01

Python, from Zero

Why it runs the AI world

Chapter 1, explained assuming you know nothing — from the fundamentals up.

01 The journey — five things to understand

You already know how to program. What you're really learning is a second dialect — and the culture every AI library lives inside. Each stop below is one idea: explained whole, then connected.

Follow the arrows → by the end it's one connected picture.

01

First fundamental: how does code run at all?

Two ways a computer turns your code into action

COMPILED (languages like C, Rust, Go)

A 'compiler' translates your whole program in one go, BEFORE it runs, into the raw instructions the computer follows directly. Then it runs that.

→ Runs fast. But you wait for the translate step first.

INTERPRETED (Python)

An 'interpreter' reads your program and runs it one line at a time, live, each time you run it. Nothing is translated ahead.

→ Easy and flexible. But slower, line by line.

Python is interpreted — that is exactly where its reputation for being 'slow' comes from. The next slides show why, for AI work, that barely matters.

01

'Slow language, fast libraries'

The single most important idea in the whole chapter

What you type in Python:
a @ b
(this multiplies two big tables of numbers)

hands the job over in microseconds

THE FAST ENGINE

Written in C — a low-level language the computer runs directly (on a graphics chip it's called CUDA).

It does the millions of tiny sums — very fast.

Picture Python as the REMOTE CONTROL, and the fast C code as the machine it points at. Python barely does any work — in a split second it says 'do this', and the fast code does the heavy lifting. So calling Python 'a wrapper around C' is a compliment, not an insult.

01

So what ARE NumPy, PyTorch, pandas?

They look like Python, but the real work happens in fast C code underneath

You write Python — short, easy commands

NumPy / PyTorch / pandas — the easy Python commands you type

Fast code underneath (C · C++ · CUDA · Rust) — this does the real work

Reading tip: if you see someone stepping through a big list of numbers one-by-one in Python, that's usually the slow way. The fast way hands the whole list to the library in one command (the word for this is 'vectorising').

01

Why the whole field chose Python

A 20–year head start, readability, glue — and now GenAI–first

2006 NumPy → 2008 pandas → 2015 TensorFlow → 2016 PyTorch → Today LLMs

20 years of scientific tools stacked up. When deep learning arrived, it was built on this pile — so researchers already lived in Python.

Reads like pseudocode
Clean, low–ceremony syntax. When the IDEA is the hard part, the plumbing gets out of the way.

The glue language
Talks to databases, REST APIs, files and other programs. The natural layer wiring everything together.

GenAI is Python–first
The official code kits (called SDKs) from OpenAI, Anthropic & Google — plus tools like LangChain and Hugging Face — are all built for Python first.

