The AI Agent Developer's Toolbelt

A Small Practical Manual for Building Agents from Scratch

Every command in here was run before it was written down

A Workshop Companion

Ayman Hamed

uv · Ollama · Git · Pydantic · FastAPI · Uvicorn · Nginx

Contents

How to use this book

This is not a workshop. It is the plumbing every agent project assumes you already have — so your time goes on building agents, not on installing Python.

Set it up once → **Build agents** → **Come back when it breaks**

Two ways to read it

- **Front to back, once, before you build anything.** Parts 1 and 2 are marked *ESSENTIAL*. That takes one evening, and it saves you time every week after that.
- **By symptom, forever after.** Parts 3 to 6 are reference. The *Troubleshooting index* near the back is the fastest page in the book — start there when something is broken.

EVERY COMMAND HERE WAS EXECUTED FIRST

Nothing on these pages was written from memory. Commands were run and their output checked; where the result was worth recording, the page carries a **VERIFIED** note. Two of those cover traps that cost beginners real time: git creating a branch called *master* when GitHub wants *main*, and a committed key surviving in history after you delete the file.

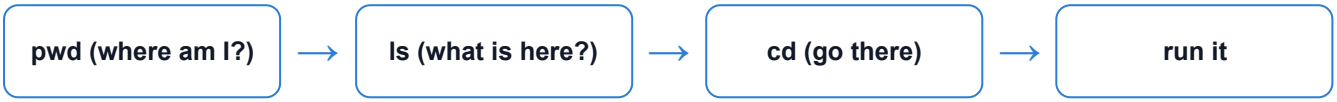
WHERE THIS CAME FROM

This is the setup companion I hand to students of my live workshop series, *Build a Real AI Agent — From Scratch*. You do not need the workshop to use it: everything here is standard developer tooling that any Python or AI project assumes you already have.

YOUR MACHINE · ESSENTIAL

The terminal: where you are

You do not need to master the terminal. You only need to know which folder you are in. Most "it does not work" problems are just the wrong folder.




```
pwd                 # print working directory — where am I?
ls                  # list what is in this folder
ls -a               # include hidden files like .env and .git
cd w1               # go into the w1 folder
cd ..               # go up one level
cd ~                # go to your home folder
cd -                # go back where you just were
```

EXAMPLE

```
You type: python agent.py
You get: can't open file 'agent.py': No such file or directory

Run pwd. It prints /Users/you — your home folder, not the project.
Run cd my_agent, then python agent.py. It works.
The file was never missing. You were in the wrong folder.
```

What to remember

- A path that starts with / or ~ is a full path. Anything else starts from your folder.
- Press Tab to finish a name. If Tab does not finish it, the file is not there. You just found your problem.
- The up arrow brings back your last command. Press it, edit it, run it again.
- Drag a folder onto the terminal window. It types the full path for you.

THE HABIT THAT SAVES THE MOST TIME

When a command fails, run pwd and then ls. Very often the answer is right there: you are in the wrong folder, or the file has a different name.

YOUR MACHINE · ESSENTIAL

The terminal: files, output, and stopping things

A few more commands cover the rest: reading a file, making a folder, and stopping a program that will not stop by itself.

```
cat notes.md            # print a whole file
head -20 log.txt        # first 20 lines (use for big files)
mkdir my_agent          # make a folder
cp a.py b.py            # copy · mv renames or moves
rm b.py                 # delete a FILE — there is no recycle bin

python agent.py > out.txt   # send output to a file instead of the screen
history | grep ollama       # what was that ollama command again?
```

EXAMPLE

```
You start a server. The terminal stops answering you.
It is not frozen — the program is running and waiting.

Press Ctrl-C. The program stops and your prompt comes back.
Now try Ctrl-C in a Python prompt: it does not exit. Ctrl-D does.
```

What to remember

- Ctrl-C stops a running program. It is the most useful key in this book.
- Ctrl-D means "I am done typing". It exits Python and most prompts.
- clear only cleans the screen. Nothing is deleted. Scroll up and it is all still there.
- The | symbol sends the output of one command into another one.

RM DOES NOT ASK, AND YOU CANNOT UNDO IT

There is no recycle bin here. `rm -rf folder` deletes the folder and everything inside it at once, forever. Type the path, then read it again before you press Enter.

