response model, choose sensible REST paths and status codes, separate transport objects from business logic, and test an API without launching a real browser.

Web

HTTP

FastAPI

Pydantic

Testing

Lecture 8 turns our to-do app into a service other programs can talk to.



From CLI to Web Services

The same Python ideas, but a very different runtime model



Mental Model Shift: CLI App vs Web App

>_ CLI app

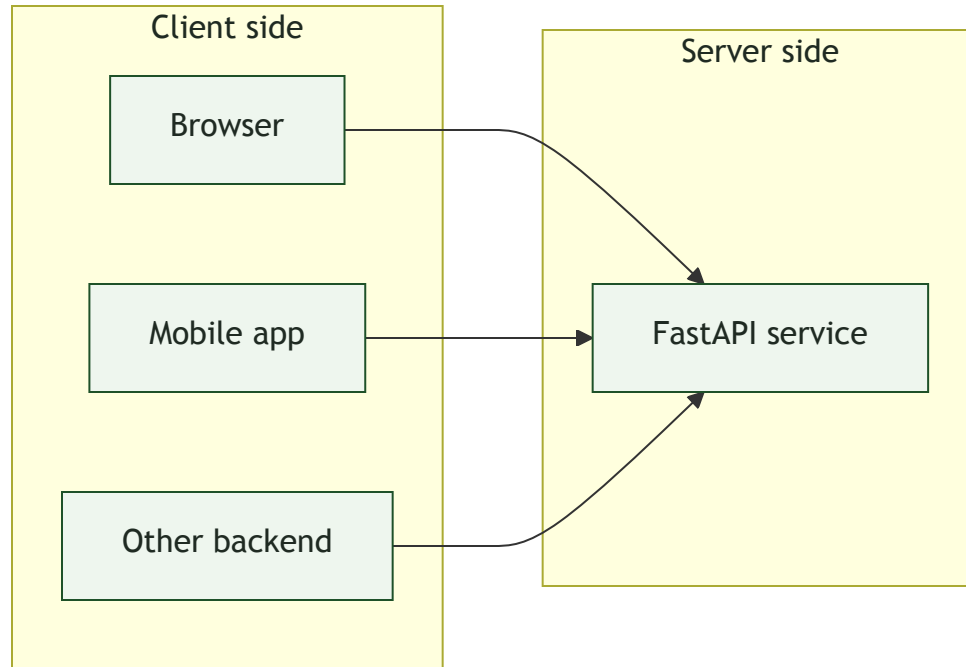
- Starts when `python ...` is executed
- Input often comes from arguments, files, or terminal prompts
- Output is usually text printed for one human user
- Stops as soon as the command finishes

☰ Web service

- Starts once, then keeps listening for requests
- Input arrives as HTTP requests from many clients
- Output is usually structured data such as JSON
- Must survive many users, failures, and repeated calls

Same Python, different job: instead of solving one problem and exiting, the program becomes a long-running service.

The Client-Server Model



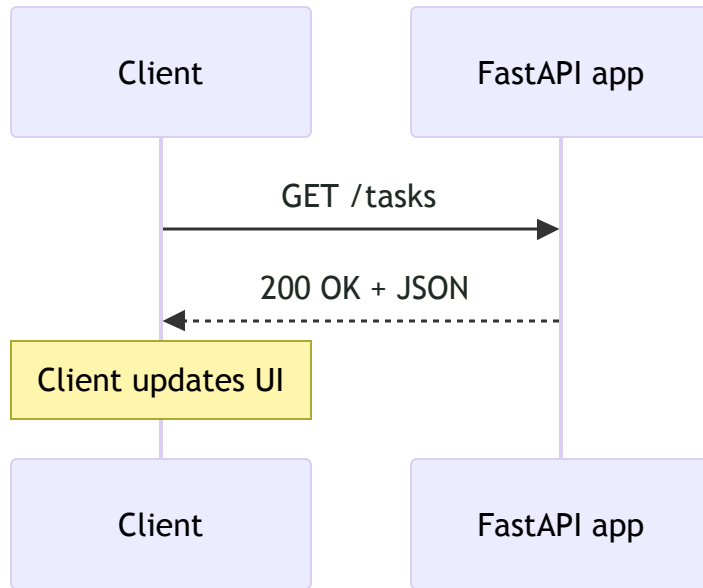
Who plays which role?

- **Client:** asks for data or sends commands
- **Server:** validates input, runs logic, returns a response
- **Network:** carries requests and responses between them

One API can serve a browser, a mobile app, and another service at the same time.



HTTP: One Request, One Response



Key HTTP idea

- The client sends a request message
- The server handles it and sends back one response
- HTTP itself is stateless: each request should stand on its own

Fun web fact

Opening one modern page can trigger dozens of HTTP requests for HTML, CSS, JavaScript, fonts, and images.



Checkpoint: From CLI to Web Services

Q1

Which statement best describes the main runtime difference between a CLI tool and a web API?

- A) A web API usually keeps running and handles many requests over time.
- B) A CLI tool must always return JSON.
- C) A web API can only be used by browsers.
- D) A CLI tool works only on Tuesdays.

Show Answer

Q2

If HTTP is stateless, what should a protected request generally include?

- A) Enough information for the server to understand and authorize that request.
- B) A memory dump of the previous request.
- C) The entire source code of the client app.
- D) A handwritten apology letter.

Show Answer



REST and HTTP Design

Turning nouns, verbs, and status codes into a usable contract

What Is an API?

Short definition

An API is an interface that lets one program ask another program to do something in a defined, predictable way.

Interface

Contract

Communication

Restaurant analogy

- **Client:** the customer placing an order
- **API:** the waiter carrying structured requests
- **Server logic:** the kitchen doing the real work
- **Response:** the finished dish delivered back

The client does not need to know how the kitchen works internally. It only needs to know the menu and the rules.



Why APIs Matter

Separation

Frontends and backends can evolve independently as long as the contract stays stable.

Reuse

The same service can power a web UI, mobile app, tests, or another backend.

Interoperability

Programs written in different languages can still cooperate over shared protocols.

Real examples: payment gateways, map services, login providers, and AI APIs all let you reuse complex systems instead of rebuilding them.



REST: Resources, Methods, and a Contract

Think in resources

- `/tasks` means the whole collection
- `/tasks/42` means one specific task
- The URL names the thing; the HTTP method names the action

Fun API fact

The term `REST` was coined by Roy Fielding in his 2000 PhD dissertation.

Path	Method	Meaning
<code>/tasks</code>	<code>GET</code>	List tasks
<code>/tasks</code>	<code>POST</code>	Create a new task
<code>/tasks/42</code>	<code>GET</code>	Read one task
<code>/tasks/42</code>	<code>PUT</code>	Replace one task (full update)
<code>/tasks/42</code>	<code>PATCH</code>	Partially update one task
<code>/tasks/42</code>	<code>DELETE</code>	Remove one task

Real systems are often pragmatically RESTful: consistency matters more than purity debates.



Good Paths Feel Like Nouns

Prefer	Avoid	Why
GET /tasks	GET /getTasks	The method already says "get".
POST /tasks	POST /createTask	Use the collection path for creating a new resource.
PATCH /tasks/42	POST /updateTask/42	Let the method express update semantics.
DELETE /tasks/42	GET /deleteTask/42	Do not hide destructive actions behind read-looking URLs.

Rule of thumb: the path identifies the resource, the method tells the server what kind of action the client wants.

Anatomy of an HTTP Request

```
POST /tasks HTTP/1.1
Host: api.example.com
Content-Type: application/json
Authorization: Bearer <token>

{
  "description": "Buy milk"
}
```

Requests have a start line, headers, a blank line, and an optional body.

Read it top to bottom

- **Method:** what action is requested
- **Path:** which resource is targeted
- **Headers:** metadata such as format or auth
- **Body:** extra data, often JSON

Common pattern: GET often has no body, while POST and PATCH usually do.



HTTP Responses and Status Codes

```
HTTP/1.1 201 Created
Content-Type: application/json

{"id": 1, "description": "Buy milk"}
```

Status line + headers + optional body; 204 means success with no body.

Same API, other outcomes

GET /tasks/999 — nothing there

```
HTTP/1.1 404 Not Found
Content-Type: application/json

{"detail": "Task not found"}
```

POST /tasks with bad JSON shape

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{"detail": [{"loc": ["body", "description"], "msg": "field required"}]}
```

DELETE /tasks/1 — gone

```
HTTP/1.1 204 No Content
```

First digit = family: 2 success · 4 fix auth/request · 5 server / temporary outage.

Code	Fam.	Typical use
200	2xx	Read or update OK
201	2xx	Created (Location often set)
204	2xx	Success, no response body
400	4xx	Bad / unusable request
401	4xx	Not authenticated
403	4xx	Authenticated, not allowed
404	4xx	Unknown path or ID
409	4xx	State conflict (duplicate, version)
422	4xx	Validation failed (FastAPI / Pydantic)
429	4xx	Rate limited
500	5xx	Unexpected server error
503	5xx	Overload or maintenance

Codes are part of the API contract. Caching & redirects (3xx , e.g. 304) — next slide.

Caching & redirects

Not a main topic in this course, but you will see these in production traces, CDNs, and auth flows — useful vocabulary to recognize.

3xx redirects

- The `Location` header tells the client where to continue; browsers follow it; HTTP libraries usually expose it for you to act on.
- `301` / `308` — permanent move (new canonical URL). `302` / `307` — temporary (e.g. OAuth login hand-off).

Caching & `304`

- `Cache-Control` and `ETag` control how long a response may be reused and how to re-check it cheaply.
- `304 Not Modified` — “your cached copy is still valid”; no new response body (common on repeated `GET`s behind CDNs).

Typical for static assets, public read APIs, and edge networks — less central than `2xx` / `4xx` for the small JSON APIs we focus on here.



Checkpoint: REST and HTTP

Q3

You want to create a new task in a REST-style API. Which option is the best fit?

- A) POST /tasks
- B) GET /createTask
- C) GET /tasks/new
- D) DELETE /tasks for dramatic effect

Show Answer

Q4

A client sends JSON with the wrong shape and FastAPI rejects it during validation. Which status code is most likely?

- A) 200 OK
- B) 404 Not Found
- C) 422 Unprocessable Entity
- D) 418 I'm a teapot, unless the JSON was brewed properly

Show Answer



Layered Architecture with FastAPI

Keep business logic reusable, transport shapes explicit, and web concerns at the edge

Hello, World! with FastAPI

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello, World!"}
```

A plain dictionary becomes JSON automatically.

What to notice

- `app = FastAPI()` creates the ASGI application object
- `@app.get("/")` binds a path and HTTP method
- The function body contains normal Python code

Modern note: use `def` for simple sync code and `async def` when you are actually awaiting async I/O.



Run It, Explore It, Reuse the Schema

Modern setup commands

```
uv add fastapi "uvicorn[standard]"  
uv run uvicorn main:app --reload
```

Useful local URLs

- `http://127.0.0.1:8000/` - your endpoint
- `http://127.0.0.1:8000/docs` - Swagger UI
- `http://127.0.0.1:8000/openapi.json` - raw schema

Why `/docs` is special

- FastAPI inspects routes and types
- It builds an OpenAPI schema automatically
- The docs page becomes a live, testable menu of your API

Fun FastAPI fact

The same OpenAPI schema behind `/docs` can also drive ReDoc pages and generated client SDKs.

Path parameters

Pieces of the URL become function arguments. Types in the signature drive parsing and validation.

```
@app.get("/items/{item_id}")
def read_item(item_id: int):
    return {"item_id": item_id}
```

`/items/42` → FastAPI passes `42` as an `int`. Non-numeric paths yield `422`.

Try the same request two ways

curl

```
curl http://127.0.0.1:8000/items/42
```

Swagger UI

- Open `/docs` → find `GET /items/{item_id}`
- **Try it out** → enter `item_id` → **Execute**

Query parameters

After `?` in the URL: `name=value` pairs, separated by `&`. Optional parameters get defaults from the function signature.

```
@app.get('/items')
def list_items(skip: int = 0, limit: int = 10):
    return {'skip': skip, 'limit': limit}
```

Omitted query args use the defaults (`0` and `10`).

Examples

```
curl "http://127.0.0.1:8000/items"
curl "http://127.0.0.1:8000/items?skip=5&limit=2"
```

In `/docs`, query fields appear as form inputs next to **Execute**.

Request body (JSON)

For `POST`, `PUT`, `PATCH`, ... declare a Pydantic model. FastAPI reads JSON, validates it, and passes a model instance to your function.

```
from pydantic import BaseModel

class ItemCreate(BaseModel):
    name: str
    price: float

@app.post("/items/")
def create_item(item: ItemCreate):
    return {"saved": item.model_dump()}
```

curl with JSON body

```
curl -X POST "http://127.0.0.1:8000/items/" \
  -H "Content-Type: application/json" \
  -d '{"name":"Milk","price":1.2}'
```

`/docs` shows a JSON editor for the body and builds the correct `Content-Type` header for you.

Path, query, and body together

One handler can take a path segment, query flags, and a JSON body. FastAPI decides each argument's source from types and defaults.

```
@app.put("/items/{item_id}")
def update_item(
    item_id: int,
    item: ItemCreate,
    dry_run: bool = False,
):
    if dry_run:
        return {"would_update": item_id, "data": item.model_dump()}
    return {"item_id": item_id, "saved": item.model_dump()}
```

item_id from path, item from body, dry_run from query.

curl

```
curl -X PUT \
  "http://127.0.0.1:8000/items/7?dry_run=true" \
  -H "Content-Type: application/json" \
  -d '{"name": "Tea", "price": 2.5}'
```

Swagger UI lists path, query, and body sections in one panel — useful to see the full contract.



When input is invalid

Bad types or missing required fields produce `422 Unprocessable Entity` with a structured `detail` list — before your route body runs.

```
# path: not an int
curl http://127.0.0.1:8000/items/not-a-number

# body: wrong types / missing field
curl -X POST "http://127.0.0.1:8000/items/" \
  -H "Content-Type: application/json" \
  -d '{"name":"Milk","price":"cheap"}'
```

Why this helps at intro level

- `/docs` shows the error response shape for failed validation
- Same rules apply whether the client is curl, a browser app, or tests
- You fix the contract (types and models), not ad-hoc string checks in every route

Next: make parameters **explicitly typed** and attach **OpenAPI-friendly metadata** with `Annotated` + `Query` / `Path`.



Best practice: keep parameters typed

Use real Python types on path, query, and body parameters — not generic strings you parse by hand.

Why it matters

- **Validation** — bad values become `422` with a clear `detail` payload
- **OpenAPI** — `/docs` shows the right types, formats, and constraints
- **Tooling** — editors and type checkers understand your handler signatures

```
# Prefer
@app.get('/users/{user_id}')
def get_user(user_id: int, active: bool = True):
    ...

# Avoid inventing ad-hoc parsing
def get_user(user_id: str): # every caller sends str
    uid = int(user_id) # manual, easy to get wrong
```

Let FastAPI coerce and validate; reserve manual parsing for rare edge cases.



Annotated + Query(...)

Put extra rules and documentation on **query** parameters without hiding the real Python type. This is the style FastAPI recommends today.

```
from typing import Annotated
from fastapi import Query

@app.get('/items')
def list_items(
    skip: Annotated[
        int,
        Query(ge=0, description='Offset into the list'),
    ] = 0,
    limit: Annotated[
        int,
        Query(le=100, ge=1, description='Page size'),
    ] = 10,
):
    return {'skip': skip, 'limit': limit}
```

What you gain

- `ge` / `le` — numeric bounds checked before your code runs
- `description` — appears in Swagger UI next to the field
- The parameter still reads as `int` for types and IDEs

Older tutorials use `limit: int = Query(10, le=100)`; `Annotated` keeps the type in front and scales better as metadata grows.



Annotated + Path(...)

Path segments use the same pattern: type first, then Path(...) for constraints and docs.

```
from typing import Annotated
from fastapi import Path

@app.get('/items/{item_id}')
def read_item(
    item_id: Annotated[
        int,
        Path(ge=1, description='Primary key of the item'),
    ],
):
    return {'item_id': item_id}
```

Same idea as Query

- Path metadata shows up under the path parameter in /docs
- Combine with Query on the same route when you need both
- For bodies, Pydantic models stay the default; use Body() inside Annotated only when you need extra schema tweaks



Who Does What: FastAPI, Uvicorn, Pydantic, OpenAPI

FastAPI

Defines routes, parses parameters, calls dependencies, and builds responses.

Pydantic

Validates request data and helps serialize well-defined response shapes.

Uvicorn

The ASGI server that actually listens on a port and forwards HTTP traffic to the app.

OpenAPI

The machine-readable schema behind `/docs`, `/redoc`, and client generation.

Domain Model vs DTO

Domain model

- Represents business state and behavior
- Can have methods like `mark_complete()`
- Should not care about HTTP or JSON

DTO

- Defines the data shape coming in or going out
- Great for validation and serialization
- Usually has fields, not business behavior

Kitchen analogy: the domain model is the meal being cooked; DTOs are the order slip and plated dish used for transport.



Our 4-Layer To-Do App

Layer 1

Web

`main.py`

HTTP in, JSON out

Layer 2

Service

`task_manager.py`

Coordinates use cases

Layer 3

Domain

`task.py`

Core business behavior

Layer 4

DTOs

`models.py`

Validated transport
shapes

The payoff is reuse: the same service/domain logic can power a CLI, a web API, and tests without being rewritten for each interface.

Modern DTOs with Pydantic v2

```
from pydantic import BaseModel, ConfigDict

class TaskCreate(BaseModel):
    description: str

class TaskRead(BaseModel):
    model_config = ConfigDict(from_attributes=True)

    id: int
    description: str
    completed: bool
```

`from_attributes=True` is useful when you want to validate directly from an attribute-based domain object.

Why this helps

- Incoming JSON gets checked automatically
- Response data stays predictable
- Bad payloads fail early, before business logic runs

Fun Python tooling fact

Pydantic v2 is powered by Rust-based `pydantic-core`, one reason validation feels so fast.



Checkpoint: Architecture and FastAPI

Q5

Which layer should know about HTTP status codes such as `404` or `201`?

- A) The web layer
- B) The domain model
- C) The database only
- D) The layer with the fanciest filename

Show Answer

Q6

What does `uv run uvicorn main:app --reload` mean?

- A) Run the object named `app` from `main.py` with auto-reload.
- B) Compile Python into C and reboot the laptop.
- C) Export the API directly to Kubernetes.
- D) Ask Uvicorn politely to find the code on its own.

Show Answer



Dependency Injection and Testing

Make route handlers smaller, replace dependencies in tests, and keep the seams explicit



Dependency Injection: Tight vs Loose Coupling

Without DI

```
@app.get("/tasks")
def list_tasks():
    manager = TaskManager()
    return manager.get_all_tasks()
```

With DI

```
def get_manager():
    return task_manager

@app.get("/tasks")
def list_tasks(manager = Depends(get_manager)):
    return manager.get_all_tasks()
```

Why better? The route no longer decides how the manager is constructed, which makes substitution in tests much easier.



Modern Dependency Signatures with `Annotated`

```
from typing import Annotated
from fastapi import Depends

ManagerDep = Annotated[TaskManager, Depends(get_manager)]

@app.get("/tasks", response_model=list[TaskRead])
def list_tasks(manager: ManagerDep):
    return manager.get_all_tasks()
```

The classic default-value style still works, but `Annotated` is the modern recommended form.

Why use it?

- Keeps the real type visible
- Makes reusable parameter aliases easy
- Works nicely with more complex parameter metadata later

Fun FastAPI fact

FastAPI normally caches dependency results per request, so the same dependency is usually resolved once and reused within that request.



The Decorator Is Part of the Contract

```
@app.post(  
    "/tasks",  
    response_model=TaskRead,  
    status_code=201,  
)  
def create_task(task_in: TaskCreate, manager: ManagerDep):  
    return manager.create_task(task_in)
```

What the decorator communicates

- `POST /tasks`: create something in the collection
- `response_model=TaskRead`: output contract
- `status_code=201`: success means "created", not just "ok"

The route body can return a domain object; FastAPI handles validation/serialization for the response contract.



Where Does Each Parameter Come From?

Path

`task_id: int`

Matches `{task_id}` in the URL and is converted to an integer.

Body

`task_in: TaskCreate`

FastAPI reads JSON from the request body and validates it with Pydantic.

Dependency

`manager: ManagerDep`

FastAPI calls the provider and injects the returned object.

Big idea: type hints and metadata are not just documentation here; FastAPI actively uses them to build the request-handling pipeline.



Tests need a clean seam: why DI matters

Production

One long-lived `TaskManager` holds tasks in memory for every request. Routes get it through

`Depends(get_manager)`.

That is correct for a demo server — but tests must not all pile into the same store or mutate the object your real app would use.

What we want in tests

- A **fresh** manager per test (predictable empty state)
- No accidental coupling: test A's tasks must not leak into test B
- Room to swap a **fake** or **stub** later without rewriting routes

Dependency injection gives FastAPI a named hook (`get_manager`). In tests we **replace** that hook — the route code stays unchanged.



Replace the provider: `dependency_overrides`

```
from fastapi.testclient import TestClient
import pytest

@pytest.fixture
def client():
    test_manager = TaskManager()
    app.dependency_overrides[get_manager] = lambda: test_manager
    with TestClient(app) as test_client:
        yield test_client
    app.dependency_overrides.clear()
```

While the fixture is active, every route that depended on `get_manager` receives `test_manager`. After the test, overrides are cleared so the real app is not left in a weird state.

What `TestClient` does

- Calls your ASGI app in-process — no open port required
- Real request/response cycle: paths, query, body, status codes
- Works with the same `Depends` graph as production

The fixture's teardown (`clear()`) matters when you run many tests: each one gets a clean override window.



2026 Testing Notes: Async Tests and Lifespan

```
from httpx import ASGITransport, AsyncClient

@pytest.mark.anyio
async def test_root():
    async with AsyncClient(
        transport=ASGITransport(app=app),
        base_url="http://test",
    ) as client:
        response = await client.get("/")
```

Use this pattern when the test itself is async and needs to await other async work too.

Modern rules of thumb

- Use `TestClient` for normal sync tests
- Use `AsyncClient` + `ASGITransport` for async tests
- Prefer app `lifespan` handlers over older `@app.on_event( ... )` patterns

Wrapping `TestClient(app)` in a `with` block also ensures startup and shutdown logic runs around the test session.



Checkpoint: Dependency Injection and Testing

Q7

Why is dependency injection especially helpful in tests?

- A) It lets us replace real dependencies with controlled test doubles.
- B) It makes HTTP disappear entirely.
- C) It guarantees zero bugs forever.
- D) It summons extra RAM from the cloud.

Show Answer

Q8

You are writing an `async def` test that also awaits other async database calls. Which client pattern fits best?

- A) `AsyncClient` with `ASGITransport(app=app)`
- B) A random browser tab and optimism
- C) `print()` statements only
- D) `DELETE /tests`

Show Answer



Lecture Summary

Web mindset

- Services stay alive and listen
- Clients and servers talk via HTTP
- Each request should stand on its own

API contract

- Paths name resources
- Methods express actions
- Status codes and DTOs make expectations explicit

Maintainability

- Keep HTTP at the edge
- Inject dependencies instead of hard-wiring them
- Test through the API surface with controlled overrides

Core message: clean contracts, clear boundaries, and testable seams make web APIs easier to evolve safely.



Thank You for Your Attention!

"APIs should be easy to use and hard to misuse."

— Joshua Bloch, Effective Java



Python Course 2025/26/2

Lecture 9: AI, ML, Data Science, and Modern DataFrames



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Agenda

- Separate AI, ML, deep learning, generative AI, and data science
- See how data projects move from questions to deployed results
- Map the modern Python stack: JupyterLab, pandas, Polars, plotting, ML
- Use current pandas 3.0 patterns for ingestion, null handling, and safe mutation
- Understand when Polars is worth the mental-model shift

Learning objective

Move from buzzwords and notebooks-by-habit to a clearer mental model of modern data work: what the terms mean, which tools fit which job, and how to write safer dataframe code.

By the end, you can

Explain the AI/ML/data science landscape, choose between pandas and Polars for common scenarios, and recognize the main pandas 3.0 habits you should actually trust in 2026.

Concepts

Workflow

JupyterLab

pandas 3.0

Polars

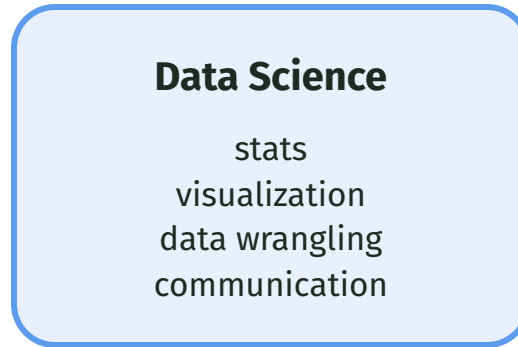
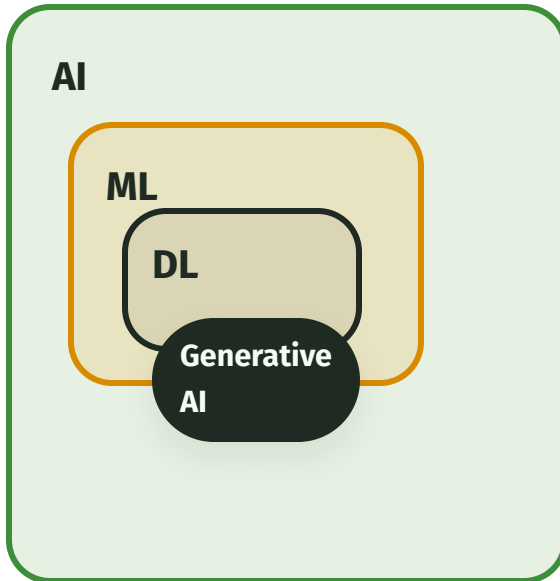
Lecture 9 is the bridge from "AI words" to hands-on dataframe work.



AI, ML, and Data Science

A mental map before we touch the tools

The Taxonomy: Related, But Not Identical



Data science overlaps with AI/ML when it uses predictive or generative models.

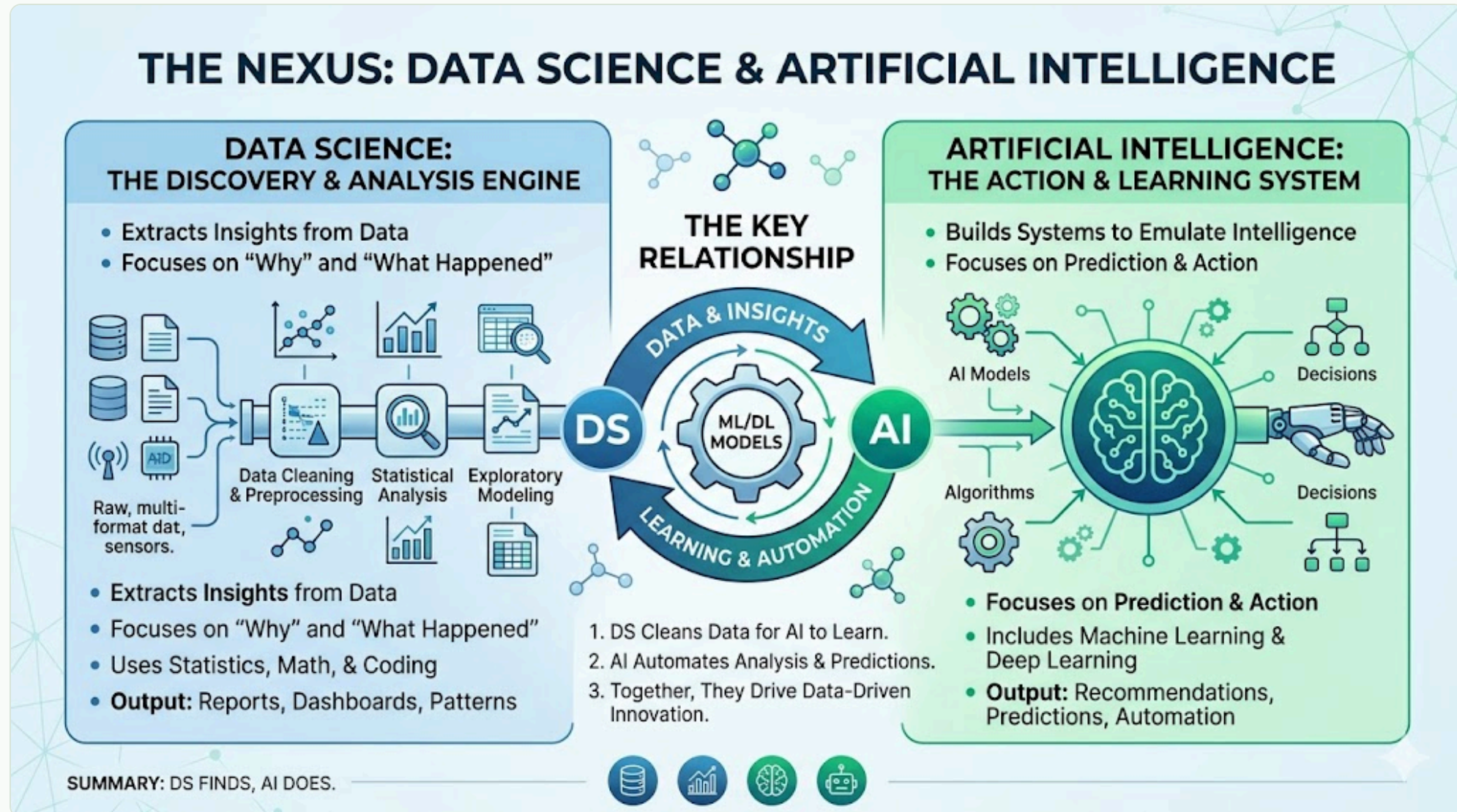
Use the terms precisely

- **AI:** the broad goal of building systems that act intelligently
- **ML:** AI methods that learn patterns from data
- **DL:** ML with layered neural networks
- **Generative AI:** systems that produce new content; flagship tools are usually deep nets, but classical or hybrid generators (templates, grammars, statistical models) still count
- **Data Science:** framing questions, preparing data, analyzing results, and communicating decisions, often using ML as one tool

Fun history fact

The term **Artificial Intelligence** was coined for the Dartmouth workshop in 1956, not in the deep-learning era people usually associate with AI.