01

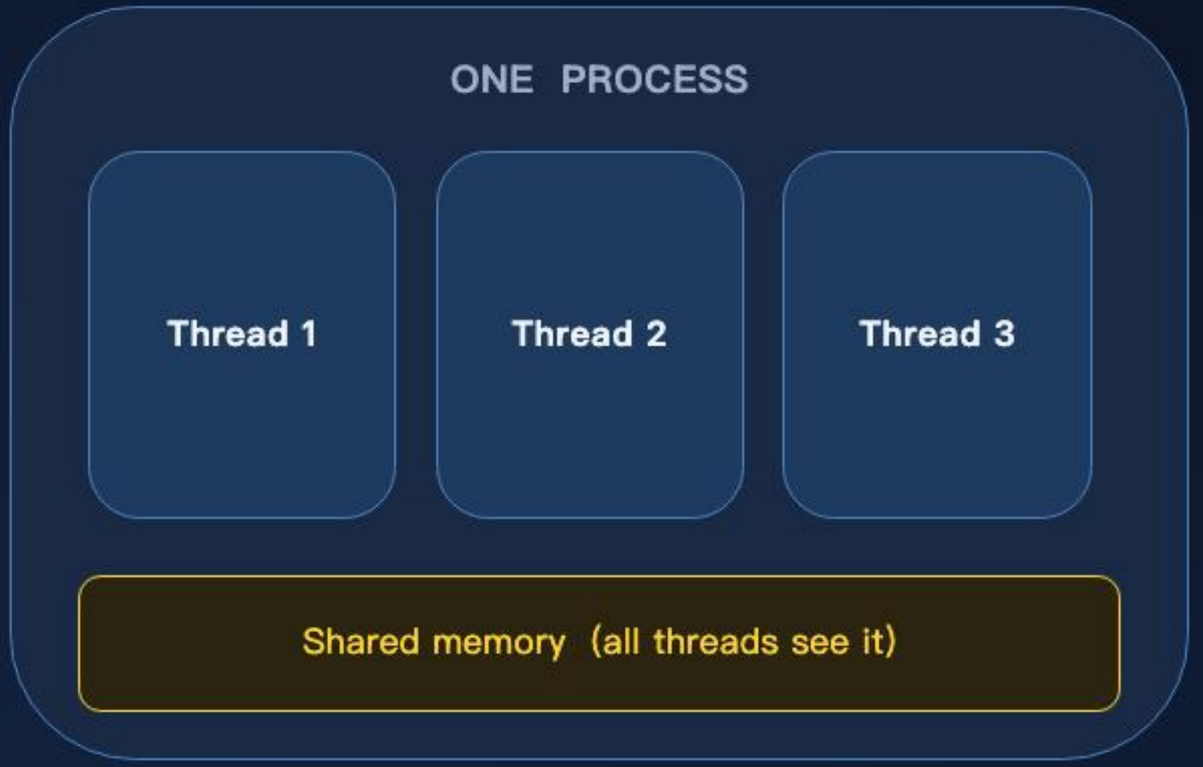
Now, toward the GIL — first, two words

A process, and a thread

PROCESS

A running program with its OWN private memory. Open two apps → two processes. They can't see each other's memory.

A THREAD is a worker inside a process. One process can run many threads, and they all share the same memory. Threads are how a program does several things 'at once' — in theory.



01

Concurrency is not parallelism

A distinction the GIL makes very real

CONCURRENCY

ONE worker juggling many tasks, switching fast between them. It LOOKS simultaneous, but only one thing truly runs at any instant.

Like one chef minding many pots.

PARALLELISM

MANY workers each doing a task at the SAME real instant. Genuinely at once.

Like many chefs, each at their own stove.

Needs several CPU 'cores' (a core = one worker in the computer's brain; modern machines have several).

A modern CPU has many cores, so it CAN run threads in true parallel — unless something forces them to take turns. In Python, something does. Meet the GIL.

01

One more word: a lock

How shared memory is kept from corrupting

The problem

Threads share memory. If two threads change the SAME data at the same instant, they can corrupt it — half-written, inconsistent, garbage.

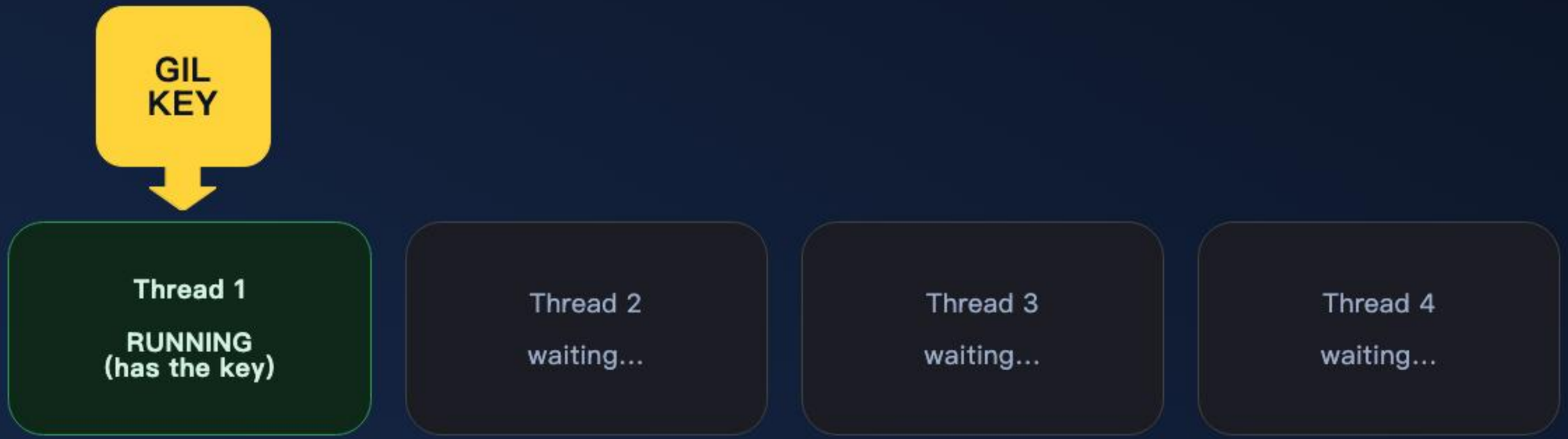
The fix: a LOCK

A single KEY. Only the thread holding the key may touch the protected data. Everyone else waits their turn, then hands the key on.

Safe — but it means access is SERIALISED, one at a time. Hold that thought: it is the whole story of the GIL.

01 The GIL — the Global Interpreter Lock

The normal Python everyone uses (its full name is CPython) has ONE big lock over the whole thing. The rule: only the worker holding this one key is allowed to run Python code. One key — one runner at a time.



So extra threads do NOT speed up pure-Python number-crunching. They take turns; they don't run in parallel.

01

But the GIL has two escape hatches

This is why Python is still great for AI agents

Escape 1 — WAITING hands the key over

When a worker is just waiting — for the internet, a file, or a reply from another online service — it isn't using the brain. So it lets go of the key, and another worker runs.

→ Most AI-agent code spends its time waiting for replies, so this works beautifully.

Escape 2 — the fast C code hands the key over

While NumPy or PyTorch are busy in their fast C code, they let go of the key too. So heavy maths DOES run on all your cores at once.