YOUR MACHINE · ESSENTIAL

Python and uv

uv installs Python, creates environments, and installs packages. One tool instead of four. If you ever fought with pip and venv, this page is for you.


```
# macOS / Linux
curl -LsSf https://astral.sh/uv/install.sh | sh
# Windows (PowerShell)
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"

uv --version              # prove it installed
uv python install 3.12    # a Python of your own
uv python list            # which Pythons do I have?
```

EXAMPLE

```
$ uv --version
uv 0.6.12

$ uv python install 3.12
Installed Python 3.12

The Python that came with your computer is untouched.
```

What to remember

- uv replaces pip, venv, pyenv and virtualenv. You no longer choose between them.
- It never changes the Python that came with your computer. So it is safe to try.
- It is fast. Installing packages takes seconds, not minutes.
- The commands are the same on macOS, Linux and Windows.

YOU NEVER ACTIVATE ANYTHING

You will not see "source .venv/bin/activate" in this book. `uv run` finds the right environment for your folder by itself. Forgetting to activate is a common beginner problem, so we removed that step.

YOUR MACHINE · ESSENTIAL

uv in a real project

Four commands cover almost everything you will do. This is the page to keep open.

```
uv init --name my_agent     # start a project (pyproject.toml + git)
uv add ollama               # install a package AND record it
uv run python agent.py      # run inside this project's environment
uv sync                     # rebuild the environment from the lock file
```

EXAMPLE

```
$ uv add ollama
+ ollama==0.6.2 (and 7 more packages)
$ uv run python -c "import ollama" # no error: it worked
```

What to remember

- `uv add` installs the package and writes its name in your project file.
- You can delete .venv any time. `uv sync` builds it again in a few seconds.
- Give someone your folder. They run `uv sync` and get exactly your setup.

What each file is for

File	What it means
pyproject.toml	What you asked for. You may edit it. Commit it.
uv.lock	Exactly what got installed, to the version. Commit it.
.venv/	The installed packages themselves. Never commit — it is rebuildable.
.python-version	Which Python this folder uses. Commit it.

VERIFIED — I RAN ALL FOUR

`uv init` created the project and a git repo. `uv add ollama` installed 8 packages. I deleted .venv, ran `uv sync`, and it came back.

YOUR MACHINE · ESSENTIAL

uv: three things worth knowing later

Skip this page for now. Come back when one of these three things happens. Each one can save you an hour.

```
# 1. A one-file script that needs a package, with no project at all
uv add --script solo.py ollama     # writes the dependency INTO the script
uv run solo.py                     # uv installs it on the fly and runs

# 2. A command-line tool you want everywhere, not in one project
uvx ruff check .                   # run it once, install nothing
uv tool install ruff               # or keep it permanently

# 3. Tools you develop with but do not ship
uv add --dev pytest
```

EXAMPLE

You want to test one idea in one file. You do not want a project.

```
$ uv add --script solo.py ollama
Updated `solo.py`

uv wrote a small comment block at the top of solo.py listing ollama.
Now `uv run solo.py` installs it and runs the file.
```

What to remember

- One small script can carry its own packages, written in a comment at the top.
- `uvx` runs a tool once without installing it. There is nothing to clean up after.
- `--dev` keeps test tools separate from what your project needs in order to run.
- `uv remove <package>` uninstalls it and removes its name from the project file.

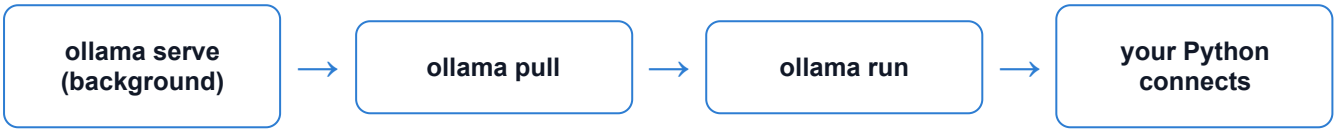
VERIFIED ON UV 0.6.12

`uv add --script solo.py ollama` added a small `# /// script` block at the top of the file. Then `uv run solo.py` ran it — with no project, no .venv, and no install command from me.

YOUR MACHINE · ESSENTIAL

Ollama: your local model

Ollama runs a real language model on your own computer. No API key, no credit card, no cost per message. It even works with no internet.




```
# install from ollama.com, then:
ollama --version
ollama serve                  # start the background service
ollama pull qwen2.5           # download the course model (~4.7 GB)
ollama run qwen2.5            # chat with it in the terminal
                              # (type /bye to leave)
```

EXAMPLE

```
$ ollama run qwen2.5
>>> say hello in five words
Hello there, how are you?
>>> /bye

No API key. No internet needed. The model is on your disk.
```

What to remember

- `ollama serve` must be running before any Python in this book will work.
- On macOS and Windows the app usually starts it for you. Check with `ollama ps`.
- Your Python talks to it at http://localhost:11434 — your own computer.
- If you see ConnectionError, check this first: Ollama is probably not running.