Data Science and AI (reference diagram)





AI Capabilities by Problem Type

Perception

Vision, sensors, detection, segmentation, tracking

Language

Classification, translation, summarization, question answering

Reasoning

Planning, search, ranking, decision support, constraint solving

Retrieval

Search, memory systems, recommendations, knowledge grounding

Generation

Text, image, audio, video, code, synthetic data

Action

Robotics, control loops, autonomous behavior, physical systems

One product, many capabilities

A modern assistant can combine retrieval, language generation, reasoning, and user feedback in the same workflow.

That is why "AI product" is broader than "one model."

Before choosing a model or library, ask: **what capability do we actually need?**



Three Core Learning Paradigms

Supervised

Learn from labeled examples.

Example: predict spam vs not spam from emails with known labels.

Unsupervised

Find structure without target labels.

Example: cluster customers by behavior or detect anomalies.

Reinforcement

Learn through actions, rewards, and repeated feedback.

Example: game playing, robotics, sequential decision tasks.

Hype vs. Reality: Generative AI gets the spotlight, but classical ML still powers many core business decisions.

The Advantage: For structured-data problems, simpler models are usually faster, cheaper, and easier to explain and govern.

The Golden Rule: Start with the simplest model that solves the problem, and only scale up when the data or use case truly demands it.



Why This Matters Outside the Classroom

Transport

Dynamic pricing, maintenance prediction, routing, and demand forecasting.

Healthcare

Medical imaging support, triage analytics, and resource planning.

Retail & Finance

Fraud detection, recommendations, customer segmentation, pricing, and risk analysis.

Important reality check

Useful systems need far more than a model: data collection, labeling, evaluation, interfaces, monitoring, and often human review.



CRISP-DM: Still the Skeleton of Many Projects

1. Business

What decision or problem are we trying to improve?

2. Data

What data exists, and how trustworthy is it?

3. Prepare

Clean, join, reshape, validate, and document the dataset.

4. Model

Train or configure the analytical method that fits the task.

5. Evaluate

Check quality against both metrics and business meaning.

6. Deploy

Ship results into reports, apps, services, or monitoring loops.

What 2026 teams add

- Version control for data and code
- Tests, review, and reproducible environments
- Monitoring for quality drift after deployment
- MLOps or analytics ops on top of the core loop

Fun workflow fact

CRISP-DM comes from the late 1990s, but the loop still survives because most real projects still bounce between goals, messy data, evaluation, and deployment.



Checkpoint: AI, ML, and Data Science

Q1

A team groups customers into segments without any target labels. Which learning paradigm fits best?

- A) Supervised learning
- B) Unsupervised learning
- C) Reinforcement learning
- D) Mystical spreadsheet telepathy

Show Answer

Q2

A hospital team cleans records, visualizes trends, and uses a model to forecast bed demand. Which label is most accurate?

- A) Only deep learning
- B) Only data engineering
- C) Data science, possibly including ML
- D) A printer driver with ambition

Show Answer



Modern Python Data Workflow

From workspace to dataframe to first answer



The Stack as a Workflow

Learning objective

See the modern Python data stack as a workflow from question to trustworthy answer: workspace, dataframe tools, explanation, and only then modeling if needed.

By the end of this section, you can

- Explain where JupyterLab, pandas, Polars, plotting, and ML libraries fit
- Describe the fast loop: load, inspect, answer, refactor
- Recognize the dataframe as the main working table
- Use a first filtered question to build confidence in a dataset

Workspace

JupyterLab, notebooks, scripts, terminals

Working table

pandas by default; Polars when lazy/selective pipelines matter

Explanation

summary tables, plots, and short written conclusions

Modeling

scikit-learn, PyTorch, and friends after the data is trustworthy

JupyterLab

pandas

Polars

plots

ML

In many classroom and business tasks, the work ends in a table, chart, or decision note. Modeling is a later step, not the starting point.



Notebook First, Module Next

The fast loop for data work

Load

Open a local CSV or parquet file and define what one row means.

Inspect

Check `head()`, dtypes, and missing values before transforming.

Answer

Filter, group, summarize, and plot to answer one concrete question.

Extract

Move stable logic into modules, tests, and reusable scripts.

Good notebook habits

- Keep one question or step per cell
- Restart and rerun from the top when the notebook stabilizes
- Write short markdown notes about assumptions and findings

What should leave the notebook

- Reusable cleaning functions
- Shared pipeline logic and package code
- Tests, packaging, and deployment details



The DataFrame Is the Working Table

How to think about it

- **Series:** one labeled dimension, often one column or one summary result
- **DataFrame:** multiple aligned series in one table
- **Rows:** records such as missions, observations, or events
- **Columns:** variables such as agency, destination, or cost
- **Dtypes:** part of the meaning, not just an implementation detail

In the ``space_missions`` example, one row is one mission. That mental model makes filtering and grouping much easier to explain.

A tiny dataframe example

Species	Island	Records	Sampled
Adelie	Torgersen	152	True
Chinstrap	Dream	68	True
Gentoo	Biscoe	124	True

A `Series` might be the `records` column alone, or the result of a summary like `value_counts()`.



First Real Table: Read, Inspect, Ask

```
import pandas as pd

missions = pd.read_csv(
    "space_missions.csv",
    dtype_backend="pyarrow",
)

moon_missions = missions.loc[
    missions["destination"] == "Moon",
    ["mission_name", "launch_year", "agency", "cost_million_usd"],
].sort_values("launch_year")

agency_counts = missions["agency"].value_counts()
```

Read a real table, inspect it, and ask one focused question before doing heavier transformations.

Patterns to recognize quickly

- `head()`, `dtypes`, and null checks after loading
- `df.iloc[0]` for a first positional row
- `df.loc[mask, cols]` for clear filtering
- `value_counts()` for a first categorical summary

Question-first analysis

"Which Moon missions are in this dataset?" and "Which agency appears most often?" are good first questions because they build confidence in both the table and the access patterns.



Checkpoint: Modern Python Data Workflow

Q3

After loading a new CSV into a dataframe, what is the best next move?

- A) Inspect the first rows, dtypes, and missing values
- B) Start model training immediately
- C) Ignore the table and only rename files
- D) Choose chart colors before reading the data

Show Answer

Q4

In the `space_missions` dataframe, one row most naturally represents what?

- A) One mission record
- B) One Python package
- C) One notebook tab
- D) One plotting theme

Show Answer



pandas in Practice

From raw tables to filtered, cleaned, and grouped answers

What pandas Is Good At

Learning objective

Use pandas as the standard tool for tabular data work: load a dataset, inspect it, transform it, and answer concrete questions with readable code.

By the end of this section, you can

- Read a CSV into a dataframe and inspect its structure
- Filter rows and columns to answer a focused question
- Create new columns and lookup joins from clear, auditable rules
- Clean text-heavy or partially missing columns explicitly
- Summarize grouped or time-based patterns into decision-ready tables

Typical problems pandas solves

- **Exploration:** inspect columns, dtypes, and missing values
- **Selection:** keep only the rows and columns relevant to the question
- **Feature engineering:** derive labels, bands, dates, or rates from raw columns
- **Integration:** attach metadata from another table with a validated join
- **Cleaning:** turn messy strings into useful typed columns
- **Reporting:** build grouped summaries that someone can actually discuss

CSV filtering cleaning groupby joins

Pandas shines when the data is tabular, labeled, and small-to-medium enough to fit in memory while you iterate.



Scenario 1: Inspect the Table Before Touching It

```
import pandas as pd

missions = pd.read_csv(
    "space_missions.csv",
    dtype_backend="pyarrow",
)

missions.head(5)
missions.dtypes
missions.isna().sum()
```

The first task is not clever analysis. It is understanding the table: what one row means, which columns are numeric or text, and where missing values may matter later.

What to inspect immediately

- What does one row represent?
- Which columns are text, numeric, or boolean?
- How many missing values does each column have?
- Which columns look like identifiers, categories, or measurements?

Good analysis starts with structure, not with a clever one-liner. In these demos, `dtype_backend="pyarrow"` keeps nullable columns easy to reason about.



Scenario 2: Filter to the Relevant Rows

```
moon_missions = missions.loc[
    missions["destination"] == "Moon",
    [
        "mission_name",
        "launch_year",
        "agency",
        "cost_million_usd",
    ],
].sort_values("launch_year")
```

Pick Moon rows, keep four useful columns, then sort by year for a clean answer table.

Pattern: `.loc[rows, columns]` keeps both filters explicit.

Moon missions in the current dataset

Mission	Year	Agency	\$M
Apollo 8	1968	NASA	2500
Apollo 11	1969	NASA	25400
Apollo 13	1970	NASA	25400
Chang'e 4	2018	CNSA	170
Chandrayaan-2	2019	ISRO	140
Chang'e 5	2020	CNSA	200
Artemis I	2022	NASA	4100
Chandrayaan-3	2023	ISRO	75
Artemis II	2026	NASA	4500

All 9 Moon rows in `space_missions.csv`, sorted by `launch_year`.



Scenario 3: Derive Meaningful Columns Safely

```
missions["mission_style"] = "uncrewed"
missions.loc[missions["crew_size"] > 0, "mission_style"] = "crewed"

missions["distance_band"] = "local"
missions.loc[missions["distance_km"] ≥ 384_400, "distance_band"] = "lunar_or_beyond"
missions.loc[missions["distance_km"] ≥ 1_000_000, "distance_band"] = "deep_space"
```

A typical dataframe task is to turn raw numeric columns into labels that are easier to group, plot, and explain.

What this gives us

- Apollo 11 -> crewed, lunar_or_beyond
- Voyager 1 -> uncrewed, deep_space
- ISS Expedition 1 -> crewed, local
- James Webb -> uncrewed, deep_space

This is feature engineering in dataframe form. Use explicit column assignment and `.loc` updates when the rule depends on a condition.



Scenario 4: Parse a Messy Text Column

```
wkt_pattern = (  
    r"POINT Z \((?P<longitude>.*?) "  
    r"(?P<latitude>.*?) "  
    r"(?P<depth>.*?)\)"  
)  
  
coords = (  
    quakes["geometry"]  
    .str.extract(wkt_pattern)  
    .astype("float32[pyarrow]")  
)
```

The key idea is not the regex itself. The key idea is that the same parsing rule is applied to the whole column, producing new numeric columns in one step.

From one cell to three columns

Raw geometry cell	Extracted values
POINT Z (-169.6766 52.3338 10)	longitude=-169.6766 latitude=52.3338 depth=10

Prefer vectorized string methods over Python row loops when the same cleaning rule applies to every row.



Scenario 5: Build a Summary Table

```
destination_summary = (
    missions[missions["destination"].isin(["Moon", "Mars", "ISS"])]
    .assign(cost_per_day=lambda df: (df["cost_million_usd"] / df["duration_days"]).round(2))
    .groupby("destination")
    .agg(
        mission_count=("mission_name", "size"),
        avg_cost_per_day=("cost_per_day", "mean"),
        crewed_missions=("mission_style", lambda s: (s == "crewed").sum()),
    )
    .sort_values("avg_cost_per_day", ascending=False)
)
```

This pipeline filters, derives, groups, and sorts to answer a reporting question.

What this summary answers

Which destinations are most expensive per day, and how many missions are crewed?

- **Moon:** count 9, avg/day 937.90, crewed 4
- **ISS:** count 1, avg/day 23.44, crewed 1
- **Mars:** count 9, avg/day 0.83, crewed 0

Why `groupby().agg()` matters:

it compresses many detailed rows into an answer table you can explain in one sentence.



Scenario 6: Attach Lookup Data with `merge()`

```
agency_info = pd.DataFrame(  
    {  
        "agency": ["NASA", "CNSA", "ISRO"],  
        "region": ["USA", "China", "India"],  
        "crew_program": ["yes", "planned", "planned"],  
    }  
)  
  
moon_with_agency = moon_missions.merge(  
    agency_info,  
    on="agency",  
    how="left",  
    validate="many_to_one",  
)
```

Real analysis often starts by attaching metadata from a small lookup table. Use `validate=` to catch broken join assumptions early.

Question

How do we add agency metadata without handwritten `if/else` rules?

Mission	Agency	Region	Crew program
Apollo 11	NASA	USA	yes
Chang'e 4	CNSA	China	planned
Chandrayaan-2	ISRO	India	planned
Artemis I	NASA	USA	yes
Artemis II	NASA	USA	yes

Best practice: if the right-hand table should have one row per key, say so with `validate="many_to_one"`.



Scenario 7: Handle Missing Values Explicitly

```
quake_alerts = (  
    quakes.loc[:, ["place", "mag", "mmi", "alert"]]  
    .assign(alert=lambda df: df["alert"].fillna("unknown"))  
    .dropna(subset=["mmi"])  
    .sort_values(["mmi", "mag"], ascending=[False, False])  
    .head(5)  
)
```

Not every missing value deserves the same fix. Fill a category only when an honest fallback exists; drop rows when the missing measurement would break the comparison.

Column-specific rule

Column	Missing	Action
alert	5 rows	fillna("unknown")
mmi	1 row	dropna(subset=["mmi"])

Best practice: do not blindly fill numeric missing values with `0`. Pick a domain-safe rule per column.



Scenario 8: Parse Time Columns Before Grouping

```
quakes["time_utc"] = pd.to_datetime(
    quakes["time"], unit="ms", utc=True
)

quakes_by_year = (
    quakes.assign(year=quakes["time_utc"].dt.year)
    .groupby("year")
    .agg(event_count=("id", "size"), max_mag=("mag", "max"))
    .sort_index()
    .tail(6)
)
```

Parse timestamps once, then use the `.dt` accessor for year/month/day logic. Avoid slicing raw integers or timestamp strings by hand.

What this summary answers

How many earthquakes are in the file each year, and what was the largest magnitude?

Year	Events	Max mag
2020	121	7.8
2021	157	8.2
2022	127	7.6
2023	147	7.8
2024	99	7.5
2025	100	8.8

Best practice: use `utc=True` for machine timestamps so time zones stay explicit from the start.



Checkpoint: pandas workflows

Q5

You want only Moon missions, selected columns, ordered by year. Best pattern?

A) `missions.loc[mask, cols].sort_values("launch_year")`

B) Turn the dataframe into a list first

C) Loop through rows manually

D) Randomly reorder the dataframe

Show Answer

Q6

Need mission counts and average duration per agency. Best operation?

A) `groupby("agency").agg( ... )`

B) `iloc[0]`

C) `str.extract( ... )`

D) `rename( ... )`

Show Answer



When Polars Is Worth It

Keep pandas for routine notebook work; switch when optimized pipelines matter



When pandas Is Enough, When Polars Is Worth It

Learning objective

Choose between pandas and Polars based on workload: use pandas for routine notebook work, and reach for Polars when query planning and execution strategy start to matter.

By the end of this section, you can

- Explain why pandas remains the default first choice
- Spot workloads where lazy execution makes Polars attractive
- Read a Polars query that filters, groups, and ranks
- Move results back into pandas when needed

pandas is enough when

- The data fits comfortably in memory
- You want quick notebook exploration
- The next step is plotting or scikit-learn
- The team already reads pandas fluently

Polars is worth it when

- You read large or wide CSVs but use only part of them
- You run heavy filter/select/group pipelines on one machine
- You want lazy planning before execution
- You need streaming or window-heavy transforms



Scenario 1: Large CSV, Small Answer Table

```
quake_query = (  
    pl.scan_csv("earthquakes.csv")  
    .select("net", "mag", "tsunami")  
    .filter(pl.col("mag").cast(pl.Float64) ≥ 6.0)  
    .group_by("net")  
    .agg(  
        pl.len().alias("quake_count"),  
        pl.col("mag").cast(pl.Float64).mean().alias("avg_magnitude"),  
        pl.col("tsunami").cast(pl.Int64).sum().alias("tsunami_events"),  
    )  
    .sort("quake_count", descending=True)  
)
```

This is the kind of workload where Polars starts to pay off: a wide CSV, only a few useful columns, and a grouped summary at the end.

What the summary tells us

- **us:** 1324 quakes, avg mag 6.38, tsunami 415
- **ak:** 8 quakes, avg mag 6.75, tsunami 8
- **nc:** 4 quakes, avg mag 6.40, tsunami 3

On a small classroom file, pandas would also work. Polars becomes worth trying when the same filter/select/group pattern is aimed at larger files and you want `scan_csv( ... )` to stay lazy until `collect()`.



Scenario 2: Rank Within Each Agency

```
missions = pl.read_csv("space_missions.csv")

ranked_missions = (
    missions.select("agency", "mission_name", "cost_million_usd")
    .with_columns(
        pl.col("cost_million_usd")
        .rank("dense", descending=True)
        .over("agency")
        .alias("cost_rank_within_agency")
    )
    .filter(pl.col("cost_rank_within_agency") ≤ 2)
    .sort(["agency", "cost_rank_within_agency"])
)
```

This example from the local mission dataset asks for a top-2 ranking within each agency, which is a natural place to introduce Polars expressions and window logic.

What this answers

- **NASA:** Apollo 11 rank 1, Artemis I rank 2
- **NASA/ESA:** James Webb rank 1, Hubble rank 2
- Smaller agencies keep only their most expensive mission

`over("agency")` is a window expression: rank inside each group without manually splitting the dataframe. If you already know pandas and only need one quick notebook answer, pandas is still fine. Polars starts to feel worth it when many expression-based per-group transforms stack up.



Scenario 3: Optimize in Polars, Deliver in pandas

```
ranked_missions_pd = ranked_missions.to_pandas()  
# hand off to plotting / scikit-learn / older code  
  
round_tripped = pl.from_pandas(ranked_missions_pd)
```

You do not have to choose one library forever. A practical workflow can do the heavy dataframe step in Polars and still hand the result to the pandas ecosystem.

Scenario	Better first pick
Notebook exploration, plotting, scikit-learn handoff	pandas
Large CSV with select/filter/group pipeline	Polars
Expression-heavy ranking or window logic	Polars
Heavy transform first, pandas ecosystem next	Polars -> pandas

Start with pandas by default. Add Polars when the workload justifies lazy planning, selective CSV scans, or heavier expression pipelines.



Checkpoint: pandas vs Polars

Q7

Your data fits in memory, you are exploring in a notebook, and the next step is plotting plus scikit-learn. Best first choice?

- A) pandas
- B) Polars, no discussion needed
- C) Rewrite everything in assembly
- D) A blockchain-powered CSV

Show Answer

Q8

You have a wide CSV, but the answer needs only a few columns after filtering before a grouped summary. Which Polars pattern is worth trying?

- A) `pl.scan_csv( ... ).select( ... ).filter( ... ).group_by( ... ).agg( ... ).collect()`
- B) Read everything eagerly and loop row by row in Python
- C) Convert the file to screenshots first
- D) Use `iloc[0]` and hope the first row explains the whole dataset

Show Answer



Lecture Summary

Workflow

- Start in a notebook or script with one concrete question
- Load, inspect, answer, then extract stable logic
- Modeling comes after the data is trustworthy

Working Table

- Treat the dataframe as the main working table
- Read, inspect, filter, derive, parse, summarize
- Use `.loc`, vectorized code, and clear pipelines

Tool Choice

- JupyterLab for fast exploration and teaching
- pandas by default for routine notebook workflows
- Polars when lazy scans or heavier pipelines justify it

The real skill is not memorizing library names. It is following a clean workflow from question to table to answer, then choosing the lightest tool that fits the workload.



Thank You for Your Attention

“Without data you're just another person with an opinion.” — W.
Edwards Deming

AI

Data Science

pandas

Polars



Python Course 2025/26/2

Lecture 10: Data Visualization with Matplotlib, Seaborn, and Plotly



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Agenda

- Choose charts by question, not by habit
- Use Matplotlib with the explicit object-oriented API
- Think in mappings, marks, stats, scales, and facets
- Use Plotly when interactivity truly adds value
- Prepare data for maps and large visual workloads sensibly

Learning objective

Move from "I know how to call a plotting function" to a better engineering habit: pick the right chart, the right library layer, and the right amount of data for the reader or the browser.

By the end, you can

Explain which visual encoding matches a question, write readable Matplotlib and seaborn code, and decide when Plotly maps or aggregation-heavy workflows are worth the extra machinery.

Decision-making

Static vs interactive

Accessibility

Geospatial

Performance

Lecture 10 is about visual thinking with the modern Python data stack, not just memorizing plotting syntax.



Why Visuals Matter

Start with the question, then choose the visual encoding

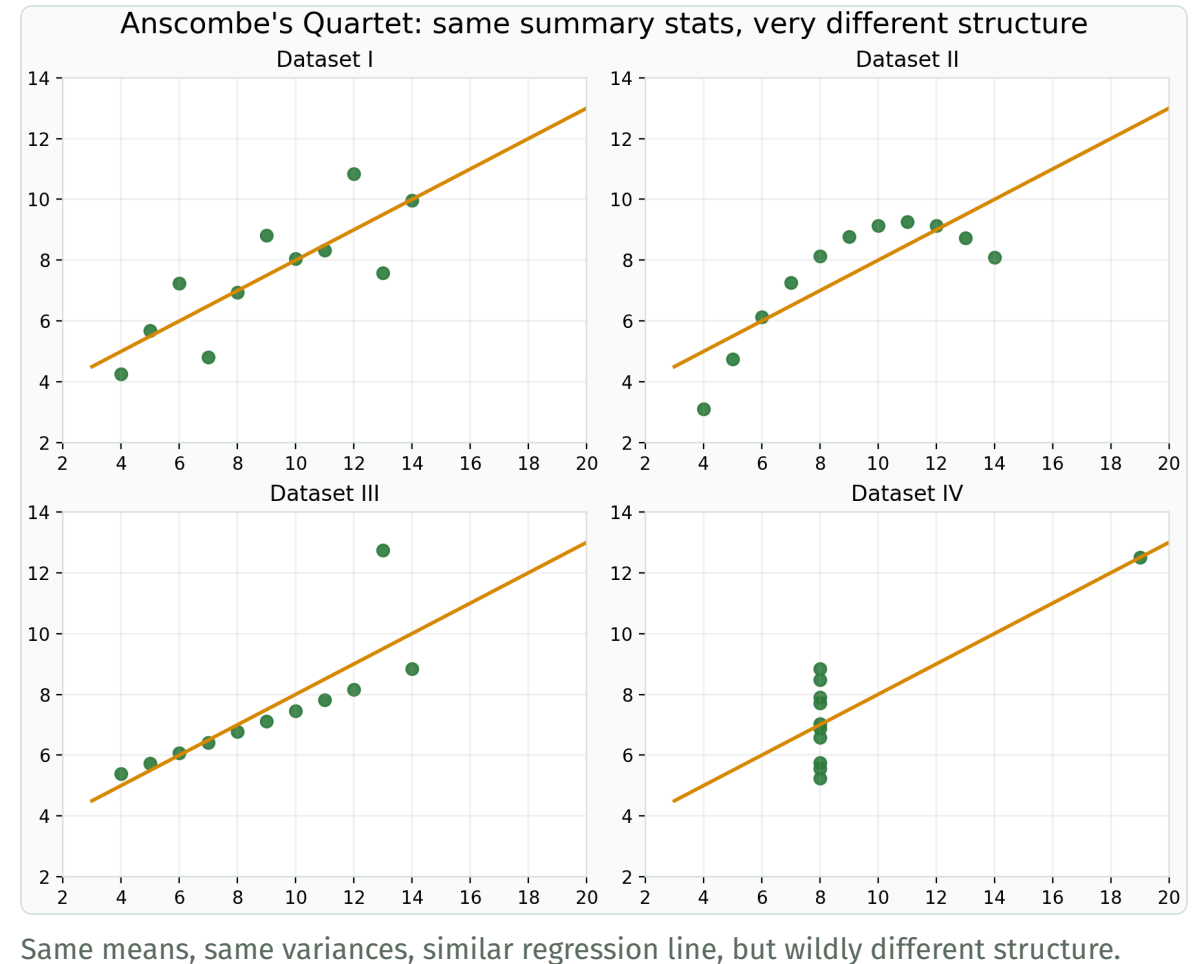
Why Visualize? Patterns Beat Raw Tables

The cognitive bandwidth problem

- Tables are read row by row and column by column: that is slow and serial.
- Plots let you see trend, spread, outliers, and clusters in parallel.
- Good charts compress many numbers into a readable shape without hiding the logic.

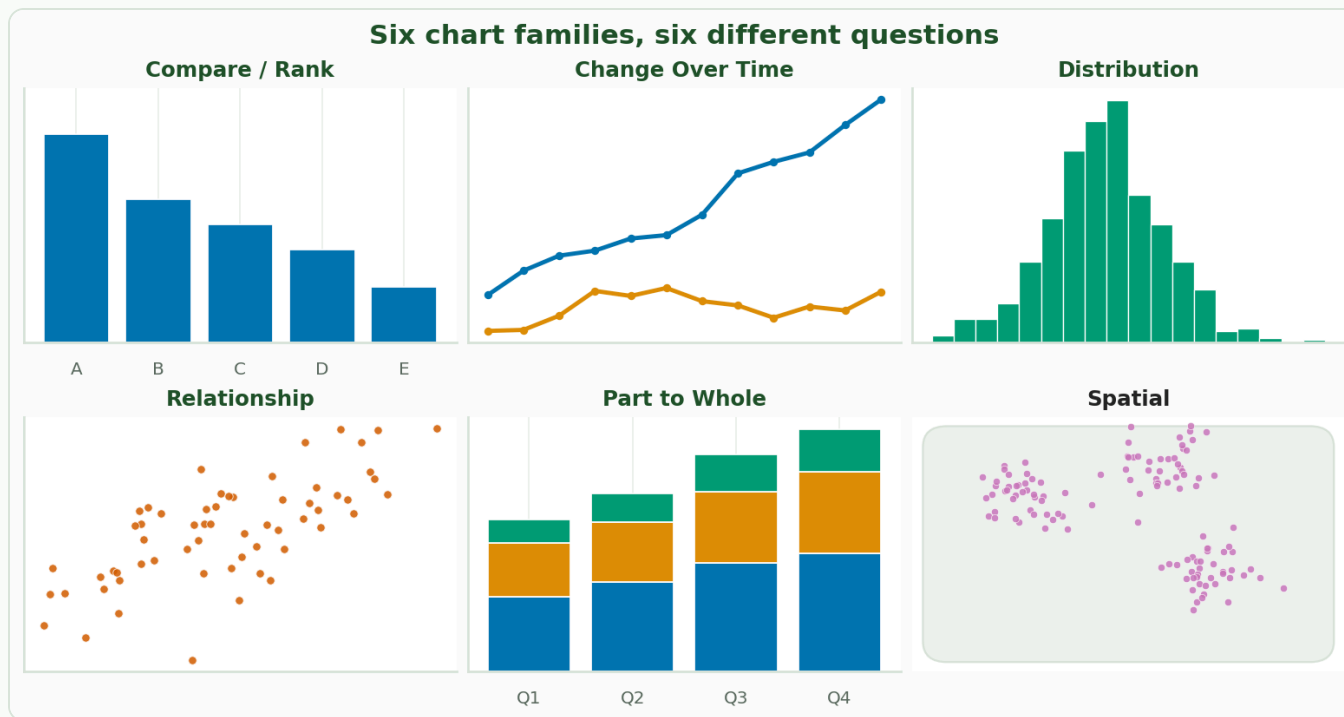
Fun fact 1

Anscombe's quartet is the classic reminder that four datasets can share almost identical summary statistics and still tell four completely different visual stories once plotted.





Choose by Question, Not by Chart Habit



Six chart families, six different questions.

Compare / Rank

Bar, dot, lollipop — categories are the story.

Change Over Time

Line or small multiples; bars only for a few periods.

Distribution

Histogram, density, box, violin, ECDF — show spread.