→ Python isn't even holding the lock during the work that actually matters.

If you really need lots of plain-Python work running at once, you start several separate copies of Python together, each with its own key (this is called 'multiprocessing'). And Python 3.13 now has an experimental version with no lock at all — the limit is finally easing.

01

So which tool for which job?

The practical takeaway from all of that

Mostly WAITING — for the internet, files, or replies from other services? (tech term: I/O-bound)	→ **async / threads (do other work while waiting)**
Heavy maths on big tables of numbers?	→ **NumPy / PyTorch (already runs on all cores)**
Lots of plain-Python work, across cores?	→ **multiprocessing (run several Pythons at once)**

01

One more trade-off: dynamic typing

What a 'type' is, and why Python won't catch you

STATIC (Java, C#)

You declare each value's type. The compiler CHECKS them before the program runs — mismatches are caught early.

DYNAMIC (Python & JS)

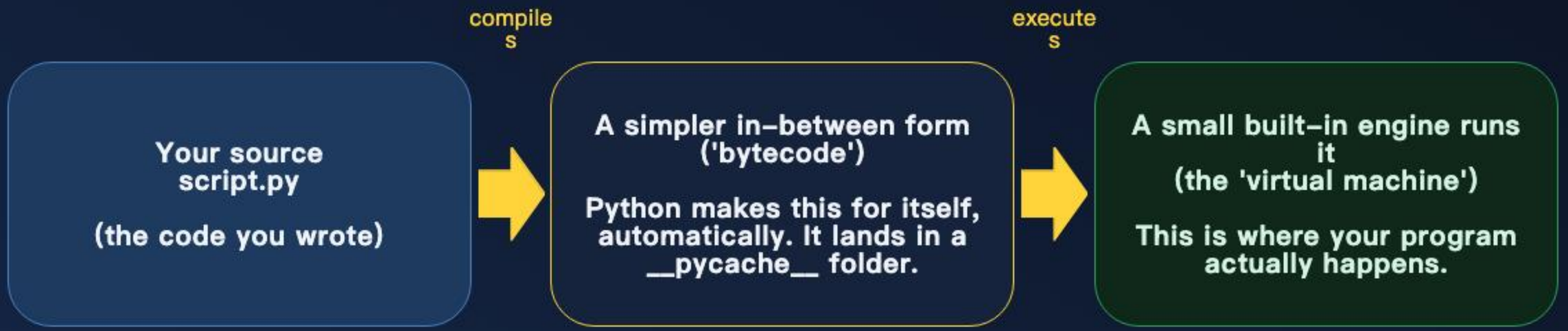
A variable just holds whatever you give it. Types are only checked as the code RUNS — so a wrong type blows up at run time, not before.

Type HINTS — e.g. def get_name(id: int) -> str

Optional labels you can add to say what goes in and what comes out ('-> str' means 'this gives back text'). Helper tools — like mypy, or your code editor — check them for you. But Python itself does NOT enforce them while running, so a function labelled '-> str' can still hand back nothing (None). Good AI codebases use them a lot: treat them as trustworthy notes, not a cast-iron guarantee.

01 What happens when you run a Python file

python script.py



There is NO separate build step you run — the compile-to-bytecode happens automatically, on import. That is why Python feels so immediate: edit, run, see the result.

01

Three ways you'll see Python run

And the notebook trap that will bite you

Script
python app.py

Runs top to bottom, then exits. This is where production code lives.

A live window (its name: the REPL)
type: python

You type ONE line of Python and it answers back straight away. Great for trying things out and seeing what a bit of code actually does.

Notebook (Jupyter / Colab)

A page split into blocks you run one at a time. A live Python session behind it (the 'kernel') remembers everything, so you load a big model ONCE and keep poking at it. Where AI people live.

The trap: a notebook's cells share ONE memory and can run in ANY order. The [n] beside a cell is the order it was RUN, not its order on the page. So a notebook can 'work' for its author yet be broken when you run it top to bottom. Trust a result only after Restart Kernel → Run All.

Reading & judging: check the [n] counters run in order down the page. Out-of-order or gaps = the outputs may not be reproducible.

01 The culture that makes code judgeable

snake_case, not camelCase
Python joins words with underscores: user_name, not the humped userName. The humped style instantly looks 'not from around here'.

Batteries included
json · pathlib · itertools · dataclasses are built in. No install for the basics — Python ships a workshop.

The Zen of Python
import this → 'one obvious way to do it.' Uniformity is what makes deviations — and bugs — stand out.

THE FLIP that catches JavaScript people out: Python treats an empty list [], empty text, zero, and 'nothing' (None) as meaning 'no, there's nothing here' — the word for that is 'falsy'. In JavaScript an empty list means 'yes, something' — the opposite! So in Python, 'if not items:' correctly means 'the list is empty'.

The teaching — Python is the remote control; the C/CUDA ecosystem is the machine; the culture is what makes code judgeable. You already program. Now you speak the dialect the whole AI field is written in.