DOWNLOAD THE MODEL BEFORE THE SESSION

qwen2.5 is a few gigabytes. On shared wifi with twenty people downloading at the same time, it will not finish. Download it at home, then run `ollama run qwen2.5` once to check that it answers.

YOUR MACHINE · ESSENTIAL

Ollama: managing models and disk

Models are big files. These commands show what you have, what is running now, and let you delete what you do not need.

```
ollama list                # what have I downloaded, and how big?
ollama ps                  # what is loaded in memory right now?
ollama show qwen2.5        # context length, parameters, licence
ollama stop qwen2.5        # unload it from memory
ollama rm llama3.2         # delete the download, free the disk
```

EXAMPLE

```
Your lesson says: model "qwen2.5" not found.

$ ollama list
NAME SIZE
qwen2.5:latest 4.7 GB

The model is there. The name in your code was wrong.
```

What to remember

- If you see "model not found", run `ollama list` and copy the name exactly, including the part after the colon.
- A model loads into memory when you first use it. So the first answer is slow, and the next ones are faster.
- Small models are faster but weaker at calling tools. A model can chat well and still fail to use your tools.
- The lessons read WORKSHOP_MODEL, so you can point them at any model you have.

VERIFIED ON OLLAMA 0.32.7

`ollama ps` printed a table of the models loaded right now. `ollama list` printed every model I had downloaded, with its size.

YOUR MACHINE · ESSENTIAL

VS Code: a workable setup

Any editor works. If you do not have a favourite, here is a ten-minute setup — and one setting that prevents many confusing errors.

EXAMPLE

```
The editor draws a red line under: import ollama
But `uv run python agent.py` runs perfectly.

Your code is correct. The editor is looking at a different Python.
Choose the interpreter inside .venv and the red line disappears.
```

What to remember

- Install the **Python** extension from Microsoft. You do not need any other one.
- Open the FOLDER, not the file: File → Open Folder. Then the editor knows where your project is.
- Use the terminal inside the editor (Ctrl-` or Cmd-`). It opens in your project folder.
- Choose the interpreter: Cmd/Ctrl-Shift-P → "Python: Select Interpreter" → pick the one inside `.venv`. This makes imports and autocomplete work.
- Python runs the file **saved on disk**, not the text on your screen. Save before you run.

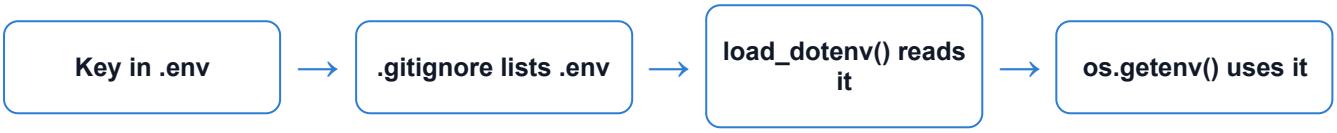
THE MOST COMMON EDITOR CONFUSION

The editor shows a red line under an import, but `uv run python agent.py` works. Your code is fine. The editor is looking at the wrong Python. Choose the interpreter inside `.venv` again.

YOUR MACHINE · ESSENTIAL

.env and secrets

If you write a key inside your code, it goes public the moment you push. The fix is simple: keep keys in one file, and tell git to ignore that file.




```
# .env  — NEVER committed
GOOGLE_API_KEY=your_key_here
TELEGRAM_TOKEN=8123456789:AAH...

# .gitignore  — ALWAYS committed
.env
.venv/
__pycache__/

# in your Python
from dotenv import load_dotenv
import os
load_dotenv()                       # reads .env in the CURRENT folder
key = os.getenv("GOOGLE_API_KEY")   # None if it is missing
```

EXAMPLE

`print(os.getenv("GOOGLE_API_KEY")) prints None`

`The key is in the file. But you ran python from the folder above.`
`load_dotenv() looked there, found no .env, and stayed quiet.`

`Run pwd. Move to the folder with the .env. Run again.`

What to remember

- `load_dotenv()` looks in the folder you ran python from, not where the file is saved.
- `os.getenv(name)` gives you None if the key is missing. Check for that and print a clear message.
- Save a `.env.example` with the names but no values, so others know what to fill in.
- Use one `.env` for each project. Do not share one file between projects.

YOUR MACHINE · ESSENTIAL

Secrets hygiene

Almost everyone shares a key by mistake one day. The difference between a small mistake and a real problem is knowing what to do next.

EXAMPLE

```
You share your screen and your .env is open for two seconds.

Wrong answer: "nobody was really looking".
Right answer: open the provider's page, delete that key, make a new one,
and paste the new one into .env. Two minutes, and it is over.
```

What to remember

- A shared key is not a disaster if you replace it. It becomes one if you hide it.
- Even a free key costs you. Someone else can use up your limit and block your account.
- Anything you commit to git stays in the history of every copy anyone downloaded.