Relationship

Scatter, hexbin, 2D density when points overlap.

Part to Whole

Stacked bars or treemaps first; pies only for tiny n.

Spatial

Symbol map, choropleth, density map — quiet basemap.

Fun fact 2

William Playfair introduced line and bar charts in the late 18th century, and the pie chart followed in 1801. Some of our most useful chart ideas are older than computers.



Workhorse Charts Beat Fancy Charts

Default to these

- `bar` / dot / lollipop for categories
- `line` for trends over time
- `histogram` / box / violin for distributions
- `scatter` for relationships
- Small multiples when one chart gets too crowded

Use with caution or redesign

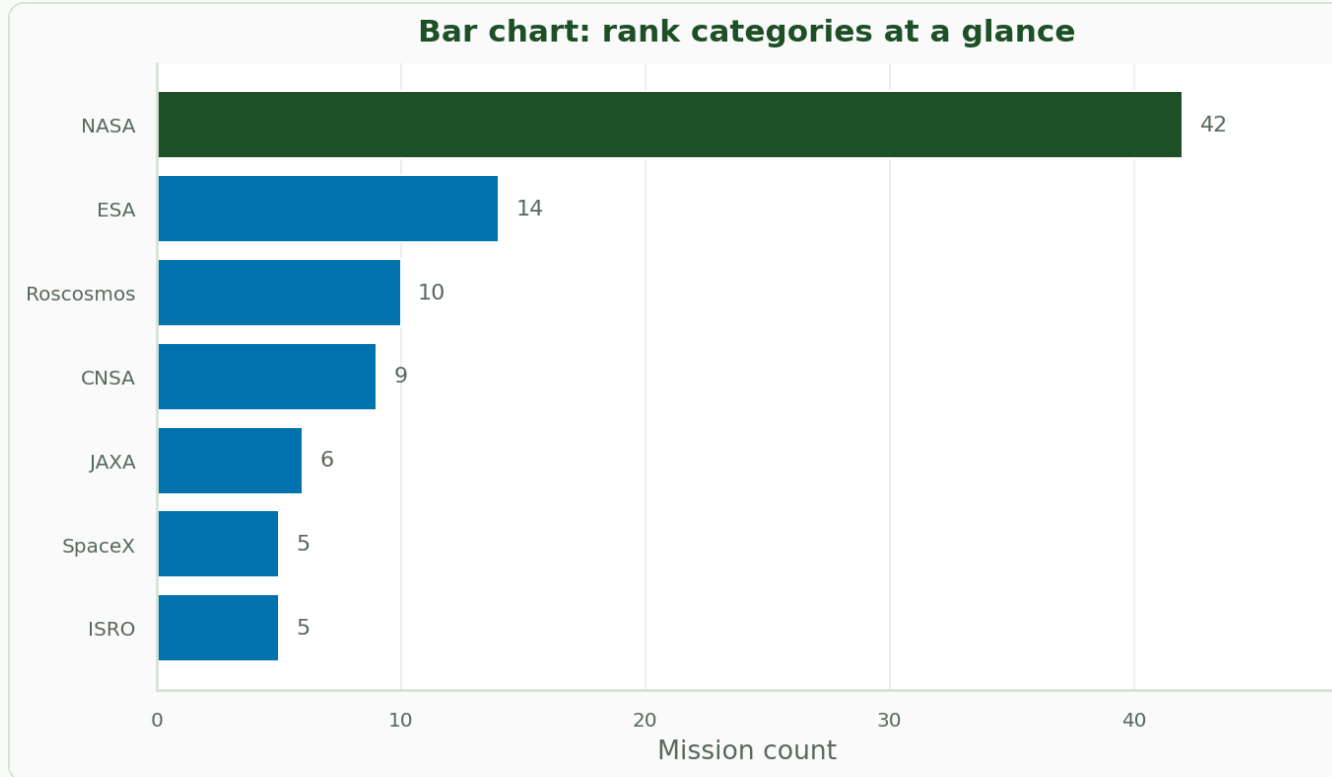
- 3D charts: decoration usually hurts reading
- Bubble charts: size is a weak perceptual channel
- Dual axes: easy to fake a relationship by scaling the axes
- Radar charts: shape changes when axis order changes
- Large pies and donuts: angles are harder to compare than lengths

Fun fact 3

Charles Minard's 1869 chart of Napoleon's Russian campaign is still widely cited as one of the greatest statistical graphics ever made because it encodes geography, direction, army size, and temperature in one coherent story.



Workhorse 1 — Bar Chart for Comparing Categories



When to reach for it

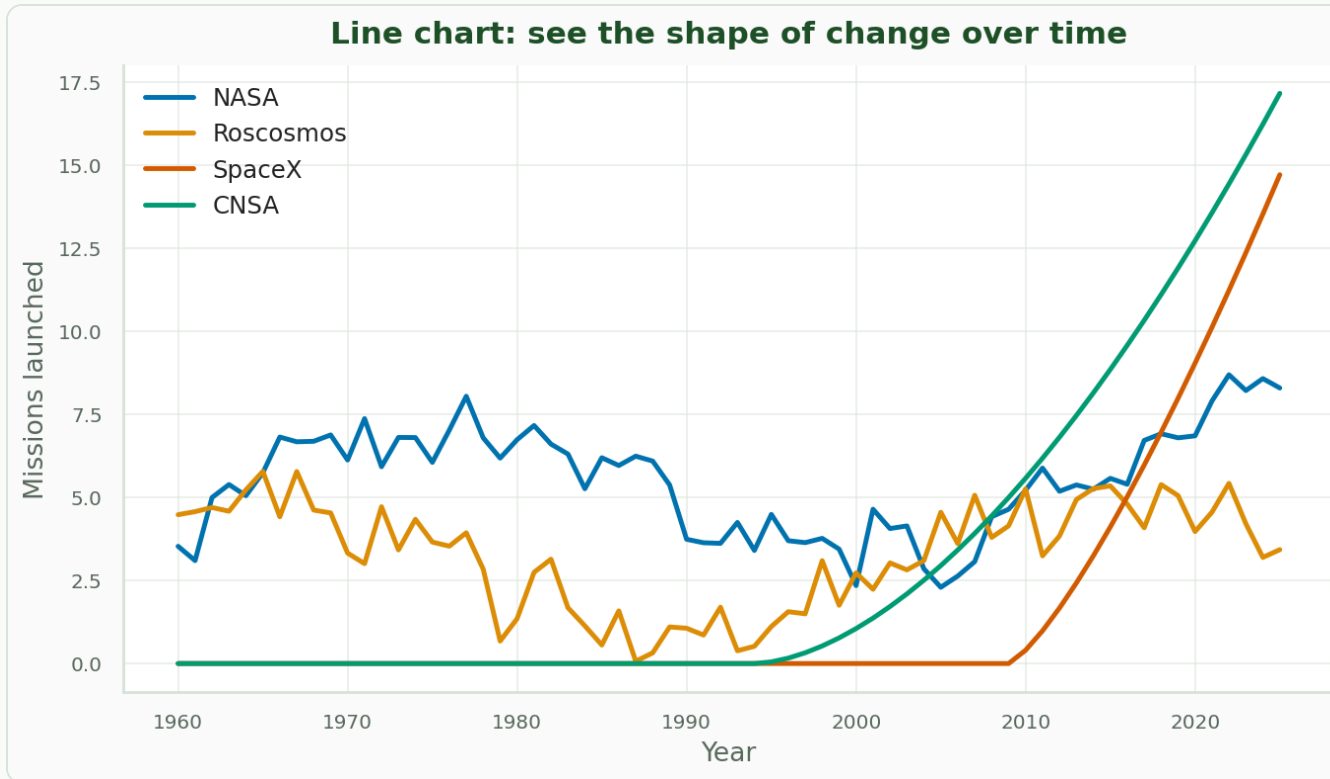
- Ranking categorical totals or rates.
- Comparing many items with long, descriptive labels.
- Putting the answer to "who is biggest?" in front of the reader.

Habits that pay off

Sort by value, not by alphabet. Start the quantitative axis at zero. Use a horizontal layout when labels would otherwise rotate.



Workhorse 2 — Line Chart for Change Over Time



When to reach for it

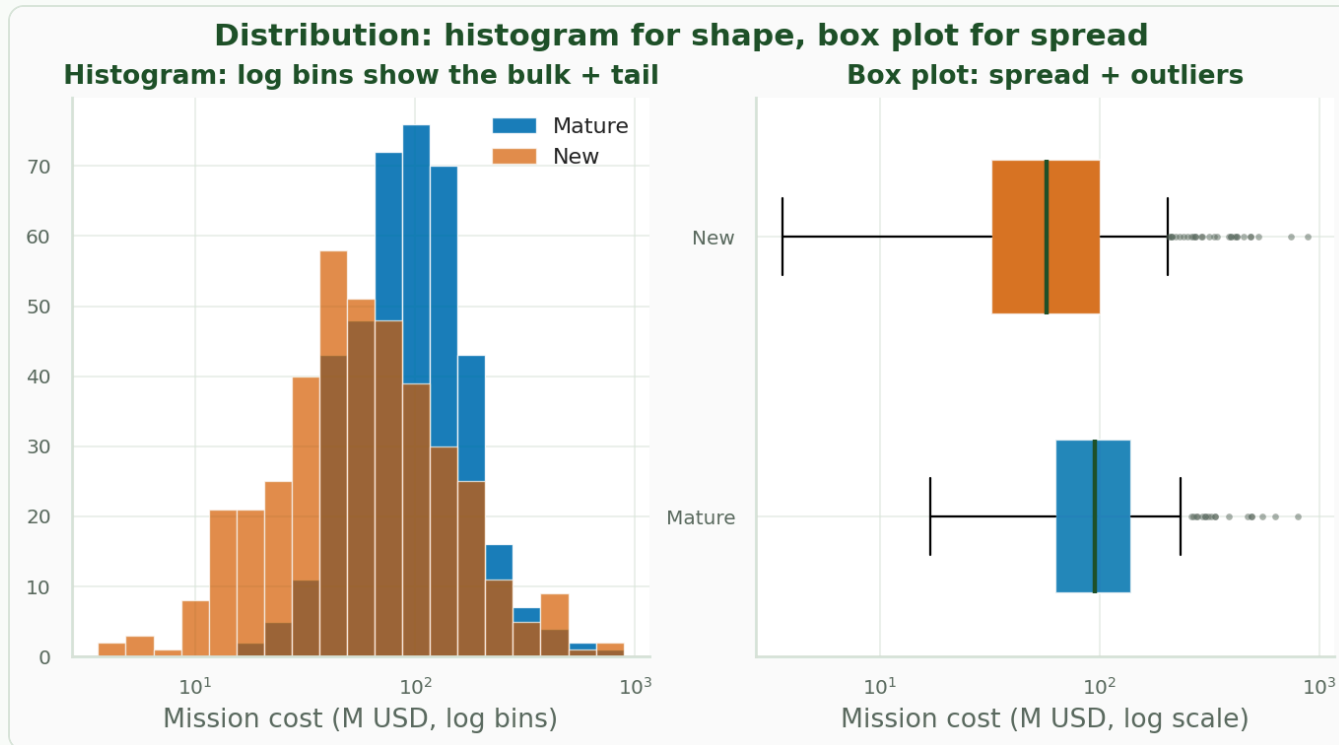
- Continuous time (years, weeks, minutes) on the x-axis.
- Multiple series with one shared scale.
- You want the eye to see direction and inflection.

Habits that pay off

Anchor the time axis to real years. Label the lines directly when there are only a few. Avoid dual axes — if scales differ, split into facets instead.



Workhorse 3 – Histogram and Box for Distributions



When to reach for it

- Asking "how is one variable spread out?"
- Comparing the shape and tail of two or three groups.
- Surfacing skew, multimodality, and outliers explicitly.

How to read the box plot

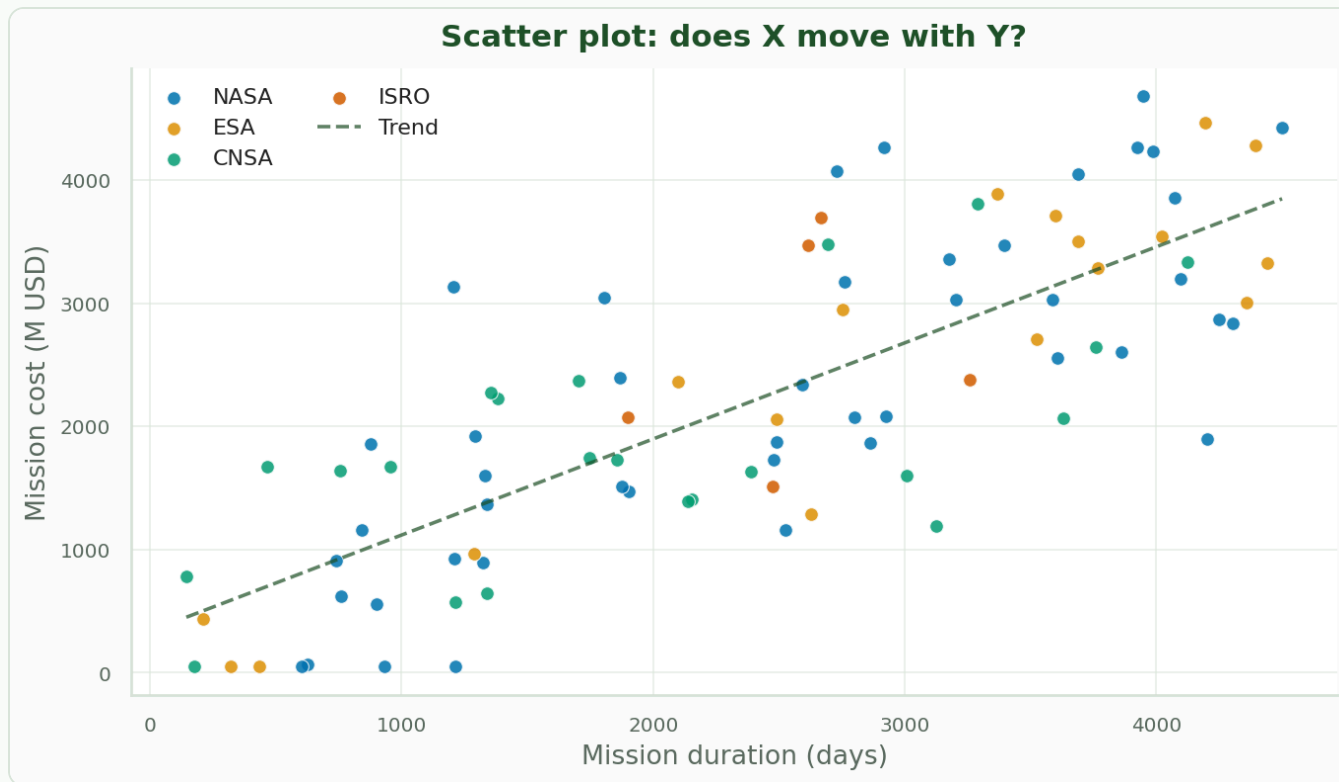
- **Median:** the thick line inside the box is the 50th percentile.
- **IQR box:** spans the middle 50% from the 25th to the 75th percentile.
- **Whiskers:** extend to the most extreme points still considered "inside the bulk" (Matplotlib defaults to 1.5×IQR past each quartile).
- **Outliers:** individual points beyond the whiskers are plotted explicitly so the tail is not hidden.
- **Log axis note:** here the x-axis is log-scaled, so distances on the page are multiplicative in dollars, not additive.

Habits that pay off

Histogram for shape, box (or violin) for spread + outliers.
Pair them when you want both stories on one slide.



Workhorse 4 — Scatter Plot for Relationships



When to reach for it

- Two continuous variables and the question "do they move together?"
- Color or shape add a third categorical variable cheaply.
- Trend lines or smoothers explain the pattern, not just the dots.

Habits that pay off

Use alpha or aggregation when points overpaint. Keep the trend line subtle. Annotate the few points that actually matter to the story.



Five-Second Sanity Checks

Ask before you export

- Can a reader answer the intended question in under five seconds?
- Does the title say the point, not only the topic?
- Are the most important values labeled or visually obvious?
- Is any decorative choice competing with the data?

Accessibility defaults

- Use restrained palettes and direct labels when possible.
- Do not encode meaning only by color.
- For bars, start the quantitative axis at zero.
- Use low-ink gridlines and readable text, not visual noise.
- `sns.color_palette("colorblind")` is a good safe default for categories.

Safe charts feel boring only until you compare them with a misleading one. The job is not to impress the chart gallery; the job is to make the answer obvious.

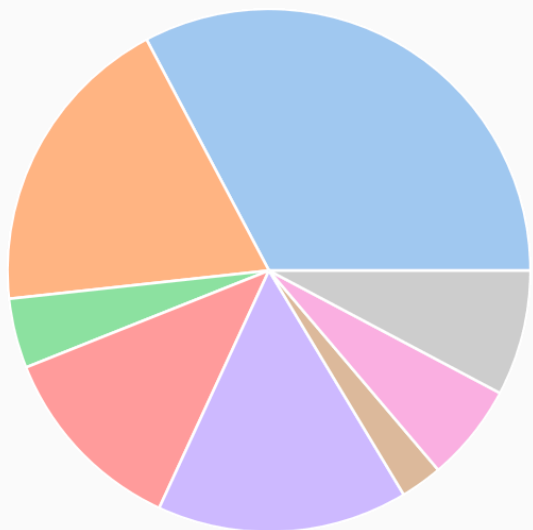


Q1 — Can a reader answer in under five seconds?

Sanity check #1 — Which feature drives the most engagement?

BAD

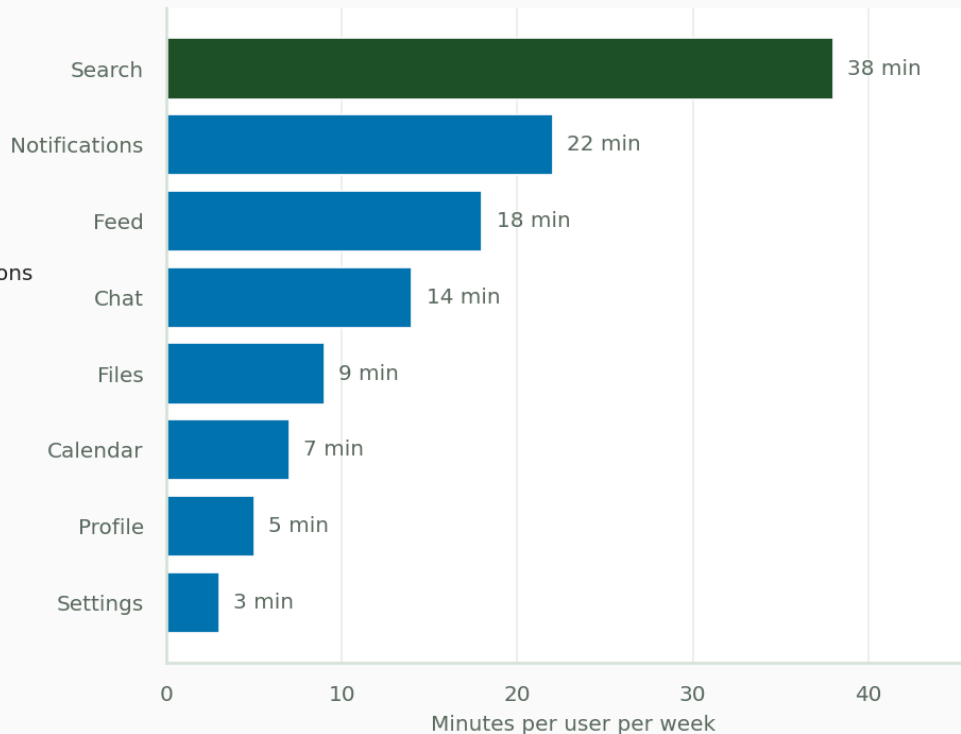
Engagement by feature



Feature
Search
Notifications
Profile
Chat
Feed
Settings
Calendar
Files

GOOD

Search leads engagement by a wide margin

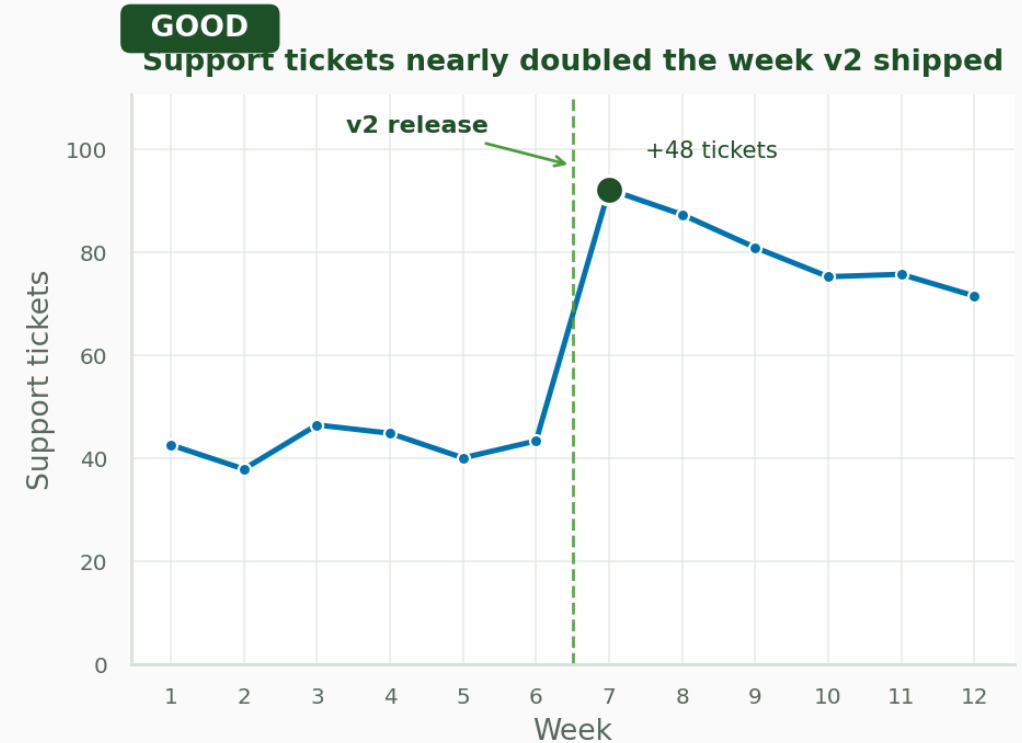
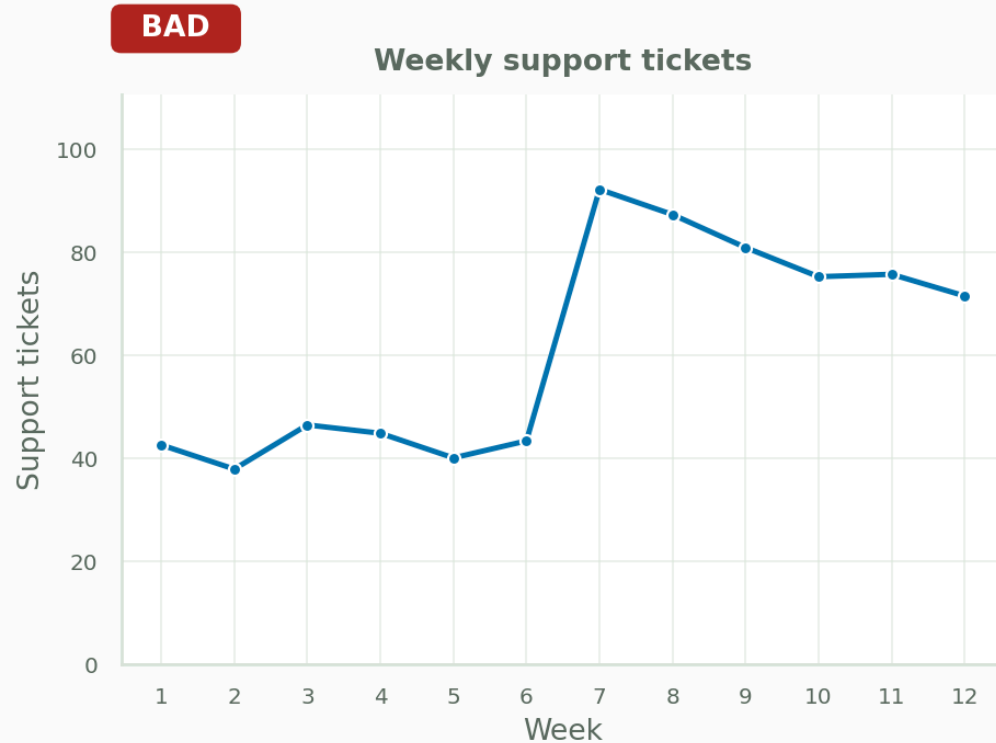


Takeaway. A pie with many similar slices and a side legend forces the reader to match colors before ranking. A sorted horizontal bar with the winner highlighted and direct labels answers "which feature wins?" in a single glance.

Q2 — Does the title say the point, not only the topic?



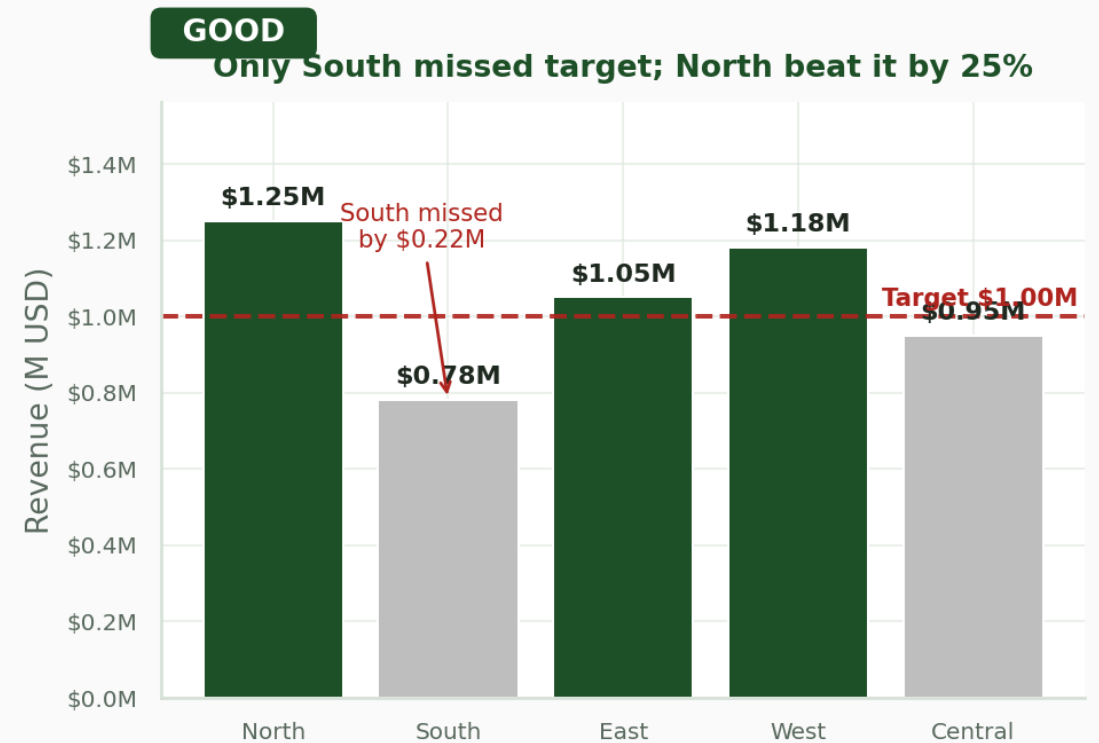
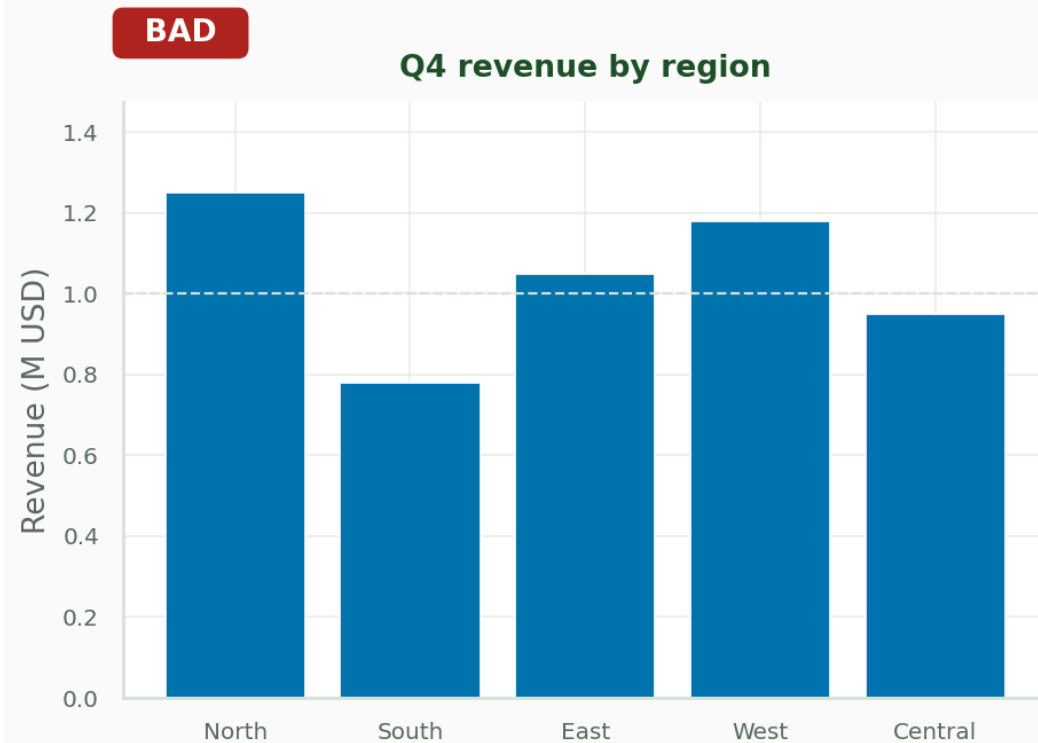
Sanity check #2 — Same chart, two different titles



Takeaway. Identical data; only the title changes. A topic title (Weekly support tickets) makes the reader discover the story. A point title plus a light annotation hands the story over for free.

Q3 — Are the most important values labeled or visually obvious?

Sanity check #3 — Q4 revenue by region vs 1.0M target



Takeaway. If the key number is not labeled, colored, or annotated, it is effectively hidden. Direct labels plus one accent color for the comparison of interest make the critical value impossible to miss.

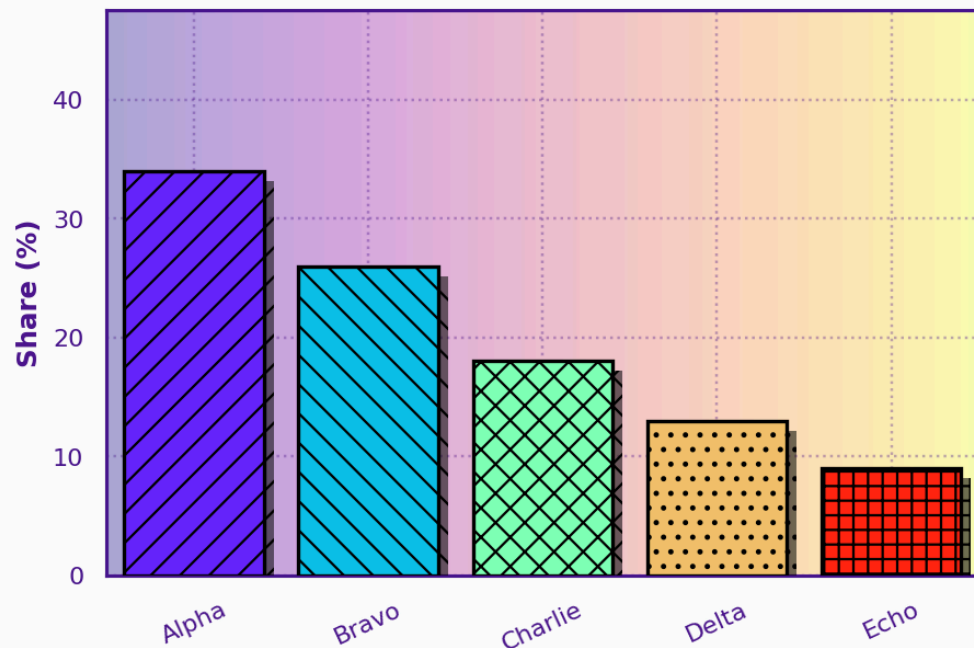
Q4 — Is any decorative choice competing with the data?



Sanity check #4 — Same numbers, different amount of ornament

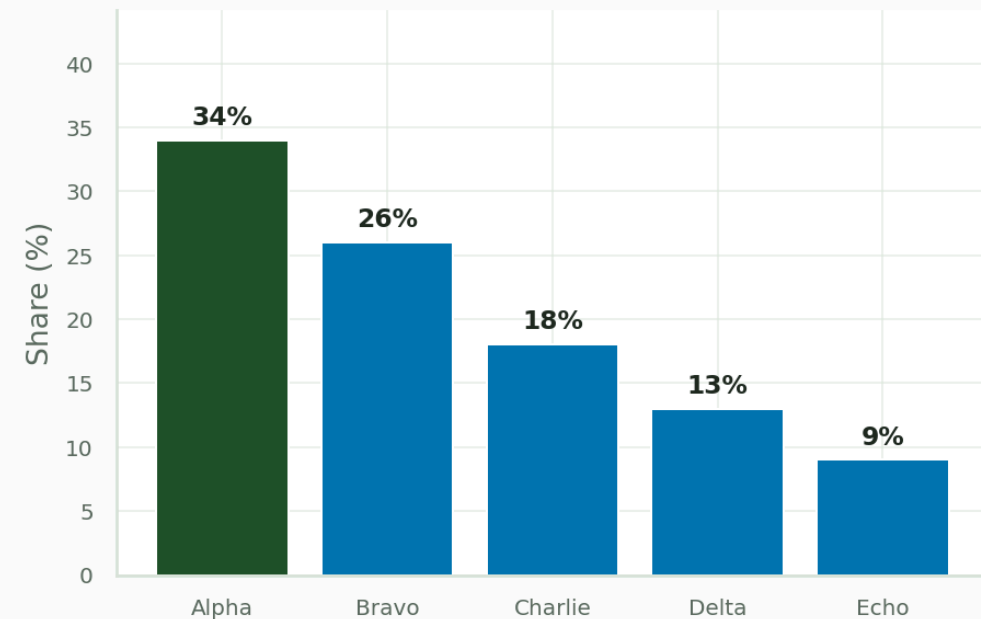
BAD

★ *Market Share by Product* ★



GOOD

Alpha leads with 34% share



Takeaway. Rainbow palettes, hatch patterns, gradient backgrounds, and heavy shadows all fight the numbers for attention. One accent color on the value that matters carries the same data with a fraction of the ink.



Checkpoint: Chart Choice

Q1

You want to show how one store's weekly revenue changes across 52 weeks. Which chart is the safest default?

- A) A line chart
- B) A pie chart with 52 slices
- C) A radar chart
- D) A contour map, for dramatic effect

Show Answer

Q2

Which chart is the best default for comparing 12 categories with long labels?

- A) Horizontal bar or dot chart
- B) Exploded 3D pie chart
- C) Dual-axis line chart
- D) Whichever chart won an argument on social media

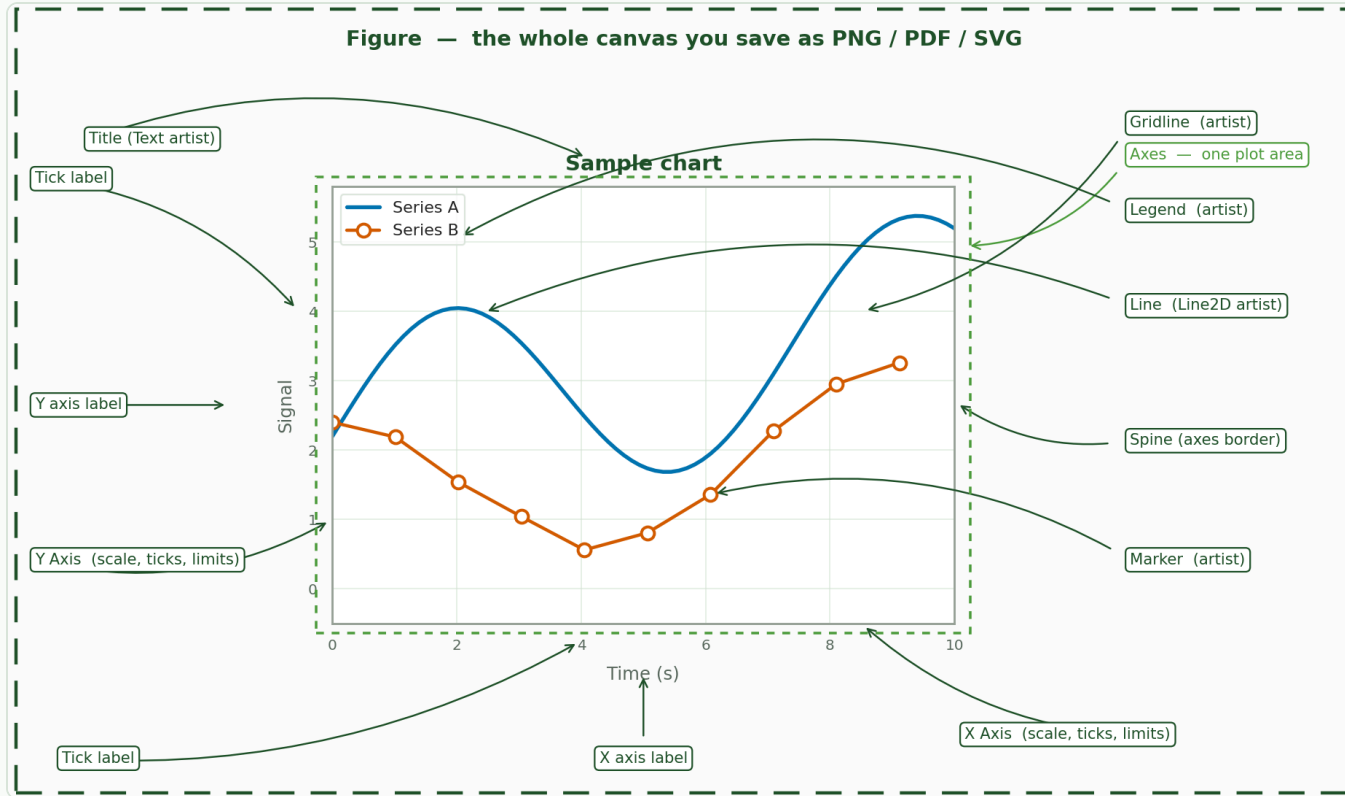
Show Answer



Matplotlib Foundations

Know the engine that higher-level libraries stand on

Figure, Axes, Axis: Know the Hierarchy



Same chart, annotated with the actual Matplotlib vocabulary.

Mental model

- **Figure**: whole canvas or exported page
- **Axes**: one plotting region
- **Axis**: scale, limits, ticks, formatting
- Everything visible is an artist attached to an axes

Even when pandas or seaborn make the chart, Matplotlib still draws the pixels.
Understanding the hierarchy is what makes debugging possible.



Prefer the Explicit Object-Oriented API

State-machine style

```
import matplotlib.pyplot as plt

plt.plot(years, revenue)
plt.title("Revenue")
plt.xlabel("Year")
plt.ylabel("USD")
```

Fine for quick sketches, but global state becomes awkward in larger figures.

OO style

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(layout="constrained")
ax.plot(years, revenue)
ax.set(title="Revenue", xlabel="Year", ylabel="USD")
```

You know exactly which axes you are modifying, which matters for subplots and reusable code.

Default lecture habit: create `fig, ax` first, then configure `ax`. It scales better than hoping Matplotlib guesses the target plot.

Semantic Layouts with `subplot_mosaic()`



Readable layout code

```
fig = plt.figure(layout="constrained")
axd = fig.subplot_mosaic([
    ["summary", "summary"],
    ["trend", "dist"],
])

axd["summary"].bar( ... )
axd["trend"].plot( ... )
axd["dist"].hist( ... )
```

Why it helps

- Named panels are easier to reason about than `axs[1, 0]`.
- Repeated names make one panel span multiple cells.
- `layout="constrained"` lets Matplotlib solve spacing for titles, legends, and labels.

Small dashboards become much less fragile when the layout is named semantically instead of managed by hard-to-remember indices.

Static Charts Are for Explanation

When Matplotlib / seaborn shine

- Papers, reports, and slides where the message is fixed
- Precise annotation and publication-quality control
- Custom layouts, small multiples, and exportable vector graphics

Export format guide

Format	Use it for
<code>.svg</code> / <code>.pdf</code>	Print, papers, infinite zoom, post-processing
<code>.png</code>	Slides, LMS uploads, screenshots
<code>.jpg</code>	Usually avoid for charts; compression adds artifacts

```
fig.savefig("mission_costs.svg", dpi=300, bbox_inches="tight")
```

SVG/PDF for publication, PNG for classroom slides, and almost never JPEG for charts.



Checkpoint: Matplotlib Foundations

Q3

Which snippet best follows the explicit Matplotlib API for a reusable chart?

A) `plt.plot( ... ); plt.title( ... )`

B) `fig, ax = plt.subplots( ... ); ax.plot( ... )`

C) A handwritten screenshot of a chart

D) Whatever worked once at 2 a.m.

Show Answer

Q4

Why is `subplot_mosaic()` attractive for complex figures?

A) It returns named axes, which makes the layout easier to read and extend

B) It makes legends illegal

C) It converts plots into 3D charts automatically

D) It only works on Tuesdays

Show Answer



Seaborn Grammar and Data Prep

Move from plotting calls to clearer plotting semantics

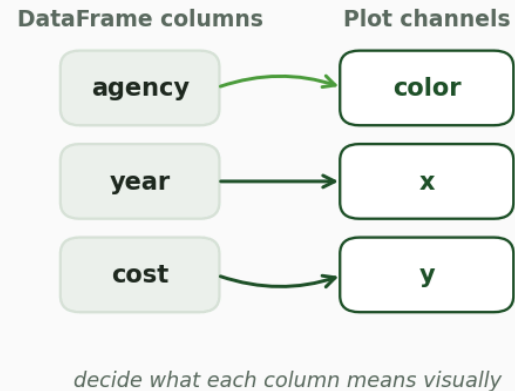
Static Stack: DataFrame -> Meaning -> Figure

1. Tidy

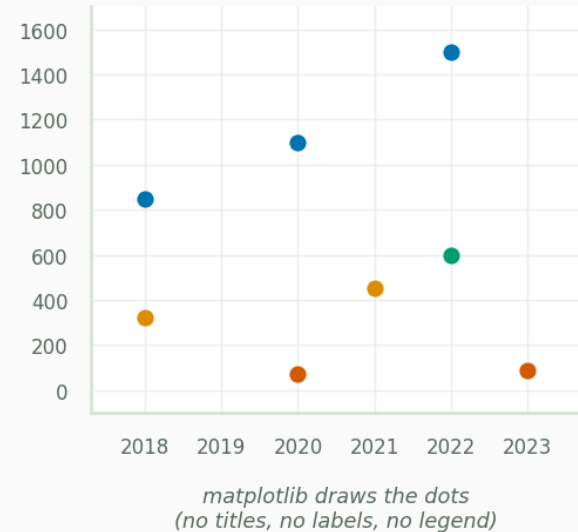
one row per mission, one variable per column

agency	year	cost (M USD)
NASA	2018	850
ESA	2018	320
NASA	2020	1100
SpaceX	2020	70
ESA	2021	450
NASA	2022	1500
CNSA	2022	600
SpaceX	2023	90

2. Map

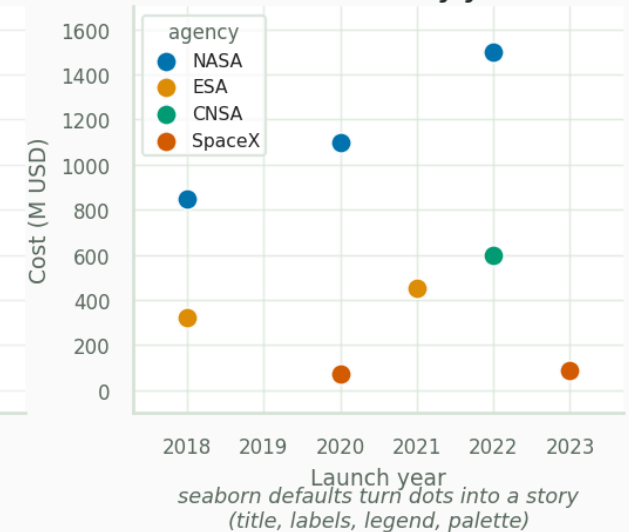


3. Render



4. Explain

Mission cost by year



Same eight rows, four stages: **tidy** shapes the data, **map** assigns meaning, **render** draws the dots, and **explain** adds the labels and legend that turn dots into a story. Use seaborn when that final step is the goal.



Where `seaborn.objects` Fits in 2026

Why it is worth learning

- Consistent property names such as `color` instead of a separate `hue` mental branch
- Composable grammar: mark + stat + move + scale
- Great for teaching plot construction explicitly

Reality check

- The classic `sns.*` API is still common and not going away.
- The official seaborn docs still describe `objects` as experimental and incomplete.
- So learn the grammar, but do not pretend the rest of seaborn disappeared.

```
import seaborn.objects as so

(so.Plot(missions, x="launch_year", y="cost_million_usd", color="agency_short")
 .add(so.Dot())
 .label(title="Mission cost by year"))
```

Demo 02 builds the same chart step by step, layering Plot → Mark → Stat → Move → Scale → Facet across ten short cells.



Marks, Stats, and Moves Are Separate Choices

Mark

Dot, Line, Bar, Bars.

What geometric shape will represent the data?

Stat

Agg, Hist, Est, PolyFit.

What transformation or summary happens before drawing?

Move

Dodge, Jitter, Stack.

How do overlapping marks separate or combine visually?

```
(so.Plot(missions, x="agency_short", y="cost_million_usd", color="decade")  
  .add(so.Bar(), so.Agg("mean"), so.Dodge()))
```

If you want a summary, ask for it explicitly. Do not rely on vague "the chart will figure it out" behavior.



Scales and Facets Rescue Clutter

When a single view fails

- Use log scale when values differ by orders of magnitude.
- Use facets or small multiples when many categories overlap.
- Use direct comparisons, not dual axes, when the audience must read the scale quickly.

Compact example

```
(so.Plot(missions, x="launch_year", y="cost_million_usd", color="agency_short")  
  .add(so.Dot())  
  .scale(y="log")  
  .facet(col="mission_class"))
```

Transform the axis when magnitude hides variation. Split the view when overlap turns the plot into spaghetti.

Fix the visual design first. If one line dominates every other line, a second axis is almost never the cleanest answer.



Data-Prep Bridge: From WKT Text to Map Columns

Vectorized parsing beats row loops

```
pattern = r"POINT Z \((?P<lon>.*?) (?P<lat>.*?) (?P<depth>.*?)\)"
coords = quakes["geometry"].str.extract(pattern).astype("float64")
quakes = quakes.join(coords)
```

Why it matters

- Plotting libraries need numeric `lon` / `lat` columns.
- Regex extraction keeps the operation vectorized and notebook-friendly.
- Once parsed, the same data can feed a static scatter, a map, or an aggregated density view.

Fun fact 5

A cell like `POINT Z (-169.6766 52.3338 10)` looks like plain text, but it is already a compact 3D geometry record waiting to become map coordinates.



Checkpoint: Static Grammar and Data Prep

Q5

What does `so.Agg("sum")` communicate in a `seaborn.objects` pipeline?

- A) Aggregate before drawing, using sum as the summary
- B) Add glitter to the bars
- C) Turn the plot interactive
- D) Ask the chart to be more aggressive

Show Answer

Q6

Which approach is the best classroom default for extracting longitude and latitude from a WKT geometry column?

- A) Loop over rows in Python and split strings manually
- B) Use `Series.str.extract(...)` with named groups
- C) Rewrite the CSV by hand
- D) Ask the projector to parse it

Show Answer



Plotly, Maps, and Performance

Interactive visuals, low-noise maps, and browser-safe workloads



Plotly Is for Exploration and Details on Demand

Use Plotly when

- You want hover details, zoom, pan, or filtering in the browser.
- The audience will explore before drawing a conclusion.
- A static PNG would collapse too much context.

Typical Plotly Express pattern

```
fig = px.scatter(  
    missions,  
    x="launch_year",  
    y="cost_million_usd",  
    color="agency",  
    hover_name="mission_name",  
    hover_data=["destination", "success"],  
)
```

The Python side describes the scene. JavaScript and WebGL render the interactive result in the browser.



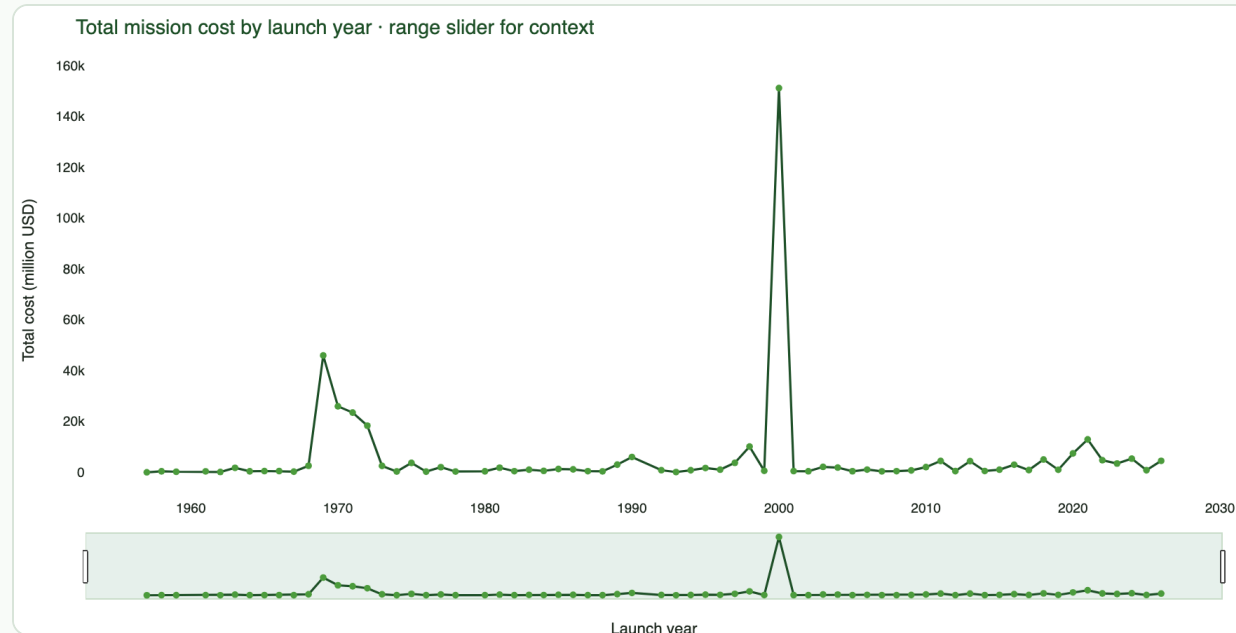
Hover, Sliders, and Focus + Context

Useful interaction patterns

- `hover_name`: one human-readable title for each point
- `hover_data`: add detail only when requested
- Range slider: show the full timeline and a zoomed window at once

Compact example

```
fig = px.line(df, x="year", y="revenue", color="sector")
fig.update_xaxes(rangeslider_visible=True)
fig.update_layout(hovermode="x unified")
```



Main view + slim navigator strip: overview first, zoom and filter, then details on demand.

Interactive features should reveal context, not replace clear design. Hover is a supplement, not a license to hide all labels and structure.



Token-Free Tile Maps with MapLibre

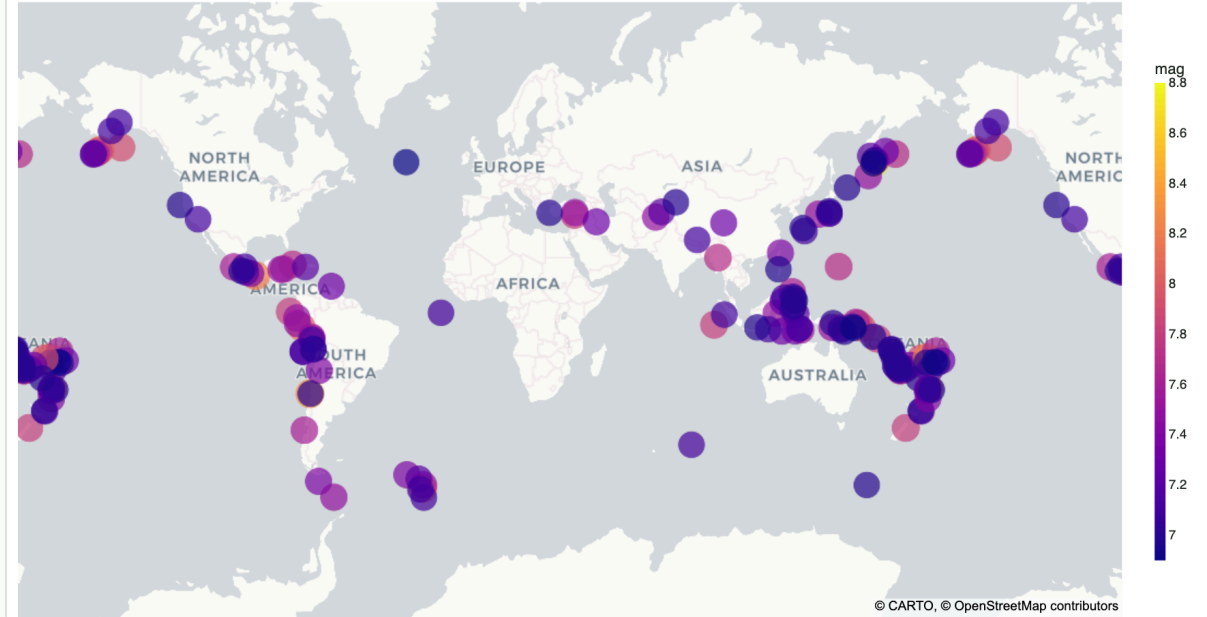
Good map defaults

- Reach for `px.scatter_map` / `px.density_map` for tile-based maps.
- Pass parsed `lat` and `lon` columns explicitly.
- Pick quiet tiles such as `carto-positron` or `carto-darkmatter`.
- Reserve color and size for meaningful data, not for map decoration.

Typical pattern

```
fig = px.scatter_map(
    quakes,
    lat="lat", lon="lon",
    size="mag", color="mag",
    hover_name="title",
    map_style="carto-positron",
    zoom=0.7,
)
```

Largest earthquakes · `px.scatter_map` + `carto-positron`



Largest earthquakes rendered with `px.scatter_map` on the `carto-positron` MapLibre basemap.

Fun fact 6

Plotly's tile maps render through MapLibre, so the built-in styles work straight out of the box — no API token, no signup, no surprise paywall mid-demo.



Large Data: Aggregate Before You Render

Browser-safe strategy

- Sample if the question tolerates approximation.
- Bin or aggregate when overlap destroys meaning.
- Use density views for huge spatial or scatter workloads.
- Let pandas / Polars do the heavy lifting upstream.

Do not send one million raw points to the browser just because you can. Make the compute layer summarize first, then let the visual layer explain.

1

Read / scan

2

Filter early

3

Aggregate

4

Hand off

5. Render the smaller result

Plotly should see an answer-shaped table, not the whole warehouse.

Special Cases: One Good Pattern, Two Risky Ones

Population pyramid: good special case

- Mirror horizontal bars around a central zero axis.
- Label the x-axis with absolute values so readers do not think negative people exist.
- Great for age structure comparisons.

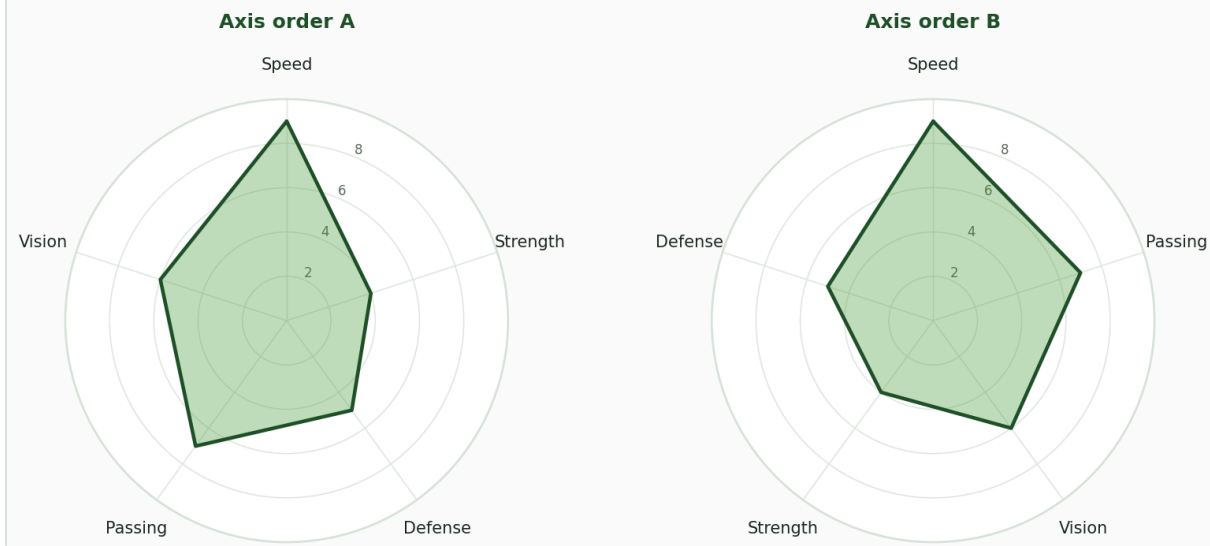
Dual axes: only use them when the relationship between two different units is the story and you can color-code axes honestly. Most of the time, split the view instead.

Export formats

`fig.write_html( ... )` for interactive sharing.

`fig.write_image( ... )` for static PNG / SVG export (with Kaleido).

Same values, different radar shape after reordering axes



Fun fact 4

Radar charts can change their visual impression simply by reordering the axes. The values stay the same, but the shape feels different, which is one reason they are a risky default.



Checkpoint: Interactive Visualization

Q7

You need to show 500,000 earthquake points on a map. What is the best first reaction?

- A) Send every point to the browser and hope the laptop forgives you
- B) Consider sampling, binning, or density aggregation first
- C) Convert the map into a pie chart
- D) Pretend the browser tab crash is an advanced feature

Show Answer

Q8

Which Plotly Express map function is the current default for token-free tile maps?

- A) `px.scatter_map`
- B) `px.scatter_mapbox`
- C) `px.globe_of_doom`
- D) The one from a 2023 tutorial copied without reading the warning

Show Answer



Lecture Summary

Question First

- Pick the chart family from the question, not from habit
- Prefer workhorse charts over decorative ones
- Check readability and accessibility before export

Static Stack

- Use Matplotlib's OO API as the foundation
- Use seaborn for clearer semantics and plotting grammar
- Transform the data shape before asking the chart to explain it

Interactive Stack

- Use Plotly when hover, zoom, or maps add real value
- Aggregate before plotting large datasets
- Choose HTML for exploration, SVG/PDF/PNG for delivery

The real skill is not knowing three plotting libraries. It is deciding what the reader should notice, then sending exactly enough data through the right layer of the stack to make that answer obvious.



Thank You for Your Attention

"The greatest value of a picture is when it forces us to notice what we never expected to see." - John Tukey

Chart choice

Matplotlib

Seaborn

Plotly

Maps

Accessibility



Python Course 2025/26/2

Lecture 11: Applied Python for Machine Learning, Computer Vision, and PyTorch



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli



Lecture Overview

Agenda

- Move from DataFrames to design matrices and target vectors
- Use scikit-learn with pipelines, splits, and cross-validation
- Treat clustering as discovery, not as a supervised shortcut
- See images as arrays before treating them as "AI magic"
- Introduce neural networks and PyTorch from scratch on the Iris dataset

Learning objective

Build a stronger engineering habit for applied ML in Python: choose the representation carefully, protect the evaluation, and understand when the problem still fits scikit-learn versus when you need PyTorch.

By the end, you can

Explain a modern classroom-safe workflow for tabular ML, clustering, computer vision, and introductory deep learning, then map each step to concrete Python tools and code.

Honest evaluation

Pipelines

Images as arrays

Transfer learning

Autograd

Lecture 11 is where the Python data stack starts to feel like applied machine learning instead of only data wrangling.



From Tables to Models

Start with the representation, then protect the evaluation

The Python ML Map for Today

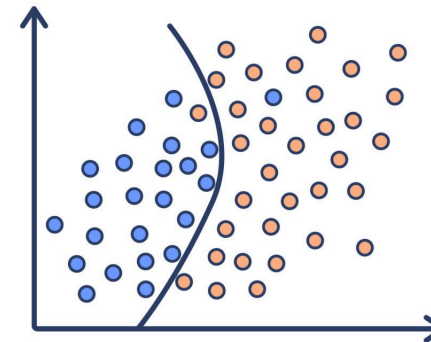
The working map

- **Supervised learning:** predict a known label from examples
- **Unsupervised learning:** discover structure without labels
- **Computer vision:** treat images as arrays, then use pre-trained models responsibly
- **PyTorch basics:** tensors, gradients, modules, and inference

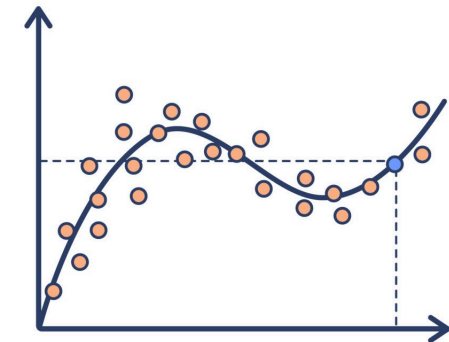
Outside this workflow

Reinforcement learning is important, but it is a different engineering loop: agent, environment, reward, exploration. Here the focus stays on the Python stack you can run and inspect in a normal data-science workflow.

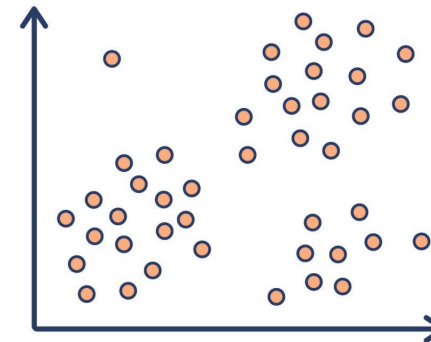
MACHINE LEARNING TASKS



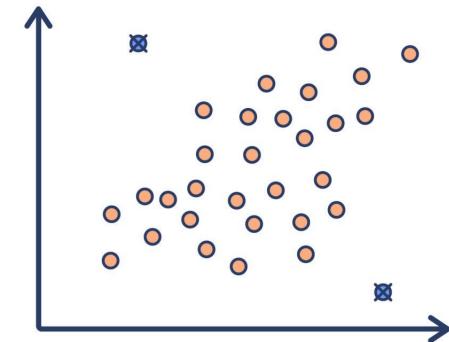
Binary classification



Regression



Clustering



Anomaly detection

Different tasks, same habit: define the question, choose the representation, then choose the model family.



First Vocabulary: Rows, Features, Targets

One table becomes learning examples

- **Row / sample / observation:** one case the algorithm can learn from.
- **Feature:** an input column that describes the row.
- **Target / label:** the answer column we want to predict.
- **Supervised learning:** features plus known targets.
- **Unsupervised learning:** features without known targets.

Iris example in plain language

Column	Role
sepal length, sepal width	features
petal length, petal width	features
species	target label

A well-defined ML task names the row, the input columns, and the answer column.

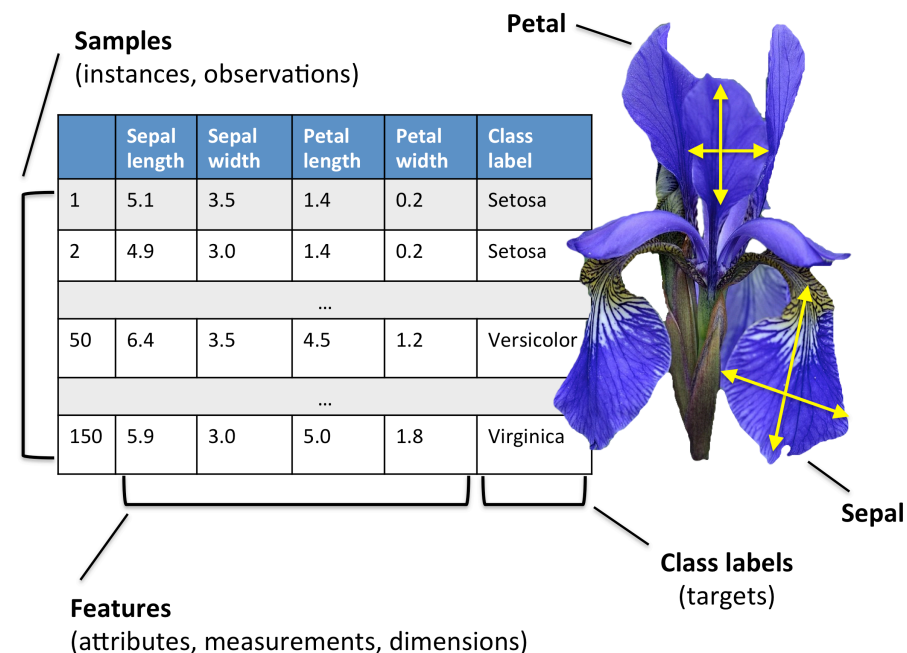
From DataFrame to X and y

The array view scikit-learn expects

- **X**: the feature matrix with shape `(n_samples, n_features)`
- **n_samples**: how many rows/examples we have
- **n_features**: how many input columns describe each row
- **y**: the target vector, one answer per row in supervised learning
- Most algorithms need numbers, so text and categories must be encoded first

Fun fact 1

The Iris dataset was published by Ronald Fisher in 1936. It is still ML's classroom "hello world" because the whole dataset is small enough to inspect directly, but rich enough to show real classification behavior.



The computer does not understand flowers. It receives consistent numeric columns and a target label.



Now Introduce the Estimator

scikit-learn vocabulary

- An **estimator** is a Python object that follows scikit-learn's learning API.
- It has `fit( ... )` when it can learn from data.
- Predictors add `predict( ... )`; transformers add `transform( ... )`.
- **Hyperparameters** are set before fitting; learned parameters appear after fitting, often with a trailing underscore.

"Model" is the intuitive word; "estimator" is scikit-learn's precise API term.

Minimal API shape

```
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

This shared shape is why different ML algorithms can be swapped in and compared without rewriting the whole notebook.



KNN: Our First Estimator

Core intuition

- Place each row as a point in feature space
- Measure distance from a new point to the stored training points
- Look at the closest `k` neighbors
- Vote on the class label
- “What are the closest examples I already know, and what do they say?”

Fun fact 2

KNN is called a **lazy learner** because fitting can be almost just "store the data," while prediction does the heavy work of measuring distances and voting.

Why KNN is a good first model

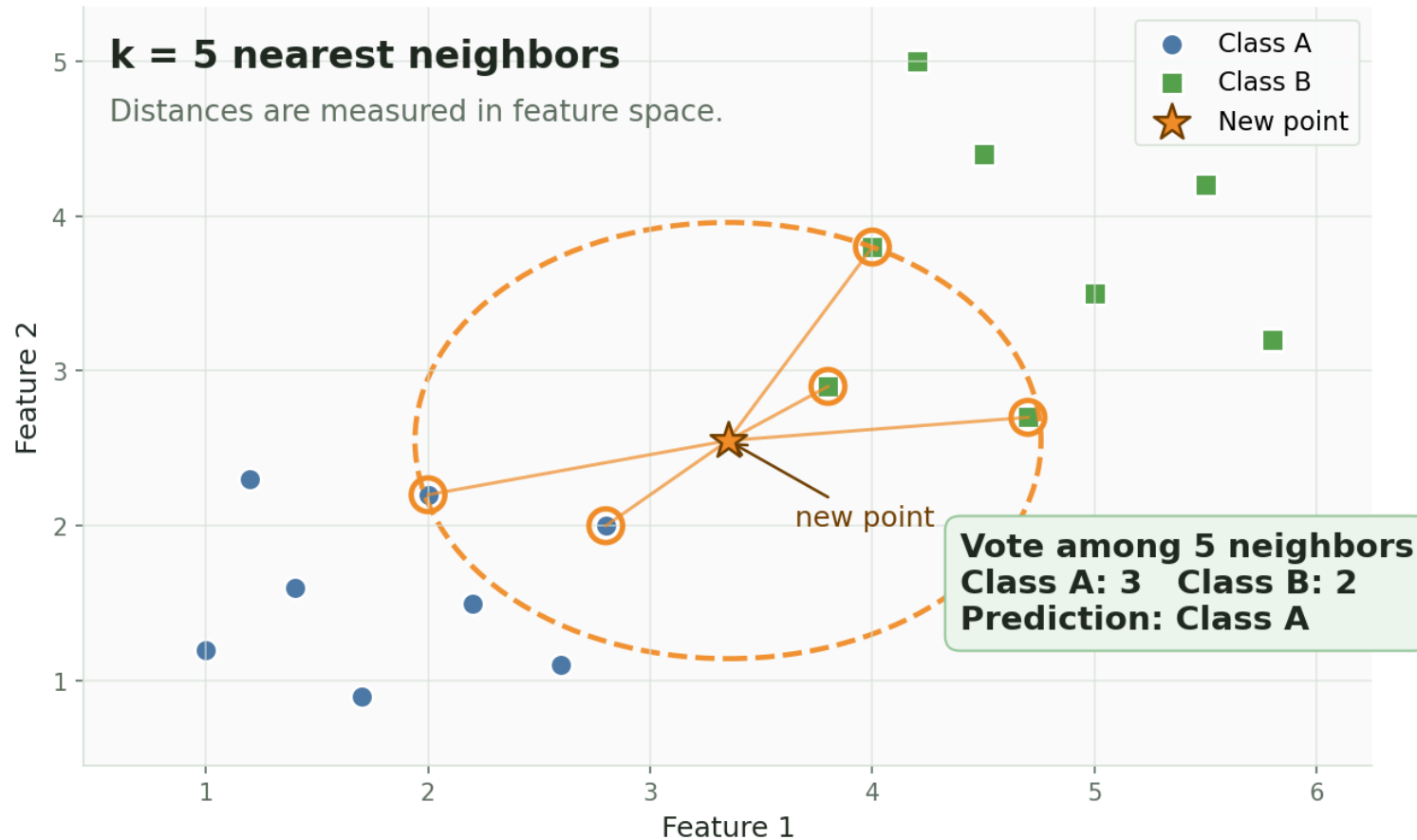
KNN starts with geometry: distance, nearest points, and voting. No calculus-heavy optimization is needed to understand the basic behavior.

$$d(\mathbf{x}, \mathbf{x}_i) = \sqrt{\sum_{j=1}^p (x_j - x_{ij})^2}$$

```
dist = ((X_train - x_new) ** 2).sum(axis=1) ** 0.5
nearest = dist.argsort()[:k]
prediction = y_train[nearest]
```

The code computes the same distance over feature columns, then sorts and votes.

KNN: Distance Turns Into a Vote



Takeaway: KNN does not learn a formula during fitting. It stores the training examples, measures distance from the new point, and lets the closest neighbors vote.

The Estimator Lifecycle

Same verbs, many models

1

Instantiate

choose
hyperparameters

2

Fit

learn from
training
data

3

Predict

apply to
unseen data

4

Evaluate

measure
generalization

If you learn this lifecycle once, you can reuse it across KNN, logistic regression, random forests, SVMs, and many more estimators.

Minimal code skeleton

```
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
score = accuracy_score(y_test, y_pred)
```

The point of scikit-learn is not only the algorithms. The API design is what makes it teachable, reusable, and production-friendly.



Evaluation: Did the Model Generalize?

General idea

- Train on examples with known answers.
- Hide some known answers in a validation or test set.
- Ask the fitted model to predict those hidden answers.
- Compare `y_pred` with `y_true`.
- Choose a metric that matches the task: accuracy, error, F1, RMSE, and so on.

Key word: generalization means performance on data the model did not use while learning.

KNN on Iris

```
y_pred = model.predict(X_test)
score = accuracy_score(y_test, y_pred)
```

- `X_test`: flower measurements kept aside
- `y_test`: the true species labels
- `y_pred`: KNN's predicted species labels
- `accuracy`: correct predictions divided by all test rows

For KNN classification, evaluation asks: when a new flower appears, does the neighbor vote match the species we already know?



Do Not Peek at the Test Set

Modern evaluation habit

- Split before learning anything from the data
- Use `stratify=y` to preserve class proportions in both splits
- Keep the test set as a final exam, not as a steering wheel
- Anything that learns parameters must learn them on the training data only

Fun fact 3

`42` became a programmer in-joke because of Douglas Adams, but the real lesson is boring and important: fixed random seeds make your demo reproducible.

Safe split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

`stratify=y` means: if Iris has one third of each species, the train and test sets keep roughly the same mix. The test data still stays untouched for model choices.



Scaling: Make Distances Fair

Why scaling exists

- Many algorithms compare numeric columns by distance or size.
- A column with a larger range can dominate the calculation.
- Scaling puts features onto a comparable numeric scale.
- `StandardScaler` subtracts the training mean and divides by the training standard deviation.

Role in KNN on Iris

- KNN decides by Euclidean distance in feature space.
- Iris features are all centimeters, but their spreads still differ.
- Without scaling, wider-spread features get more influence in the neighbor vote.
- After scaling, one unit means roughly "one standard deviation" for every feature.

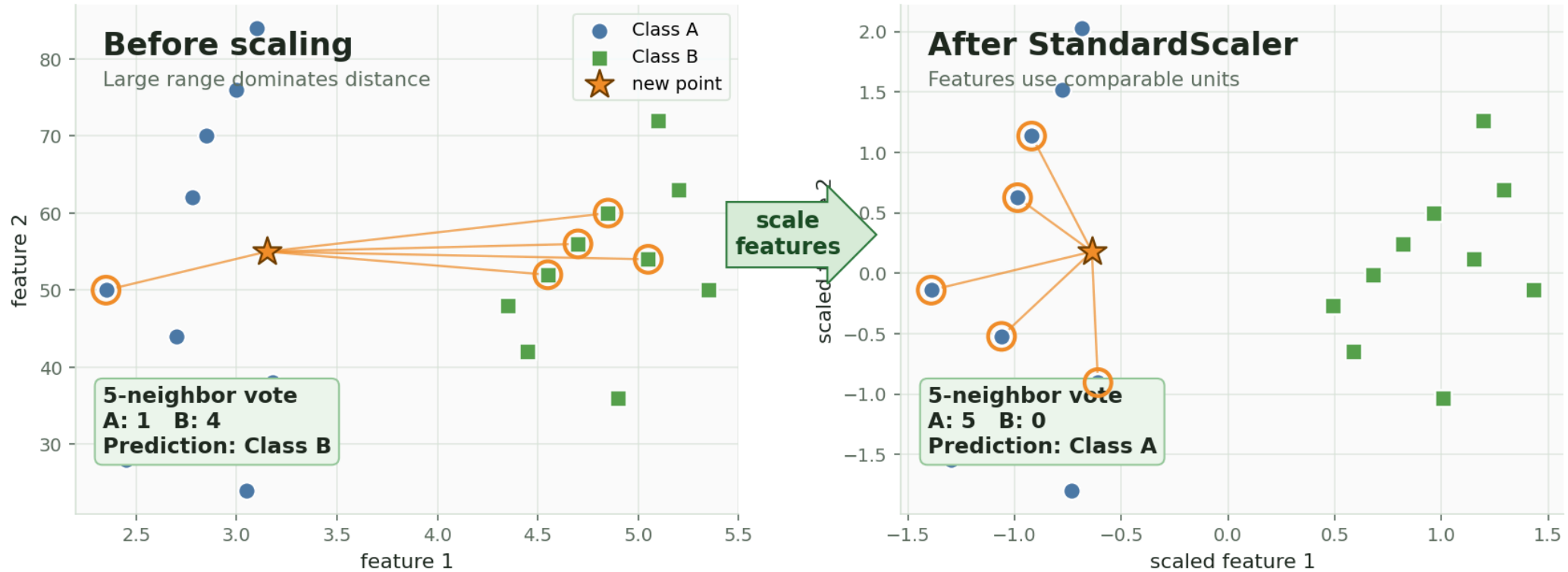
Important: scaling is a learned preprocessing step, so its mean and standard deviation must come from training data only.

In this example, scaling makes sepal and petal measurements contribute more fairly to the neighbor search.

Scaling Changes What "Nearest" Means



Scaling changes the geometry KNN sees



Takeaway: scaling does not change the labels; it changes the geometry used by distance-based models.



Pipelines: Analytical Thinking as Code

Problem-solving workflow

1. Represent

rows, features, target

2. Transform

scale / encode safely

3. Model

fit the estimator

4. Evaluate

test generalization

What the pipeline object does

- Bundles preprocessing and the estimator into one workflow.
- Makes the order explicit: scale first, then KNN.
- Prevents forgetting the same transform at prediction time.
- During cross-validation, each fold fits the scaler only on that fold's training rows.

For Iris KNN, the pipeline says: learn scaling from training flowers, scale the measurements, then run the neighbor vote.



Cross-Validation: Tune Without Touching the Final Exam

Why a single split is not enough

- A validation score can depend on which rows happened to land in the split.
- Trying many settings on the test set quietly turns the test set into training feedback.
- Cross-validation estimates performance by rotating the validation fold inside the training data.
- The held-out test set stays untouched until the final check.

Use cross-validation for model choices; use the test set once for the final estimate.

How `cv=5` works

Fold 1
validate

Fold 2
validate

Fold 3
validate

Fold 4
validate

Fold 5
validate

- For each candidate `k`, fit the pipeline five times.
- Each run trains on four folds and validates on the remaining fold.
- Average the five validation scores and choose the best `k`.

In Iris KNN, cross-validation helps choose `n_neighbors` using only `X_train` and `y_train`.



Pipelines + Cross-Validation: 2026 Default Habit

Keep preprocessing tied to the model

```
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

pipe = make_pipeline(
    StandardScaler(),
    KNeighborsClassifier(n_neighbors=5),
)

scores = cross_val_score(pipe, X_train, y_train, cv=5)
```

A pipeline protects you from forgetting transforms and from leaking test-set statistics into training.

Two definitions before the habit

- **Pipeline:** a chain of preprocessing steps plus the final model.
- **Cross-validation:** repeated train/validation splits inside the training data.
- **Leakage:** information from validation or test data sneaks into fitting.

Rule of thumb: if a step has a `fit` method, it belongs inside the training workflow, not as a detached preprocessing side quest.



Checkpoint: Tabular ML Workflow

Q1

Where should `StandardScaler()` usually live when you train a KNN model?

- A) In a pipeline fitted on the training workflow
- B) Only on the full dataset before splitting
- C) On the test set first, so it gets special treatment
- D) Inside a motivational speech to the pixels

Show Answer

Q2

You want to choose the best value of `k`. What is the safest classroom habit?

- A) Repeatedly check the test score until the number looks heroic
- B) Use cross-validation on the training set, then check the test set once
- C) Pick the largest `k` because more neighbors means more wisdom
- D) Ask the flowers which `k` they prefer

Show Answer



Discovery and Handoff

Clusters, heuristics, and what happens after the notebook



K-Means: Clustering Without Labels

Where it fits

- **Clustering** is unsupervised: there is no target vector `y`.
- The question changes from "predict the known label" to "discover useful groups."
- A **cluster** is a group of points that are close in feature space.
- K-Means asks for the number of clusters `k` up front.

```
kmeans = KMeans(  
    n_clusters=3,  
    random_state=42,  
    n_init="auto",  
)  
labels = kmeans.fit_predict(X)
```

Centroid-based idea

A **centroid** is the current center of a cluster. K-Means alternates between assigning points to centroids and moving centroids to the mean of their assigned points.

1. Initialize

place centroids

2. Assign

attach each point to the nearest centroid

3. Update

move centroids to the cluster means

4. Repeat

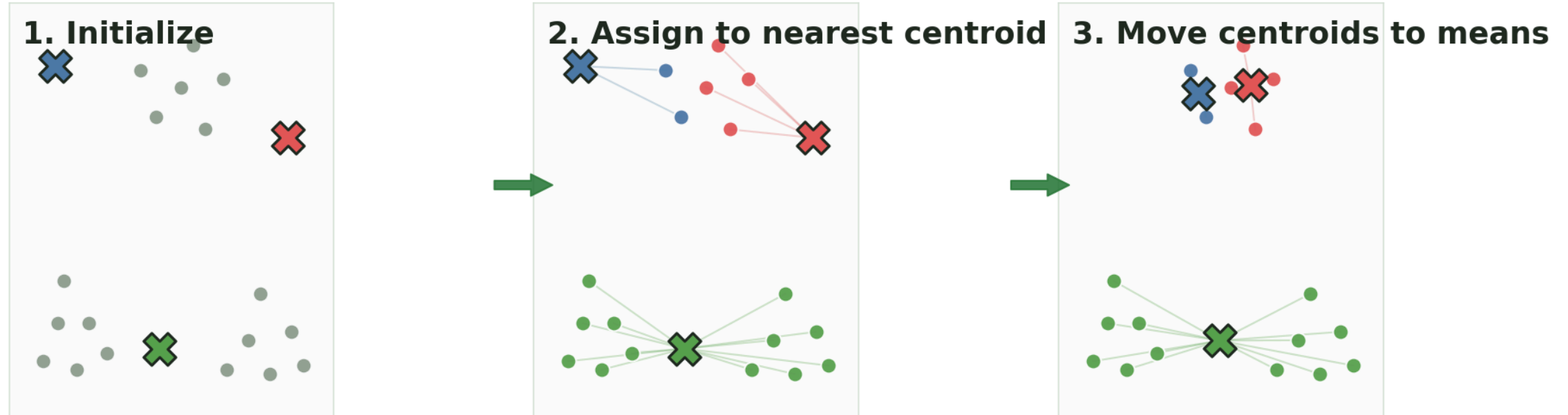
stop when the layout stabilizes

K-Means is useful when compact, distance-based groups are a reasonable model. The cluster labels are discovered IDs, not ground-truth answers.



K-Means: Assign, Update, Repeat

K-Means repeats: assign points, then move centroids



Centroid = current cluster center. Update = average the points assigned to that center.

Takeaway: K-Means keeps changing the centroids until assignments stop changing much.



Choosing k: What Are We Trying to Answer?

The question

K-Means needs `n_clusters` before fitting. We are asking: how many groups are useful, stable, and interpretable?

Inertia + elbow

Inertia is the total squared distance from points to their centroids. The **elbow** is where extra clusters stop helping much.

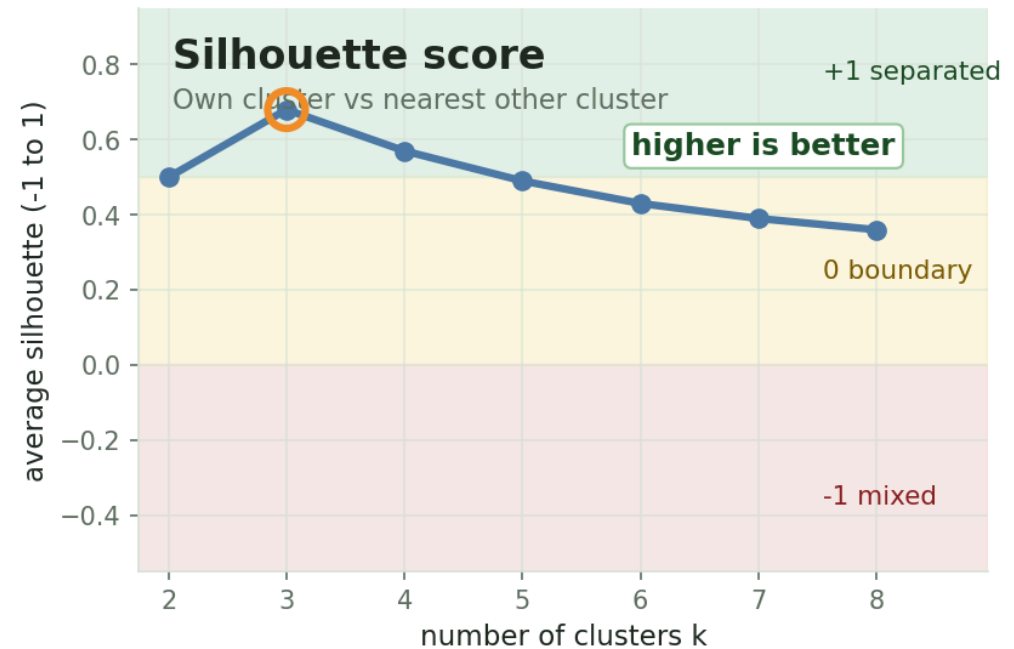
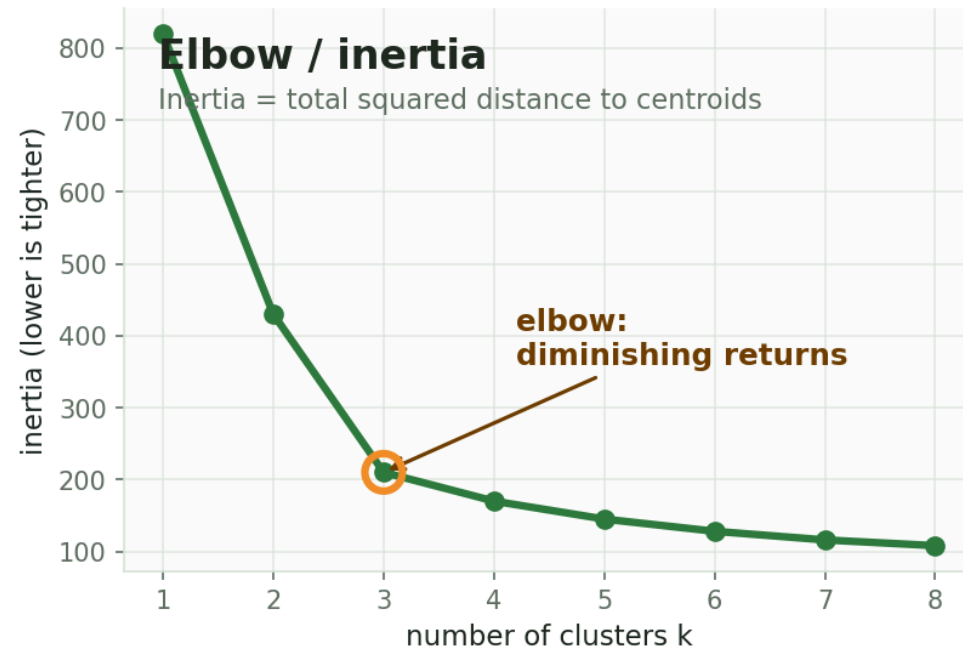
Silhouette

Silhouette compares how close a point is to its own cluster versus the nearest other cluster. Higher usually means cleaner separation.

No target labels means no ordinary accuracy. Cluster choice combines geometry-based diagnostics with domain judgment.

Choosing k: Visual Interpretation

Choosing k asks: how many groups are useful and stable?



No target labels means no ordinary accuracy. Use geometry-based diagnostics plus domain judgment.

Takeaway: the elbow suggests diminishing returns; silhouette checks whether the clusters are separated rather than merely tighter.



Elbow Is a Heuristic, Not a Law

Typical search loop

```
ks = range(1, 9)
inertia = []

for k in ks:
    model = KMeans(
        n_clusters=k,
        random_state=42,
        n_init="auto",
    )
    inertia.append(model.fit(X).inertia_)
```

Then plot `k` against inertia and look for diminishing returns.

What the elbow can and cannot tell you

- Lower inertia means tighter clusters, not necessarily better business meaning
- Some datasets show a clear elbow; others do not
- Use silhouette scores, domain knowledge, and visual inspection together

A healthy attitude: the best cluster count is the one that is useful, stable, and interpretable, not merely the one that makes a plot look pretty.



From Notebook to Reusable Artifact

Persist the whole workflow, not just the estimator

```
import joblib

joblib.dump(pipe, "iris_knn_pipeline.joblib")
reloaded = joblib.load("iris_knn_pipeline.joblib")
```

Saving the pipeline means scaling and model logic travel together.

Safe artifact handoff

- Load `pickle` / `joblib` files only from trusted sources.
- Store package versions beside the artifact.
- Keep the training script so the artifact can be rebuilt.
- Use a more reviewable format such as `skops.io` when trust or auditability matters.

Important: a saved model is executable project state. Treat stale or untrusted artifacts as security and reproducibility risks.



Why scikit-learn Transfers So Well

One API, many estimators

Task	Typical estimator	Same engineering habit
Classification	<code>KNeighborsClassifier</code>	split, fit, predict, score
Linear models	<code>LogisticRegression</code>	same API, different decision boundary
Ensembles	<code>RandomForestClassifier</code>	same API, more complex internals
Clustering	<code>KMeans</code>	fit and inspect labels / centroids instead of a target label

You are not only memorizing library syntax. You are learning a reusable workflow pattern for applied machine learning in Python.



Checkpoint: Clustering and Handoff

Q3

Why can you not talk about "accuracy" for plain K-Means in the same way as for KNN on Iris?

- A) Because clustering has no ground-truth labels built into the training task
- B) Because unsupervised learning is allergic to numbers
- C) Because scikit-learn forgot to implement it
- D) Because centroids refuse to sit still for grading

Show Answer

Q4

Which persistence statement is the safest?

- A) Any `joblib` file from the internet is fine if the filename sounds intelligent
- B) Persisted models are version-sensitive and should come from trusted sources
- C) Once saved, a model artifact is guaranteed to load forever
- D) Docker personally negotiates with broken pickles

Show Answer



Images Are Arrays

Before you use a detector, understand the numbers

Images Start as Arrays



A photo feels semantic to us. To Python, it is still an array with width, height, channels, and a numeric dtype.

Core representation

- A pixel stores intensity or color at one location.
- Channels store separate measurements such as red, green, and blue.
- Grayscale image: `(H, W)`
- RGB image: `(H, W, 3)`
- Common storage dtype: `uint8` in the range 0-255
- Common model dtype: `float32` after normalization

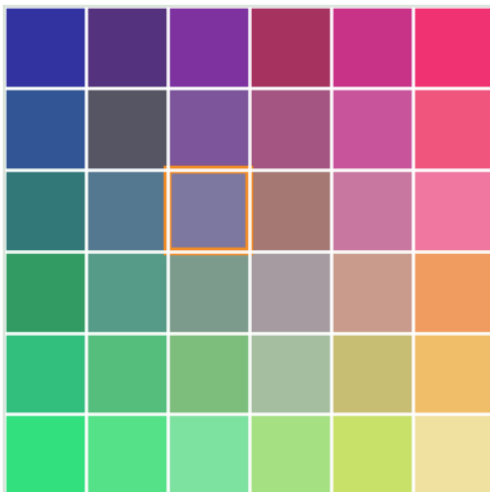
Fun fact 4

A 1920x1080 RGB frame contains 6,220,800 numbers. At 30 FPS, a webcam is handing you roughly 186 million color values every second.

Image Tensor Anatomy

An image is a numeric tensor before it is a “scene”

Image grid



height x width pixels

One pixel

RGB = 3 numbers



R = 212



G = 155



B = 106

Stored as uint8
values 0..255

Model input

shape: (H, W, 3)



uint8 0..255

float32 0..1

Computer vision starts by making this
representation explicit.

Data science view: images become features once their shape, dtype, channel order, and normalization are explicit.



Fixed Filters: Edges Are Gradients

From image to edge map

```
from skimage import color, filters, io

image = io.imread("sobel_input.jpg")
gray = color.rgb2gray(image)
edges = filters.sobel(gray)
```

Sobel is a fixed filter: it scans local neighborhoods and highlights sudden brightness changes.

Context: before learned models, many vision pipelines used hand-built features such as edges, corners, textures, and color histograms.



Input image



Gradient magnitude



Pre-trained Models Reuse Learned Visual Features

Why pre-trained models matter

- Large vision models need many labeled images, GPUs, and training time.
- Pre-trained weights already encode useful visual patterns.
- Model packages often include class names and expected preprocessing rules.
- For a data-science workflow, this turns images into reusable predictions and features.

A small local checkpoint is often the right engineering choice when reproducibility, startup time, and offline execution matter.

Detector pipeline

1. Load weights

tiny local checkpoint

2. Preprocess

resize / normalize as needed

3. Infer

boxes, labels, confidences

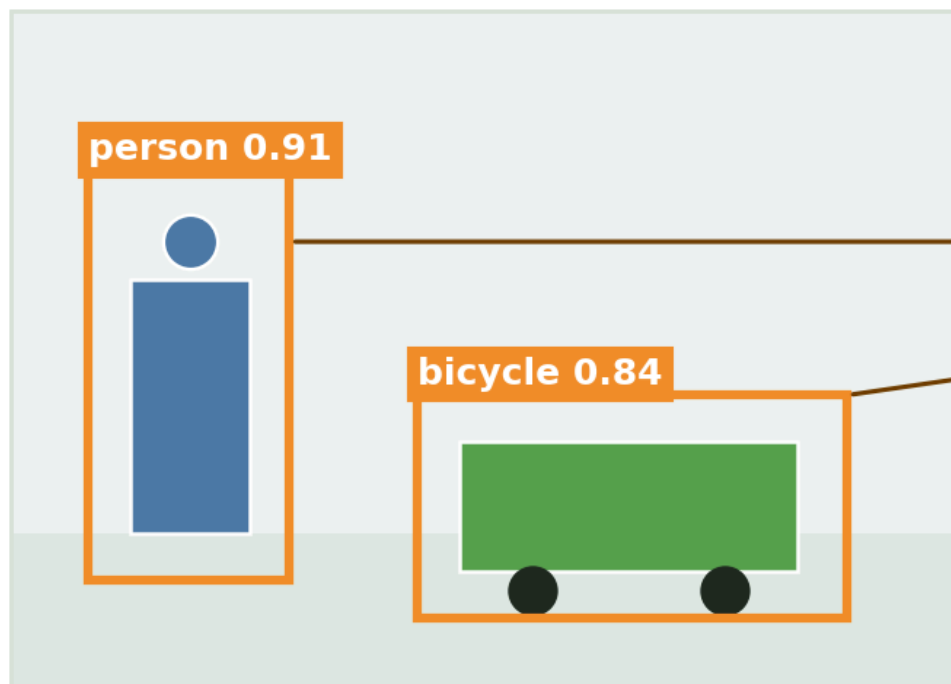
4. Visualize

annotate the original image

In 2026, model names keep changing. The durable habit is: understand what the weights expect and what the result object gives back.

Object Detection Outputs Are Structured Data

Object detection turns an image into structured records
Detector output



Bounding box

where the object is: $x1, y1, x2, y2$

Class label

what category the model predicts

Confidence

how strongly the model supports that label

Post-processing

remove duplicate boxes, then draw results

Unlike classification, detection answers both: “what is it?” and “where is it?”

Data science view: detections can be counted, filtered, joined with metadata, tracked over time, or inspected as records.

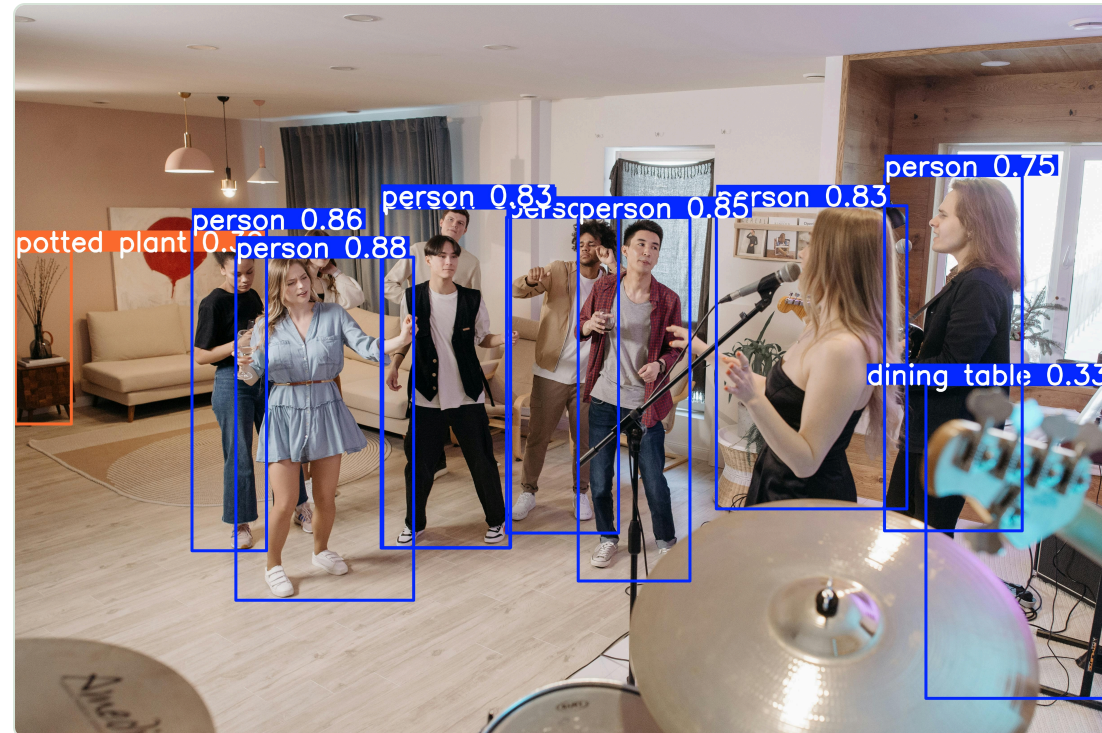
Local Object Detection Inference

Offline-friendly inference

```
from ultralytics import YOLO

model = YOLO("yolov10n.pt")
result = model("detection_input.jpg")[0]
result.save(filename="detection_result.jpg")
```

Ultralytics also supports `stream=True` when you process longer videos or larger batches and want to control memory use.



Local inference keeps the workflow reproducible and makes the detector output inspectable.



Checkpoint: Computer Vision

Q5

Which shape best matches a typical RGB image stored as a NumPy array?

- A) (H, W, 3)
- B) (3, H)
- C) (H)
- D) (very photogenic, surprisingly dramatic)

Show Answer

Q6

Why is a small local checkpoint a good practical choice for object detection inference?

- A) It makes the workflow reproducible and starts quickly
- B) It proves that smaller models are always more accurate
- C) It removes the need to inspect outputs
- D) The GPU spirits demand it

Show Answer



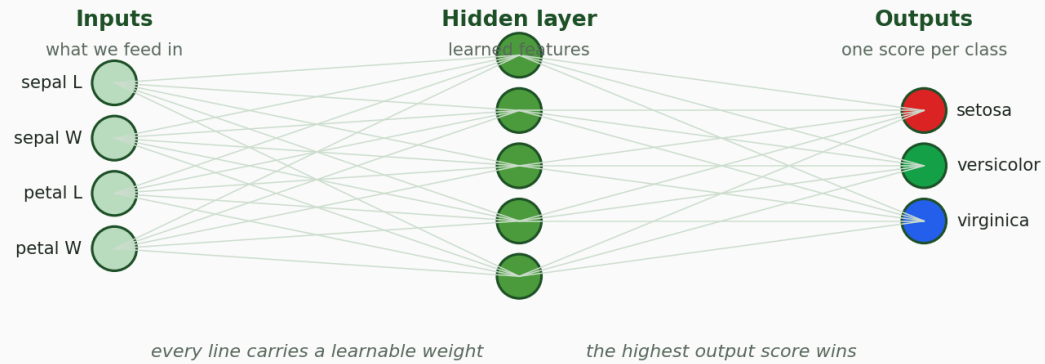
Neural Networks and PyTorch

A first, friendly tour: what they are, what PyTorch gives you, and how training really works

What Is an Artificial Neural Network?



A neural network = layers of small units that learn good combinations of inputs



A new word: neural network

- An **artificial neural network** is a stack of small math units arranged in **layers**.
- The number on each connection is called a **weight**; weights are what the network learns.
- Each unit combines its incoming numbers and passes the result on; later layers build on earlier ones.
- "Neural" comes from a loose biological inspiration; the math is just sums and simple bends.

Read the picture: 4 Iris features go in, the network mixes them through a hidden layer, and 3 output scores come out — one per species.



Why Use a Neural Network at All?

KNN works fine when ...

- Data is small and tabular (like Iris).
- The right answer is "look at nearby examples and vote".
- Plain Euclidean distance between rows is meaningful.

For these problems scikit-learn often wins on speed, simplicity, and clarity.

... but distance is not always enough

- For images, "distance between two photographs" rarely matches "same object".
- For text or audio, useful patterns are buried under raw numbers.
- A neural network **learns its own features** instead of relying on a fixed rule.

We start with familiar Iris so the mechanics are clear; the same machinery scales to images, text, and audio.



PyTorch in One Slide: What It Is and When to Use It

What PyTorch gives you

- A Python library for building and training neural networks.
- **Tensors**: numeric arrays for math (defined on the next slide).
- **Autograd**: it computes gradients automatically.
- **GPU support**: the same code runs on a graphics card for speed.

When to reach for it

- The model is custom: a layer stack, a custom loss, multiple inputs.
- The inputs are large or unstructured (images, text, audio).
- You have a GPU available and want it used.

When not: small, clean tabular problems with strong `sklearn` baselines. PyTorch is **more flexible**, not automatically **better**.



Tensors: PyTorch's Working Object

A tensor is just a numeric array — it can have any number of dimensions

Scalar

0-D tensor

7.0

shape: ()

`torch.tensor(7.0)`

Vector

1-D tensor

5.1 3.5 1.4 0.2

shape: (4,)

one Iris row

Matrix

2-D tensor

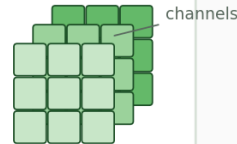
5.1	3.5	1.4
4.9	3.0	1.4
6.2	3.4	5.4

shape: (rows, cols)

the X_train table

3-D tensor

and beyond



shape: (H, W, 3)

RGB image tensor

What is a tensor?

- A numeric array, like a NumPy array.
- Two superpowers a NumPy array does not have: it can live on a **GPU**, and it can **remember the operations** done to it (used by autograd).
- Vocabulary: **scalar** (one number), **vector** (one row), **matrix** (rows × columns).

From pandas / NumPy to a tensor

```
import torch

X = torch.tensor(X_train_scaled,
                  dtype=torch.float32)
```

Devices: Where Tensors Live



Device-aware code

```
if torch.cuda.is_available():
    device = torch.device("cuda")
elif torch.backends.mps.is_available():
    device = torch.device("mps")
else:
    device = torch.device("cpu")

model = model.to(device)
X = X.to(device)
```

Do not hardcode `.cuda()`. The same script should run on a CUDA GPU, an Apple Silicon Mac, or plain CPU.

Vocabulary

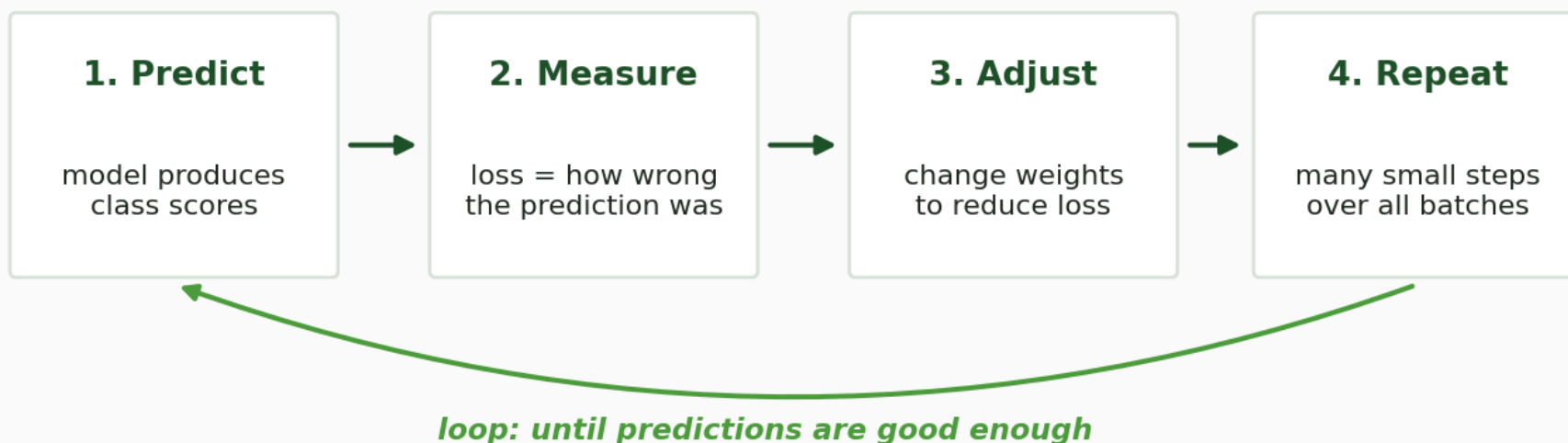
- **Device:** where a tensor lives and where the math runs.
- `cpu` : universal fallback, always available.
- `cuda` : NVIDIA GPUs.
- `mps` : Apple Silicon GPU acceleration.
- Tensors involved in one operation must live on the same device.

GPUs help neural networks because both 3D graphics and model training spend their time doing the same thing: lots of parallel multiplications and additions.



How a Network Learns: The Big Picture

How a network learns: a feedback loop, not a magic formula



Three new words: a **forward pass** is step 1 (the prediction); **loss** is the number from step 2; a **backward pass** happens inside step 3, where PyTorch figures out how to adjust each weight.

Gradient: The Direction of "Less Wrong"



Gradient points uphill — the optimizer takes a small step in the opposite direction



Plain-language gradient

- Pick one weight in the network. The **gradient** for that weight is: "if I nudge this weight by a tiny bit, how much does the loss change?"
- The gradient points **uphill** (toward bigger loss).
- Training takes a small step in the **opposite** direction (toward smaller loss).
- You do not need calculus to use this idea — but PyTorch will use calculus for you.

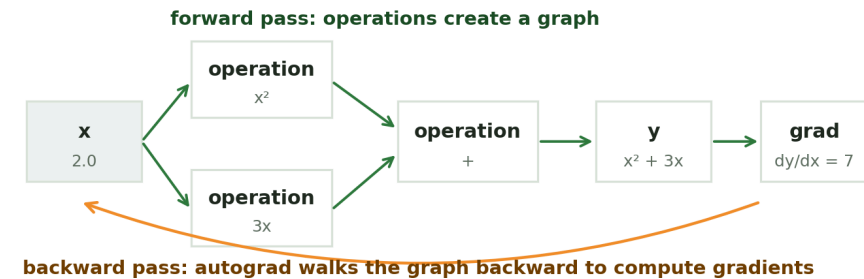
Autograd: PyTorch Records What You Did

Smallest possible example

```
x = torch.tensor(2.0, requires_grad=True)
y = x ** 2 + 3 * x
y.backward()
print(x.grad)  # 7.0
```

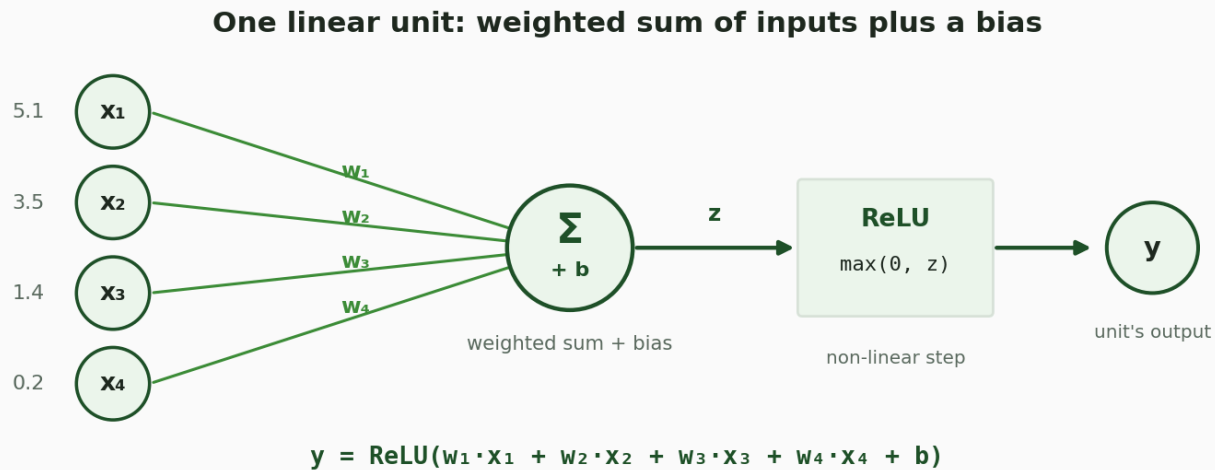
- `requires_grad=True`: track operations that touch this tensor.
- `y.backward()`: walk the recorded operations (the **computation graph**) backward to compute the gradient.
- `x.grad` now holds dy/dx at $x = 2$.

Autograd remembers operations so gradients can be computed backward



Key idea: PyTorch does not guess the derivative — it remembers the exact operations you ran and walks them backward. The same trick scales from this two-term equation to a real network with millions of weights.

Building Block: One Linear Unit



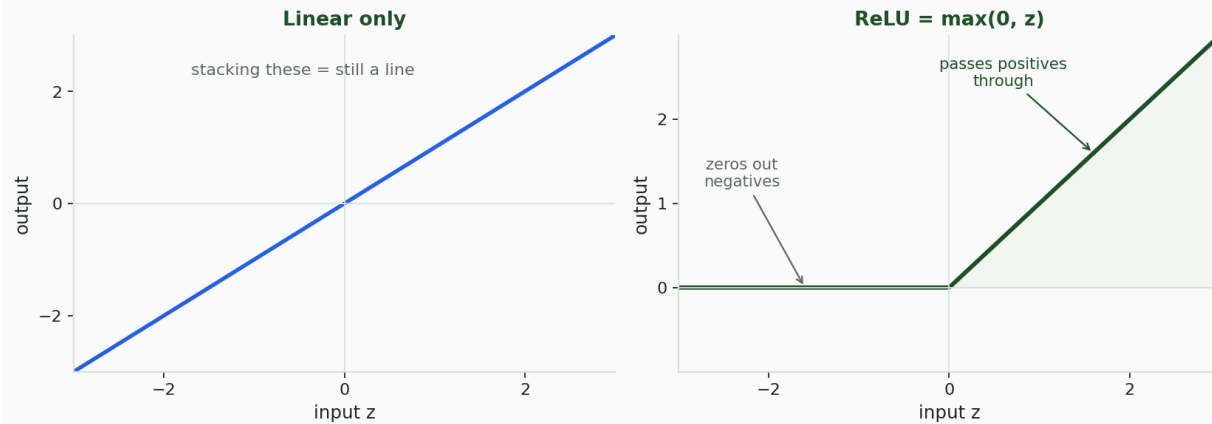
What `nn.Linear` does

- Multiply each input by its **weight**, add them up, add a **bias**: $z = w \cdot x + b$.
- A real layer does this for many units in parallel; `nn.Linear(4, 16)` has 16 such units, each with its own weights and bias.
- Then we apply an **activation function** (next slide) to introduce a non-linear bend.

Activation Function: Why a Bend Is Needed



ReLU adds the bend a linear-only stack would never have



Why we need a non-linear step

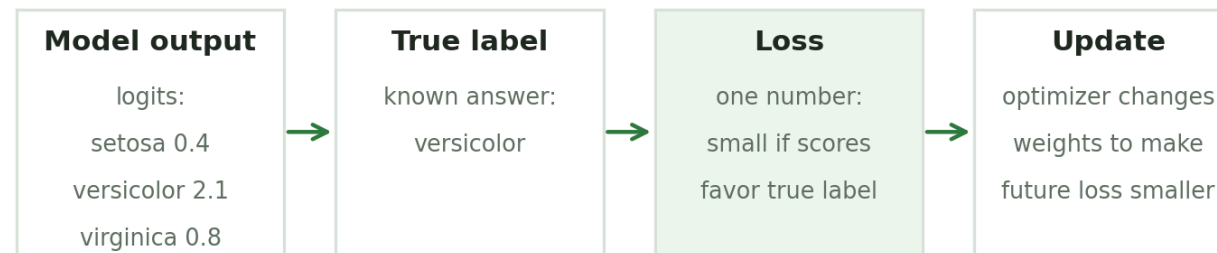
- If every layer is just a weighted sum, stacking them is still a single weighted sum — no extra power.
- An **activation function** is a tiny non-linear step applied between layers.
- $\text{ReLU}(z) = \max(0, z)$: keep positive values, zero out negatives. Simple and fast.

New term: logits are the raw scores produced by the very last Linear layer — for Iris that is one number per species.



Loss: The Number Training Tries to Reduce

Loss tells the optimizer what to improve



Loss is the training signal: it converts “how wrong was the prediction?” into a number PyTorch can minimize.

For Iris: `CrossEntropyLoss`

- Designed for multi-class classification.
- Takes raw **logits** directly — no extra conversion needed in your code.
- Returns one number per batch: smaller is better.

Loss vs accuracy: accuracy answers "how often was I right?" but is hard to optimize directly. Loss is a smooth signal PyTorch can use to nudge weights.



Optimizer: Turning Gradients Into Updates

One update step in code

```
optimizer.zero_grad()      # 1. forget previous gradients
logits = model(batch_X)    # 2. forward pass
loss = criterion(logits, y) # 3. compute loss
loss.backward()            # 4. backward pass (autograd)
optimizer.step()           # 5. apply the update
```

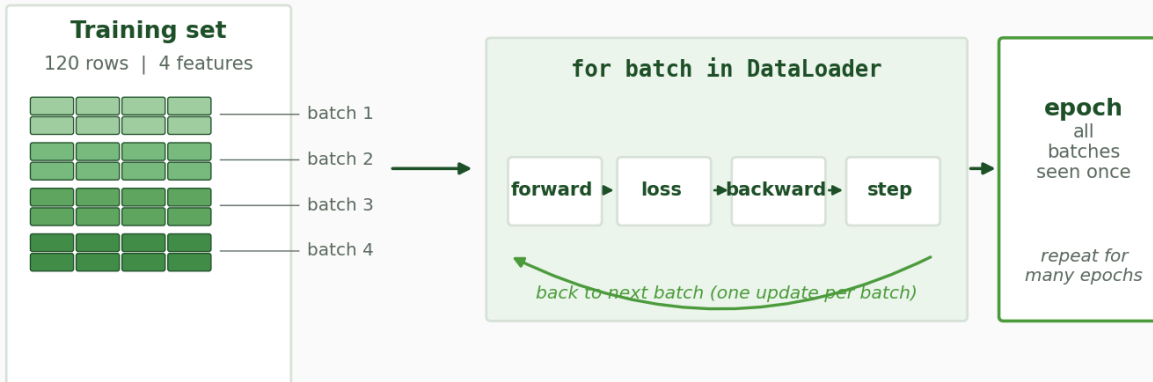
What the optimizer is

- A small Python object that knows how to take gradients and adjust the weights.
- For Iris we use `torch.optim.AdamW`, a robust default for small networks.
- The **learning rate** (e.g. `lr=1e-3`) is the step size — too big overshoots, too small crawls.

Why `zero_grad()` ? Gradients accumulate by default; reset them before each backward pass so old steps do not contaminate the new one.

Batches and Epochs

Batch = a small group of rows. Epoch = one full pass through the training data.



New vocabulary, no surprises

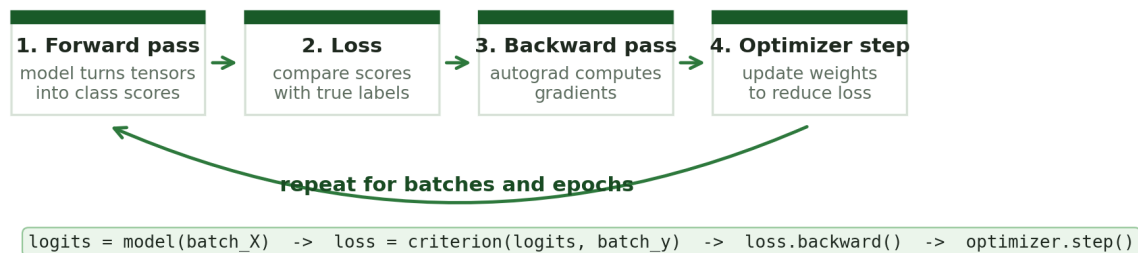
- **Batch:** a small group of rows the model sees together — one update per batch.
- **Epoch:** one full pass through the training data.
- Many epochs of small batches = many small weight updates = a useful network.

How PyTorch hands you batches

```
loader = DataLoader(  
    TensorDataset(X_train, y_train),  
    batch_size=16, shuffle=True,  
)
```



Putting It Together: The Training Loop



The same loop in code

```
for epoch in range(epochs):
    model.train()
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        logits = model(batch_X)
        loss = criterion(logits, batch_y)
        loss.backward()
        optimizer.step()
```

Every term in this loop has been introduced on a previous slide.



Train, Eval, and Inference Modes

Different code paths for different jobs

Situation	Code	Why it matters
Training	<code>model.train()</code>	Use while updating weights from training batches
Evaluation	<code>model.eval()</code>	Use while measuring validation or test performance
Pure inference	<code>with torch.inference_mode():</code>	Removes autograd overhead when you only want predictions

Some layers (e.g. dropout, batch norm — used in bigger networks) behave differently during training vs evaluation. `model.eval()` tells them which mode they are in. Always switch before running test predictions.

Iris with PyTorch — End-to-End

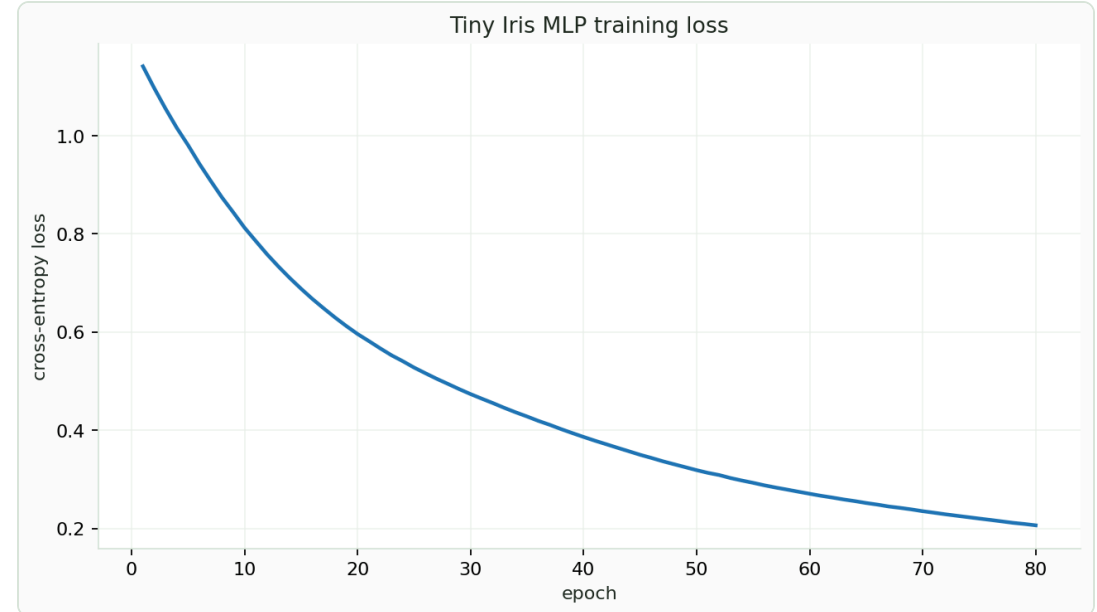
The full mini-script

```
model = nn.Sequential(
    nn.Linear(4, 16),
    nn.ReLU(),
    nn.Linear(16, 3),
).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(
    model.parameters(), lr=1e-3,
)

# loop over epochs and batches
# (see previous slide)

model.eval()
with torch.inference_mode():
    pred = model(X_test).argmax(dim=1)
```



Loss drops steadily across 80 epochs; test accuracy reaches roughly 0.97. The same machinery scales to a real image or text model.



Where PyTorch Really Shines (and Iris Does Not)

Honest take on Iris

- On Iris, scikit-learn's KNN already reaches near-perfect accuracy.
- PyTorch is not the right tool to "win" on this dataset — we used it to make every concept inspectable.
- Same loop, same vocabulary, much bigger problems work the same way.

What PyTorch unlocks beyond Iris

- **Vision:** convolutional networks for image classification, segmentation, detection.
- **Language:** transformers, classification and translation models.
- **Audio & signals:** speech, music, time series.
- **Custom losses and architectures:** anything you can write in Python.

You now know the same training loop these much bigger systems run:
forward → loss → backward → step.



Checkpoint: PyTorch Foundations

Q7

What does `requires_grad=True` tell PyTorch about a tensor?

- A) Track every operation involving this tensor so gradients can be computed later
- B) Move the tensor to the GPU automatically
- C) Convert the tensor to a Python list of floats
- D) Print its current gradient on every operation

Show Answer

Q8

Why does a neural network need an activation function like `ReLU` between linear layers?

- A) To make gradients larger so training is faster
- B) Stacking only linear layers collapses to one linear layer; the non-linear bend is what gives the network real expressiveness
- C) To make the loss negative
- D) Because PyTorch refuses to compute gradients without one

Show Answer

Q9

On Iris, scikit-learn already wins. Why introduce PyTorch here?

- A) PyTorch always beats scikit-learn on accuracy
- B) To learn a training loop that scales to images, text, and audio
- C) Because scikit-learn models stop working in 2026
- D) To make Euclidean distance computations faster

Show Answer



Lecture Summary

Honest ML

- Represent data as `X` and `y`
- Split before fitting
- Use pipelines and cross-validation

Vision Mindset

- Images are arrays first
- Filters extract structure numerically
- Pre-trained weights are reusable engineering assets

PyTorch Basics

- A neural network = layers of weighted units, trained by a loop
- Tensors + autograd + GPUs are PyTorch's three superpowers
- Forward → loss → backward → step, repeated over batches and epochs

One unifying idea: the useful abstraction is not "AI magic." It is representation -> model -> evaluation -> handoff.



Thank You for Your Attention!

Inspect the representation, protect the evaluation, and keep the workflow reproducible.

predict

inspect

measure

iterate



Python Course 2025/26/2

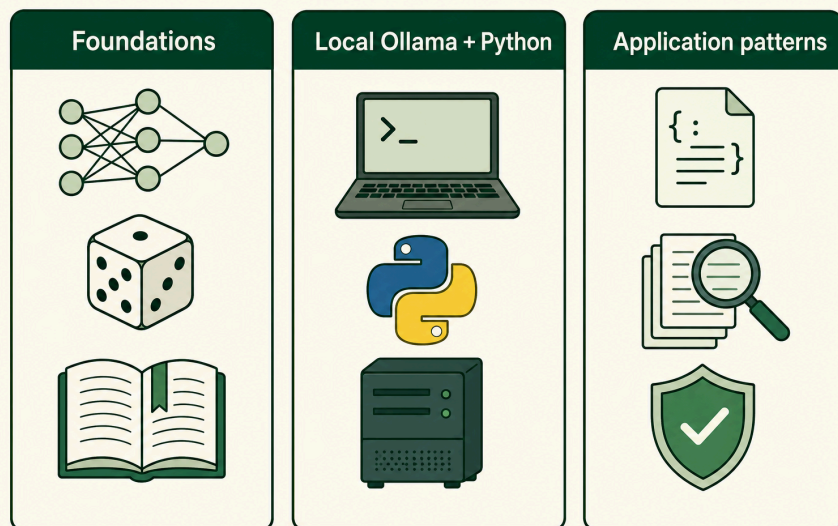
Lecture 12: Applied Python for Generative AI & Local Inference



Course coordinator: Dr Mate Tejfel

Lecturer: Zoltan Kegli

Lecture Overview



One story in three layers: ideas, then a working local call path, then patterns for real programs.

Learning objective

Connect the Lecture 9–11 to **generative** models you run locally: mechanism, Python call path, evaluation, retrieval, and safety.

By the end, you can

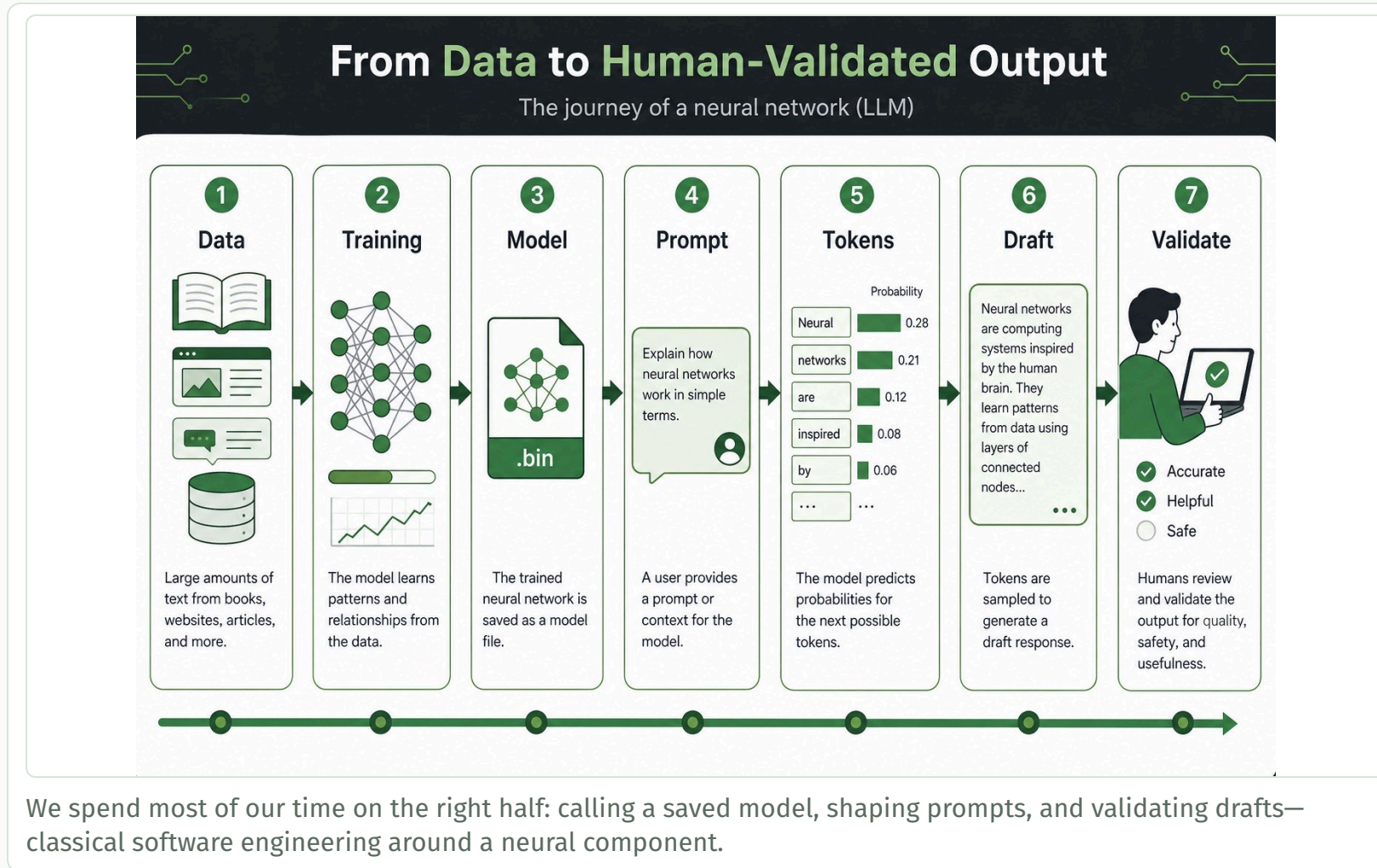
- **Foundations** — explain tokens, logits, attention, sampling, context windows, and quantization.
- **Local stack** — run `ollama.chat`, inspect drafts, store message history, and stream tokens.
- **Applications** — validate JSON with Pydantic, evaluate drafts, use RAG/embeddings, and gate tools with guardrails.



Foundations of generative AI

Vocabulary, next-token prediction, attention intuition, sampling, and the hardware bridge before we run
Ollama

Roadmap: training vs inference today





From Classification to Generation

Five words to carry over from Lecture 11

- **tensor** — an n-dimensional array of numbers (scalar → vector → matrix → tensor).
- **forward pass** — run the network once: input tensor → output tensor, no weight updates.
- **logit** — a raw unnormalized score the network emits (can be any real number).
- **softmax** — turns a vector of logits into probabilities that sum to 1.
- **argmax** — returns the index of the largest entry in a vector.

Classifier (Lecture 11)

- Input: features (pixels, measurements, embeddings of an image) packed into a tensor.
- One forward pass → logits, **one per class** (e.g. 10 for MNIST digits).
- Apply softmax to read probabilities, or take **argmax** for the best class.

Generative text model (today)

- Input: the **tokens** seen so far (the running context) as an integer tensor.
- One forward pass → logits, **one per token in the vocabulary** (typically 30k–200k entries).
- Apply softmax, pick one next token (sample or argmax), append to the context, repeat.

New idea first: a **vocabulary** is the full list of tokens the model can emit. A **token** is one item from that list (often a subword, not always a whole word).

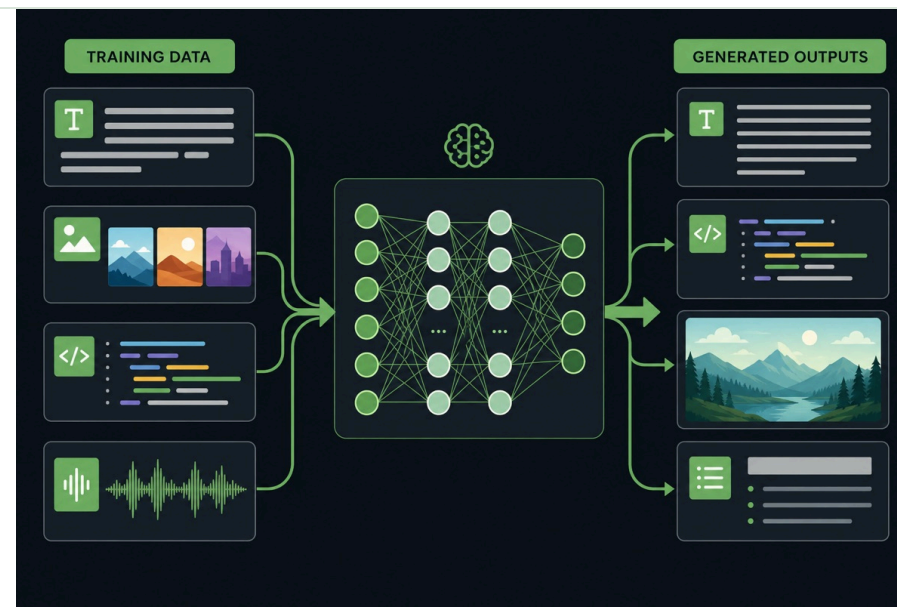


Vocabulary Words: AI, Model, Generative AI

Definitions in teaching order

- **Artificial intelligence (AI):** software that automates tasks we associate with human judgment (pattern recognition, language, planning).
- **Model:** the saved result of training: weights (the learned numbers from Lecture 11) + architecture + tokenizer rules + templates.
- **Generative AI:** systems that **produce new examples** (text, code, images, audio, video), not only pick a label.

A **modality** is a kind of data (text, image, audio). A **multimodal** model accepts more than one modality in the same interface.



Training teaches patterns; generation applies those patterns to new inputs you provide at inference time.



Foundation Models and the Training Loop

1. Collect data

Large text/code corpora plus other modalities when training multimodal systems.

2. Hide the next token

Self-supervised objective (labels come from the data itself — no humans label each example): hide a token, predict it from the context. Wrong answer → nudge the weights.

3. Ship a checkpoint

A **foundation model** is a broad base many products fine-tune, prompt, or retrieve against.

Link to Lecture 11

Same ingredients — forward pass, loss, optimizer — but the target is a full probability vector over the vocabulary, not one softmax row over classes.

Tokenization: Text Becomes Integer IDs

What tokenization is

Tokenization is a deterministic program that maps text to a list of **token IDs** (integers indexing the vocabulary). It runs before any neural math.

Text

"unbelievable"

Pieces

un

believ

able

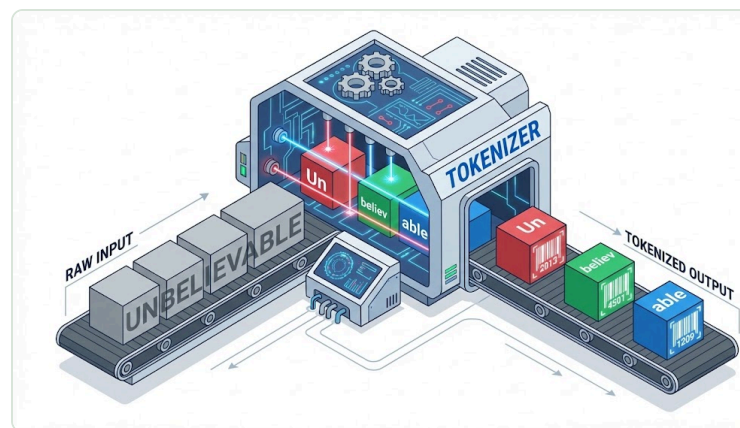
IDs

[403, 18124,
481]
(illustrative)

Tensor

`torch.LongTensor`
in PyTorch

Tensor recall: an n-dimensional array. A single prompt becomes a 1-D integer tensor of shape `(seq_len,)`; batched requests add a leading dimension $\rightarrow$ `(batch, seq_len)`. Same data type (integers) as a lookup table index.



Different models ship different tokenizers. Same Python string $\rightarrow$ different token counts $\rightarrow$ different latency and cost.

Logits, Softmax, One Probability Vector

From scores to probabilities

- **Logits:** the network's raw unnormalized scores, one per vocabulary entry. Can be negative, any magnitude.
- **Softmax:** exp-then-normalize (logits $\mathbf{x}$ are one score per vocabulary entry).

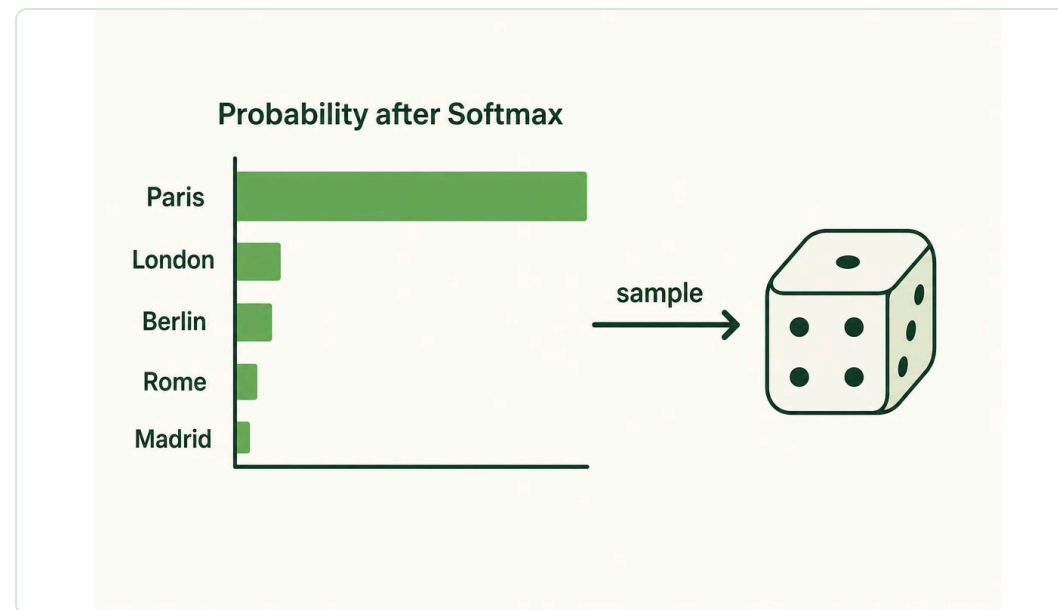
$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

`exp` makes every entry positive; dividing by the sum makes them add up to 1 → a valid probability distribution.

- **Argmax:** “arg max” = argument of the max. Returns the index of the largest entry — the greedy pick.
- **Autoregressive:** auto (self) + regressive (predicting from past values). Each new token is conditioned on every token chosen so far, including the ones the model just emitted.

Softmax is bookkeeping, not truth. A peaked distribution can still be confidently wrong.

Illustrative next-token mass



Toy view: real models expose tens of thousands of logits; demos plot only the top few for readability.



Generation Loop and Prompt

Prompt + loop

A **prompt** is the user or developer text (plus system instructions) that seeds the context.

1. Tokenize the prompt.
2. Forward pass → probabilities for the next token.
3. Sample or take argmax.
4. Append token; stop when a stop token or length budget hits.

```
context = tokenize("Write a haiku about Python")

while not done(context):
    logits = model.forward(context)
    probs = softmax(logits)
    next_id = sample(probs) # or argmax(probs): always pick the top (greedy)
    context.append(next_id)
```

Pseudocode for intuition. Production stacks add KV-cache, batching, streaming, and safety filters.

Why errors compound

- Each chosen token becomes part of the next context.
- An early mistake steers future logits even if later steps look confident.
- This is why retrieval, tools, tests, and human review matter.

Stop token: a reserved token or API flag meaning "finished".



Transformer Intuition: Attention and Cache

Attention

For each next token, the model compares the current position with earlier tokens and decides which parts of the context are useful. It is learned lookup-and-mixing over vectors, not symbolic memory.

Context engineering

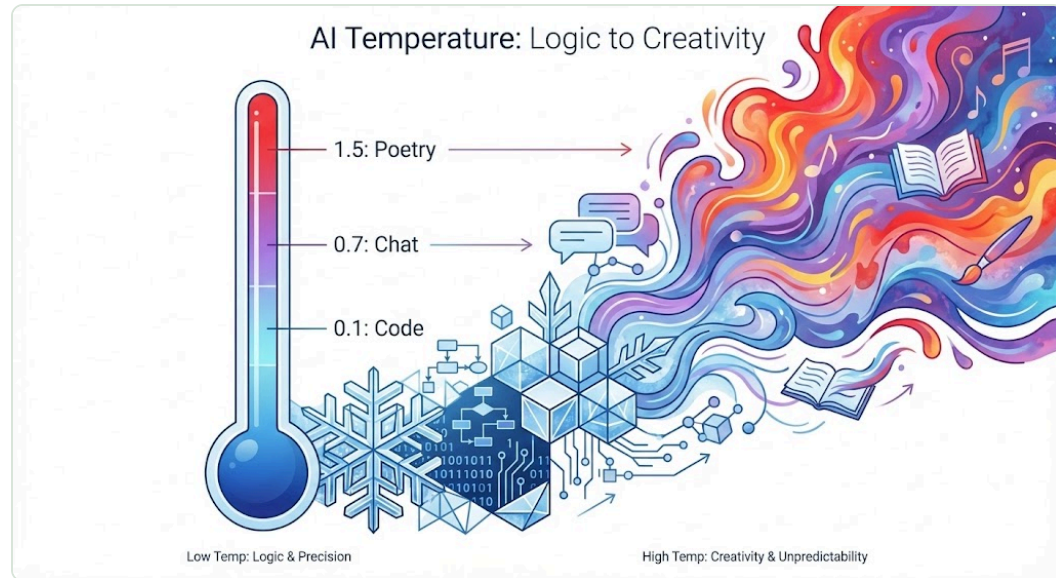
The model can only attend to tokens inside the request. Your Python code decides what history, retrieved chunks, examples, and instructions fit into that limited window.

KV cache

Runtimes cache internal key/value vectors for previous tokens, so generation can reuse prior work. Long prompts still cost time during **prefill**.

Good prompts are not magic wording. They are careful context construction: the right task, facts, examples, constraints, and output contract in the smallest useful token budget.

Sampling Knobs: Temperature, Top-k, Top-p



After softmax you may take the argmax (greedy) or draw a random sample proportional to probability.

Three controls

- **Temperature T:** divide logits by T before softmax $\rightarrow \text{softmax}(\text{logits} / T)$. Low T sharpens the distribution ($T \rightarrow 0$ is basically argmax); high T flattens it, so less likely tokens stand a chance.
- **Top-k:** keep only the k highest-probability tokens, renormalize, sample. Simple, ignores the long tail.
- **Top-p (nucleus):** keep the smallest set whose cumulative mass $\geq p$, then sample. Adapts to the shape of the distribution — wider when the model is uncertain, narrower when it is confident.

Task	Typical randomness
JSON / extraction	greedy or very low temperature
Code assist	low to medium + top-p
Brainstorming	medium + nucleus sampling



Modalities and Where Python Sits

What can be generated?

Modality	Example outputs
Text	answers, summaries, translations
Code	functions, tests, refactors
Image / audio / video	edits, narration, storyboards

Python is the control plane

Models propose tokens; your code chooses models, trims context, calls tools, enforces permissions, logs prompts, and validates outputs with pytest or data checks.

Multimodal inputs (screenshots, PDFs) explode token counts—budget before you attach images to every request.



One Mental Model: Represent, Predict, Control

Represent

Tokenize text, normalize tables, resize images—turn raw inputs into tensors the network can consume.

Predict

Run the trained deep network to map context to logits, apply softmax, then sample or take the argmax.

Control

Prompt design, retrieval, structured outputs, monitoring, least-privilege tools, humans in the loop.

Generative AI is applied data science: data hygiene, modeling, evaluation, and software engineering in one loop.

Bridge to the Local Stack: Memory, Quantization, Models

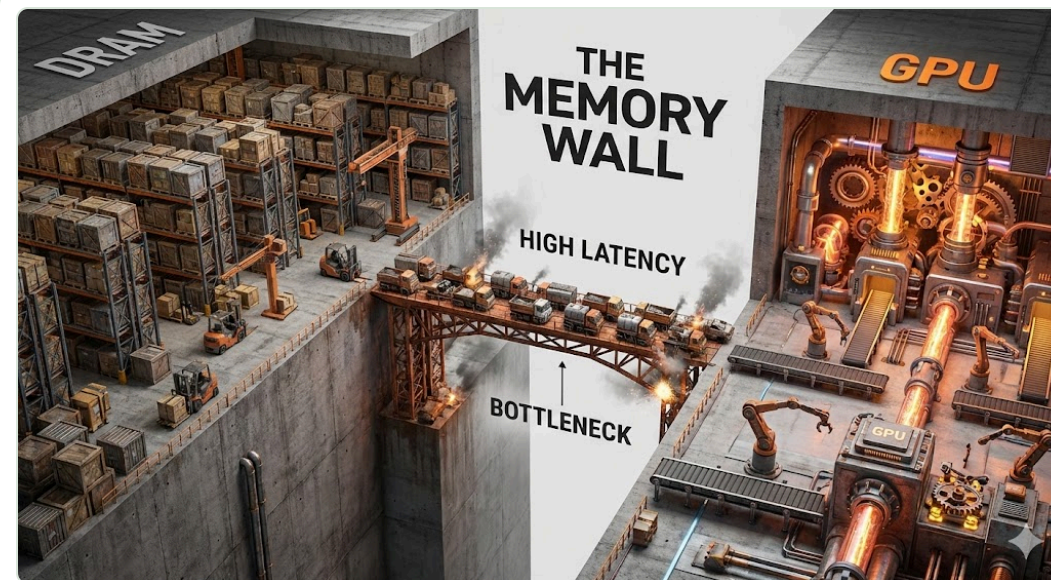
Why laptops care about bandwidth

`tokens_per_sec` $\approx$ `memory_bandwidth` $\div$

`bytes_per_weight_pass`

- Each new token touches every weight once in a naive matmul loop.
- Quantization packs more weights per gigabyte transferred $\rightarrow$ higher token throughput.

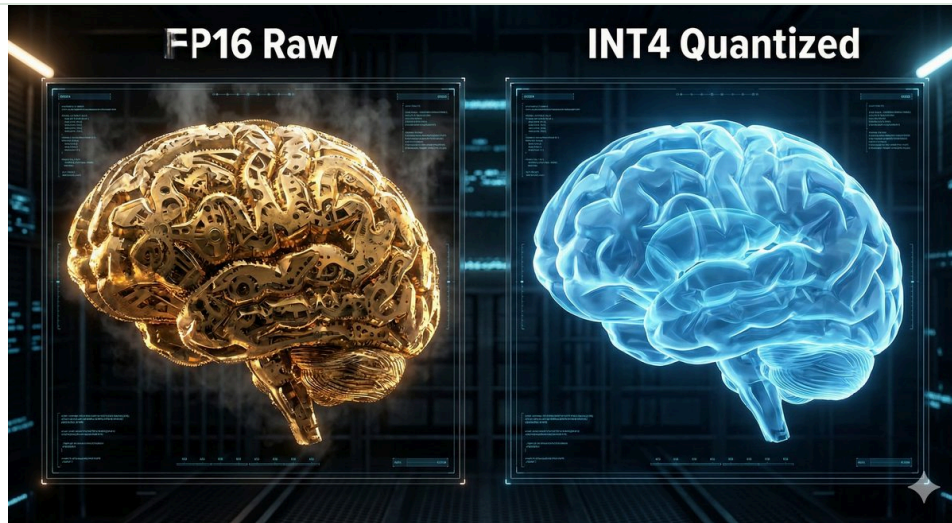
Section 2 shows the same tags running under Ollama from Python (`gemma4:e4b`, `embeddinggemma`).



If weights cannot reach the ALU fast enough, FLOPs are irrelevant.

Use case	Model tag	Approx size	Note
Validated lecture default	<code>gemma4:e4b</code>	9.6 GB	Pair with <code>embeddinggemma</code> for local RAG demos
Small text compare	<code>qwen3:4b</code>	2.5 GB	Great for experiments; chat templates may be noisy on some builds
Lab workstation	<code>qwen3:30b</code>	19 GB	Strong reasoning when VRAM/RAM allows

Quantization, GGUF, and Perplexity (PPL)



Quantization stores weights with fewer bits per number. Same chip, smaller payloads across the memory bus.

GGUF in one sentence

A single on-disk bundle holds quantized weights, lookup tables, tokenizer tables, and chat templates. Runtimes can **memory-map** (`mmap`) slices so cold pages never touch RAM (GPT-Generated Unified Format).

Perplexity (PPL)

$$\text{PPL} = \exp \left(-\frac{1}{N} \sum_{i=1}^N \log P(x_i^* \mid x_{<i}) \right)$$

- Measures average surprise on real text; **lower is better**.
- Useful for comparing quant formats; pair with task evals (coding, retrieval) before shipping.



Checkpoint: Foundations

Q1

What makes an AI system “generative”?

- A) It only chooses one label from a fixed list
- B) It creates a new draft such as text, code, image, or audio
- C) It stores every training example in a database table
- D) It never needs human review

Show Answer

Q2

In an autoregressive language model, what happens after one next token is selected?

- A) The token is appended to the context and influences the next prediction
- B) The model changes its weights permanently
- C) The tokenizer is discarded for the rest of the answer
- D) The model switches from probabilities to a database lookup

Show Answer



Checkpoint: Foundations (continued)

Q3

Why does a long prompt often increase time-to-first-token even when generation is streamed?

- A) The model must prefill/cache the prompt tokens before emitting the first new token
- B) Streaming retrains the tokenizer before every answer
- C) The context window becomes unlimited
- D) Top-p sampling disables the forward pass

Show Answer

Q4

Why does weight quantization usually increase tokens per second on the same laptop?

- A) Fewer bits per weight means the memory subsystem delivers more useful parameters per second to the math units
- B) Quantization trains the model again from scratch
- C) It disables the tokenizer to save time
- D) It replaces Python with assembly everywhere

Show Answer



Local stack: Ollama from Python

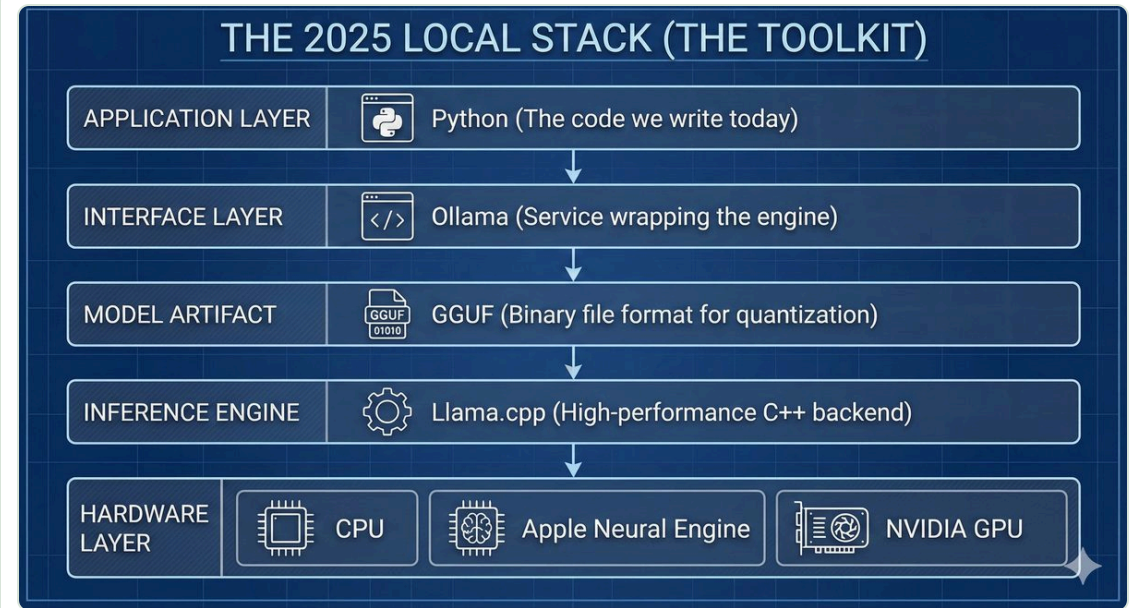
First local call, useful drafts, conversation state in your program, streaming, and time-to-first-token

What Ollama is (before any code)

Definition

Ollama is a program that runs on your computer, downloads **open-weight** models (weights you can download — license still matters), and serves them over local HTTP. The `ollama` Python package is a small client that sends requests to that server.

- You already saw **GGUF** in the foundations section: one file bundles quantized weights plus tokenizer metadata Ollama needs.
- **Inference** here means “ask the loaded model for the next tokens”; no training happens in these demos.



Python orchestrates prompts and business logic; Ollama concentrates on efficient token generation.



First call: `ollama.chat`

```
import ollama

response = ollama.chat(
    model="gemma4:e4b",
    messages=[
        {"role": "system", "content": "You are a careful Python tutor."},
        {"role": "user", "content": "Define a token in one short sentence."},
    ],
    options={"temperature": 0.3, "num_ctx": 4096},
)

print(response.message.content)
```

Aligned with `src/02_ollama_basics_and_state.py`: same defaults and metric helpers live in `demo_utils`.

Read the response object

- **Role:** each message is tagged `system`, `user`, or `assistant` so the runtime can apply the model's chat template.
- **Model name string:** the tag must match a model you pulled with `ollama pull` (for example `gemma4:e4b`).
- **options:** classroom knobs include `temperature` and `num_ctx` (how many tokens of context the runtime may use).



Strength Demo: Natural Language → Python Draft

Prompt

"Write a Python function that groups transactions by month."

Useful first draft

```
from collections import defaultdict

def group_by_month(rows):
    groups = defaultdict(list)
    for row in rows:
        groups[row["date"][:7]].append(row)
    return dict(groups)
```

Why this belongs after the first call

- Students have now seen the exact Python API that produces the draft.
- The output is useful because a draft saves time, not because it is automatically correct.
- Follow-up prompts can explain, refactor, or adapt it to pandas, SQL, or tests.

Engineering habit

Always pair generation with checks: empty input, malformed dates, time zones, huge files.



Statelessness: memory lives in Python

```
messages = [  
    {"role": "system", "content": "You are a careful tutor."},  
    {"role": "user", "content": "My name is Bela."},  
    {"role": "assistant", "content": "Nice to meet you, Bela."},  
    {"role": "user", "content": "What is my name?"},  
]  
  
ollama.chat(model="gemma4:e4b", messages=messages)
```

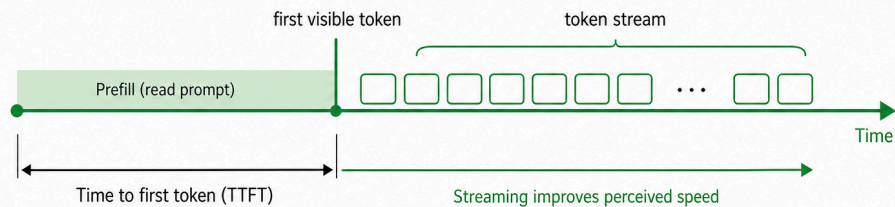
If you omit earlier turns, the model cannot see them — there is no hidden notebook inside the weights for your chat.

Terms in the list literal

- **System message:** developer-facing instructions (style, safety, output format).
- **User / assistant turns:** mimic a conversation transcript.
- **Conversation state:** the growing `messages` list you store, trim, summarize, or augment — classic Python data structures, not magic model memory.

Demo 2 in the notebook proves forgetting: change the list, watch the answer change.

Streaming tokens and time-to-first-token (TTFT)



Streaming forwards partial output while generation continues; users see progress sooner.

Define the metrics

- **Time to first token (TTFT):** seconds from sending the request until the first piece of generated text arrives.
- **Prefill:** the work the runtime does to ingest your prompt (and prior context) before it can emit the first newly generated token; large prefill often dominates TTFT.
- **Throughput:** tokens per second after streaming starts — still important, but TTFT drives perceived snappiness in chat UIs.

```
stream = ollama.chat(  
    model="gemma4:e4b",  
    messages=[{"role": "user", "content": "Stream a haiku about pandas."}],  
    stream=True,  
)  
  
for chunk in stream:  
    if piece := chunk.message.content:  
        print(piece, end="", flush=True)
```

Async AsyncClient and FastAPI StreamingResponse build the same idea for web APIs; see `src/03_streaming_and_fastapi.py`.



Validated demo defaults (your machine may vary)

Always match code, slides, and the README: students should **pull** the same tags before class.

Role	Model tag	Notes
Chat default	<code>gemma4:e4b</code>	Pairs with lecture checkpoints; balanced quality on modest GPUs/CPUs.
Embeddings for RAG demos	<code>embeddinggemma</code>	Vectors for semantic search examples in section 3.
Small alternative	<code>qwen3:4b</code>	Great for experiments; chat templates differ by release.



Checkpoint: Local Ollama stack

Q5

Where does multi-turn chat memory live unless you code it yourself?

- A) Inside the model weights forever
- B) In a hidden Ollama database that remembers every user
- C) In the `messages` list your Python program passes to each call
- D) Only in the cloud provider's logs

Show Answer

Q6

Why do UIs enable streaming instead of always waiting for the entire answer string?

- A) Streaming guarantees factual correctness
- B) Streaming lowers TTFT so users see the first tokens earlier
- C) Streaming removes the need for a tokenizer
- D) Streaming trains the model weights during the request

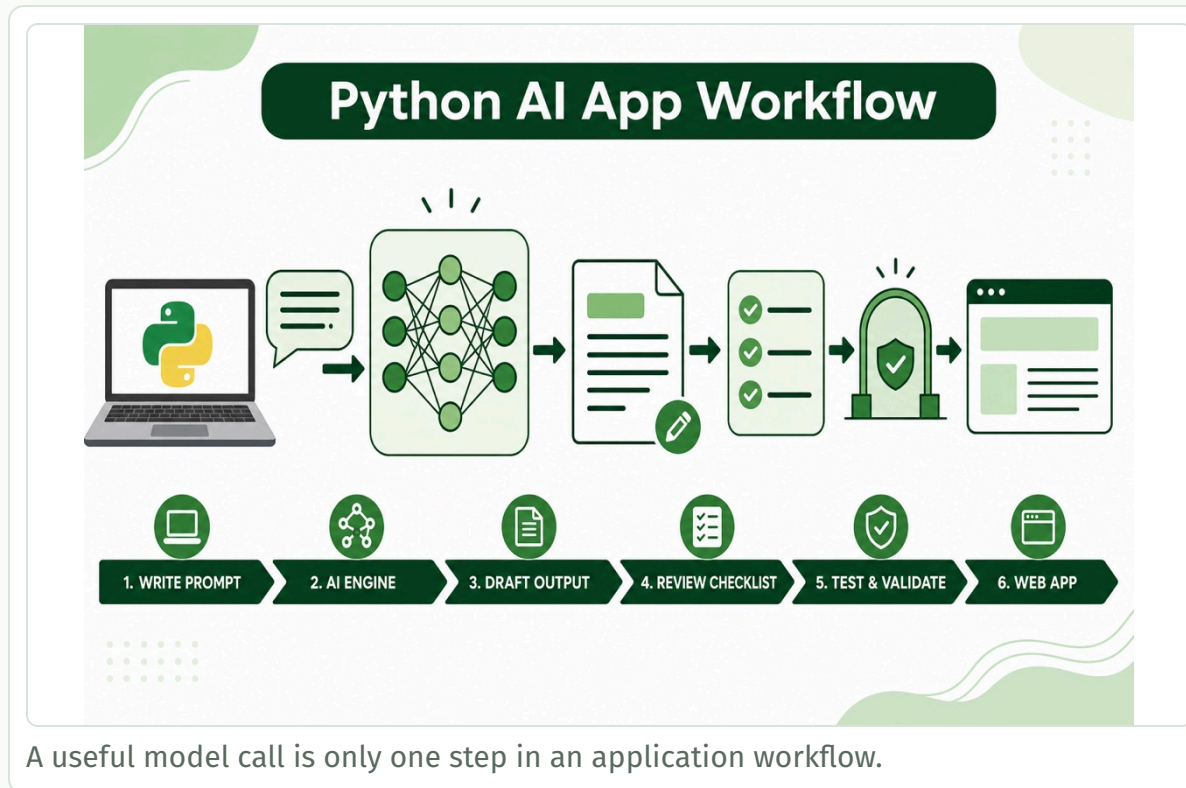
Show Answer



Application patterns

Use cases, evaluation, structured outputs, grounding, embeddings, RAG, tools, agents, guardrails

The Python App Workflow: Ask, Draft, Check



Define the words before using them

- **Prompt:** the instruction and input text sent to the model.
- **Context:** everything the model sees in one call: prompt, history, documents, and system instructions.
- **Inference:** running the already-trained model to produce output; no weights are changed.
- **Draft:** a candidate answer produced by the model, not a verified fact or finished program.

Core habit: generative AI is strongest when a checked draft is useful. Python should still validate, test, and decide what is allowed.



Where GenAI Helps — and Where It Fails

Strong fits

- Draft code, tests, docs, emails, and explanations.
- Transform text between formats when mistakes are cheap to catch.
- Search, summarize, and tutor over trusted material.

Model risks

- **Hallucination:** fluent but false or unsupported claims.
- **Bias:** skewed outputs from data, prompts, or evaluation gaps.
- **Stale knowledge:** APIs and facts move faster than weights.

System risks

- Latency spikes, token budgets, and denial-of-wallet failures.
- Privacy leaks through prompts, logs, or generated HTML.
- Non-determinism when sampling is on, even with the same prompt.

Ship only where a checked draft is useful. If the workflow cannot detect a bad answer, the model is in the wrong place.



Evaluation: Make Quality Measurable

CS habit: tests before trust

- **Golden set:** small representative inputs with expected behavior.
- **Unit checks:** schema validity, no missing fields, no dangerous sinks.
- **Regression run:** rerun the same cases after model, prompt, or retrieval changes.
- **Human review:** sample hard cases where correctness is semantic.

Measure by task

Task	Useful checks
Extraction	JSON schema + field accuracy
Code draft	pytest + review checklist
RAG answer	context relevance, groundedness, answer relevance
Tool call	authorization + dry-run + audit log

Modern GenAI engineering is less "find the perfect prompt" and more "keep a repeatable evaluation harness beside every prompt, schema, retriever, and tool."



Structured Output: Define the Shape Before You Ask

```
from pydantic import BaseModel, Field
from ollama import chat

class StudyCard(BaseModel):
    term: str = Field(..., min_length=1)
    definition: str = Field(..., min_length=20)
    python_example: str = Field(..., min_length=10)

response = chat(
    model="gemma4:e4b",
    messages=[{"role": "user", "content": "Create a study card for 'token'."}],
    format=StudyCard.model_json_schema(),
    options={"temperature": 0},
)

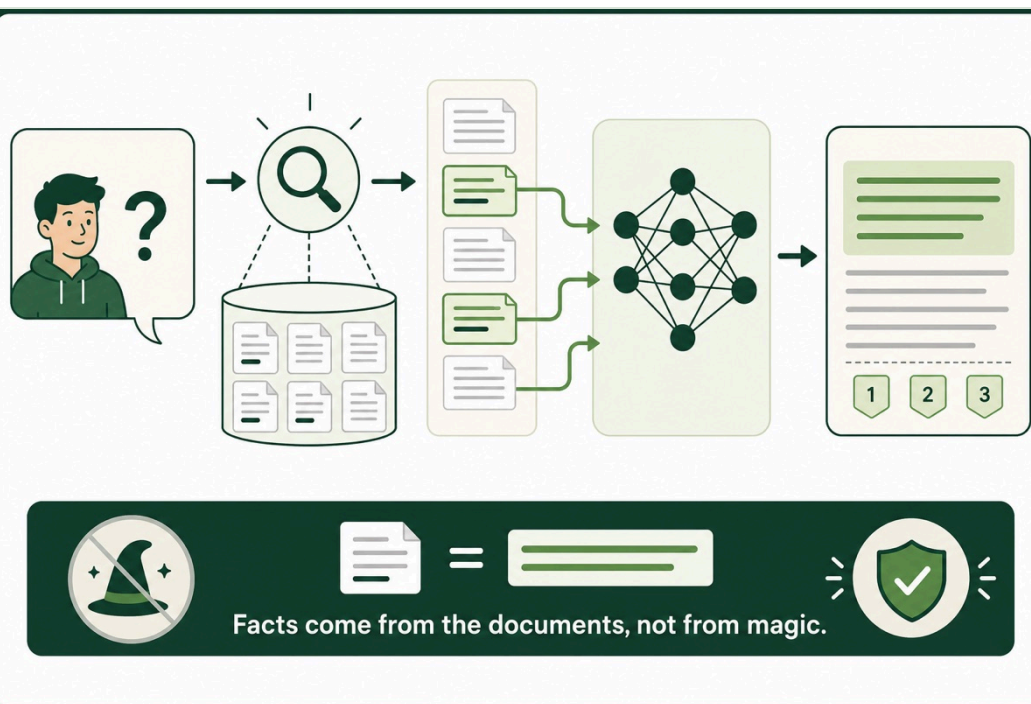
card = StudyCard.model_validate_json(response.message.content)
print(card.term, "⇒", card.definition)
```

Definitions

- **JSON:** a text format for structured data: objects, lists, strings, numbers, booleans.
- **Schema:** a formal description of allowed fields, types, and constraints.
- **Pydantic model:** a Python class that validates data against such a shape.

Structured output makes model output easier to connect to Python functions, databases, and APIs. It reduces formatting mistakes, but validation is still required. For best reliability, pass the JSON schema to `format`, set a low temperature, and ask for JSON in the prompt too.

Grounding: Give the Model the Facts



Facts should come from selected documents, not from a lucky guess.

Build the idea step by step

- **Grounding:** attaching the answer to provided facts.
- **Retrieval:** searching a knowledge source before asking the model to answer.
- **Chunk:** a small piece of a document that can fit into context.
- **Citation:** an identifier showing which chunk supported the answer.
- **RAG:** Retrieval Augmented Generation: retrieve facts, add them to the prompt, then generate.

Use RAG when the answer depends on private, changing, or course-specific documents.



Embeddings: Searching by Meaning

```
import math
import ollama

docs = [
    {"id": "returns", "text": "Opened items can be returned with a fee."},
    {"id": "shipping", "text": "Standard delivery takes 3 to 5 days."},
]

def embed(texts):
    return ollama.embed(model="embeddinggemma", input=texts)["embeddings"]

def dot(a, b):
    return math.fsum(x * y for x, y in zip(a, b))

doc_vectors = embed([doc["text"] for doc in docs])
query_vector = embed("Can I get my money back after opening the box?")[0]

best_score, best_doc = max(
    (dot(query_vector, vec), doc)
    for doc, vec in zip(docs, doc_vectors)
)
print(best_doc["id"], round(best_score, 3))
```

Definitions

- **Embedding:** a list of numbers representing the meaning of a text for search.
- **Vector:** the math name for that list of numbers.
- **Similarity score:** a number saying how close two vectors are.
- **Cosine similarity:** a common score based on vector direction; Ollama embeddings are normalized, so dot product works as the score.

This lets "money back" match "return policy" even when the words are not identical.



RAG Quality: Retrieve, Ground, Answer

1. Retrieval quality

Did search fetch the right chunks? Improve with better chunking, metadata filters, hybrid keyword + vector search, or reranking.

2. Groundedness

Does every important claim come from the retrieved context? Ask for citations and reject unsupported claims in evaluation.

3. Answer quality

Does the response answer the actual question in the right tone and format? This is measured against user-visible examples.

Security-specific checks

- Filter retrieval by user permissions before the model sees chunks.
- Treat retrieved text as data, not instructions.
- Block or quarantine obvious prompt-injection snippets.

2026 default: evaluate the retrieval step separately from the answer step. A better model cannot fix missing, unauthorized, or poisoned context.



RAG vs Fine-Tuning: Which Lever Changed?

Fine-tuning means continuing training so the model weights change. **RAG** means leaving weights unchanged and changing the context supplied at inference time.

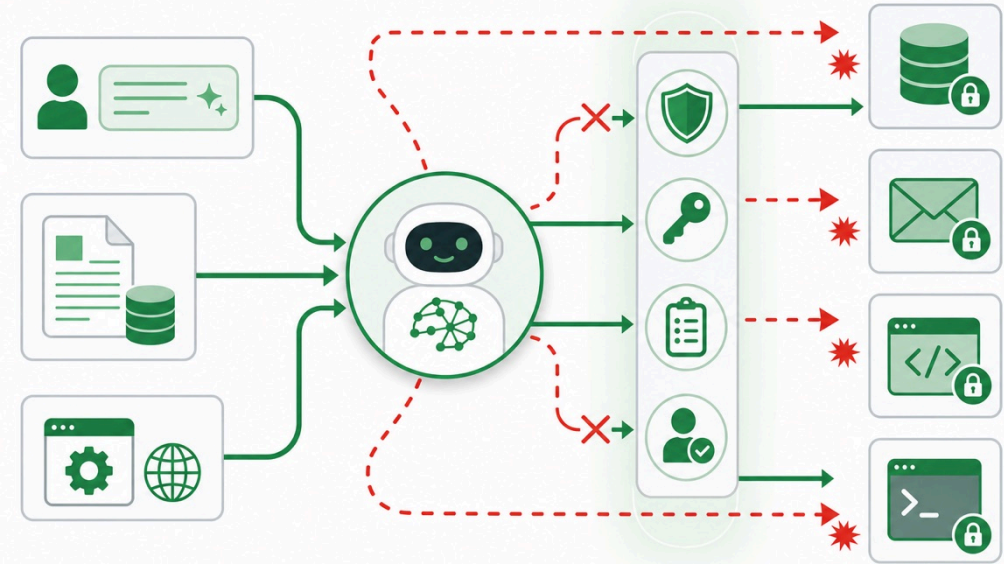
Question	Prefer RAG	Prefer fine-tuning
Need fresh facts?	Yes: update the document index.	No: weights go stale after training.
Need private course or company data?	Yes: keep access checks in the database or search layer.	Risky: data is baked into weights and hard to revoke.
Need a stable style or behavior?	Sometimes: prompt templates may be enough.	Often: examples can teach a repeated response habit.
Cost for a first prototype?	Lower: retrieval + normal model calls.	Higher: training data, GPUs, and evaluation.

Tools, Agents, and Guardrails

Definitions first

- **Tool:** a normal function or API the application can call, such as `search_docs()` or `send_email()`.
- **Agent:** a program loop where the model proposes the next step and the application decides whether to run it.
- **Guardrail:** an external check that blocks unsafe or invalid actions.
- **Authorization:** a permission check: is this user allowed to do this?

The model may suggest an action; Python enforces permissions, input validation, budgets, dry-runs, approval gates, and logging.



Security boundaries live outside the model response.

2026 Direction: Narrow, Evaluated, Governed



The mature pattern is a scoped workflow with metrics, not a chatbot that can do everything.

Proved production direction

- Small or specialized models when they meet the task; reserve huge models for hard cases.
- Structured outputs and tool calls with validation instead of free-text glue code.
- RAG with access control, citations, and retrieval evaluation for changing/private facts.

Agentic caution

- Plan-and-act loops need step-level evaluation, budget limits, and cancellation paths.
- Human approval remains the default for irreversible or high-impact actions.
- System prompts are guidance, not secrets and not authorization.



Security Checklist for Python GenAI Apps

Inputs

- **Prompt injection:** direct user text or indirect text inside retrieved documents tries to override instructions.
- Treat uploaded documents, web pages, chat messages, and tool outputs as untrusted data.
- Never place API keys, credentials, or private policy in a system prompt.

Actions

- Run authorization checks before every tool call.
- Use allow-lists for files, URLs, commands, models, and database operations.
- Set token, cost, and loop limits; ask for human approval before irreversible actions.

Outputs

- Validate JSON with Pydantic before using it.
- Escape generated text before displaying it in HTML.
- Require citations for grounded answers; reject unsupported claims in high-stakes flows.
- Log enough metadata to debug, but avoid storing private prompts unnecessarily.

Safe design assumption: the model is helpful but not trusted. Current OWASP-style risks include prompt injection, sensitive information disclosure, system prompt leakage, vector/embedding weaknesses, excessive agency, and unbounded consumption. Your Python application is the security boundary.



Putting It Together: A Responsible GenAI Loop

1. Ask

Write the prompt and define the task.

2. Retrieve

Add the relevant facts when facts matter.

3. Generate

Run inference and receive a draft.

4. Validate

Check schema, tests, permissions, and citations.

5. Use

Show, store, call a tool, or ask a human to approve.

Good classroom demo

- Small local model via Ollama.
- Short prompts students can read.
- Pydantic for structured answers.
- In-memory retrieval so the algorithm is visible.

Good production direction

- Persistent document index and access control.
- Automated evaluation on real examples.
- Monitoring for latency, cost, and failures.
- Clear policy for privacy and human review.



Checkpoint: Application patterns

Q7

Your model returns a plausible Python function. What should your application or workflow do before using it?

- A) Trust it because the model used many training examples
- B) Review it, run tests, and check edge cases before integration
- C) Increase temperature to make the code more confident
- D) Hide the generated code from the programmer

Show Answer

Q8

A course chatbot must answer from this semester's PDF handouts. Which design best matches that requirement?

- A) RAG: retrieve relevant chunks from the handouts, then generate with citations
- B) Ask the model to remember the files without sending them
- C) Put secret API keys into the prompt so it has more context
- D) Fine-tune first, even for a one-week prototype

Show Answer



Lecture Summary

1. Foundations

- Builds on Lecture 9–11: tensors, forward pass, softmax—but the head now scores every vocabulary token, not one row of class labels.
- Tokenization turns Python strings into integer IDs; attention uses the visible context; sampling turns probabilities into the next ID.
- Quantization and memory bandwidth explain why smaller weights often mean faster local inference.

2. Local stack (Ollama)

- `ollama.chat` is the synchronous client path; useful drafts still need tests and review.
- **Time to first token** (TTFT) measures wait until the first generated chunk arrives; streaming helps even when total time is unchanged.
- Validated tags: `gemma4:e4b` for chat, `embeddinggemma` for embedding vectors in RAG demos.

3. Application patterns

- **Evaluation:** keep golden examples, schema checks, regression runs, and human review beside the prompt.
- **Structured output + RAG:** validate JSON, retrieve permitted chunks, then generate grounded answers with citations.
- **Tools / agents / guardrails:** the model proposes; Python authorizes, budgets, validates, and logs.

A local **large language model (LLM)** behaves like a stateless service call: it sees only the messages and options you send now. Session memory, retrieval, JSON parsing, evaluation, and security are entirely your Python layer.



Thank You for Your Attention

"The model has knowledge but no memory. Our job as engineers is to give it a desk."