The four rules

Rule	What it means
Use app passwords	Many services give you a password for one app only. You can cancel it alone. Never put your account password in a program.
Never paste a key	Not in a chat, a screenshot, or a slide. Anything you paste may stay there forever.
Replace it early	If you only think a key was seen, replace it. Making a new key takes two minutes. Checking who saw it takes much longer.
Keep each key small	One key for each project, and give it the smallest permissions the job needs.

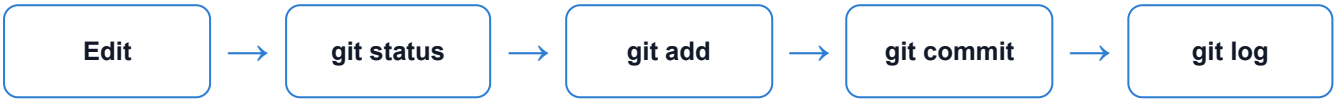
VERIFIED — DELETING THE FILE DOES NOT DELETE THE KEY

I committed a .env by mistake. Then I ran `git rm --cached .env` and added it to .gitignore. Git stopped following the file — but the key was still easy to read in the history. So: **make a new key.** That is the real fix. Do both, in that order.

GIT AND GITHUB · ESSENTIAL

git: config and the daily loop

git has hundreds of commands. You will use six of them, in the same order, every day. Learn those six as one habit. The rest can wait.




```
# once per machine, before anything else
git config --global user.name  "Your Name"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main

git init                    # start tracking this folder
git status                  # what changed? (run this constantly)
git add .                   # stage everything you changed
git commit -m "add weather tool"
git log --oneline           # the history, one line each
```

EXAMPLE

```
$ git add . && git commit -m "add weather tool"
[main 3f2a1b] add weather tool # git status is now clean
```

What to remember

- `git status` is fast and safe. Run it before and after everything.
- A commit is a save point with a short note. Many small commits are better than one big one.
- Write the note as an instruction: "add weather tool", not "added stuff".
- Nothing is shared until you push. A commit stays on your computer.

VERIFIED — SET THESE THREE FIRST

On git 2.40, `git init` made a branch called **master**, but GitHub expects **main**. Setting init.defaultBranch fixed it. Also, without user.name and user.email, your first commit fails with "Please tell me who you are."

GIT AND GITHUB · ESSENTIAL

git: the three places your work lives

Almost every confusing git moment comes from one question: where is my change right now? There are three possible places. Learn them and git makes sense.


```
git diff                  # working tree vs staged — what have I not staged?
git diff --staged         # staged vs history — what am I about to commit?
git add agent.py          # move one file into the staging area
git status --short        # M = modified · ?? = untracked · A = added
```

EXAMPLE

```
You edit agent.py.

$ git diff shows your change
$ git add agent.py
$ git diff shows nothing
$ git diff --staged shows your change again

Nothing was lost. The change moved to the next place.
```

What to remember

- Your editor saves to the working tree. git does not see it until you run `git add`.
- The staging area is where you choose what goes into this commit.
- After `git add`, `git diff` shows nothing. That is normal. Use `--staged` instead.
- Read the whole diff before you commit. It is the easiest review you will ever do.

VERIFIED — THE MOMENT THIS BECOMES USEFUL

I edited a file and `git diff` showed the change. Then I ran `git add`. Now `git diff` showed nothing, and `git diff --staged` showed the change instead. The change did not disappear. It moved to the next place.

GIT AND GITHUB · ESSENTIAL

git: undoing things safely

git can undo almost anything. You only need to pick the right undo. One of them deletes your work. The others do not.

EXAMPLE

```
You edited the wrong file and want your old version back.

git restore wrong_file.py your edits are gone, the file is back

You already committed, and already pushed:

git revert 3f2a1b a new commit undoes the old one
```

What to remember

- After you push, use `revert`, not `reset`. It keeps the true history.
- Only amend a commit you have not pushed. Amending changes history, which confuses others.
- `git log --oneline` shows the codes used in the table above. The first 7 letters are enough.
- When you are not sure, commit first. You can fix a messy commit. You cannot get back a lost edit.

Which undo do you want?

Command	What it means
git restore file.py	Delete your edits to that file. You cannot get them back.
git restore --staged file.py	Take it out of the staging area. Your edits stay. Safe.
git commit --amend -m "..."	Change the note or the content of your LAST commit.
git revert <hash>	Make a new commit that undoes an old one. Safe.
git reset --hard	Delete commits and your changes. This is the dangerous one.

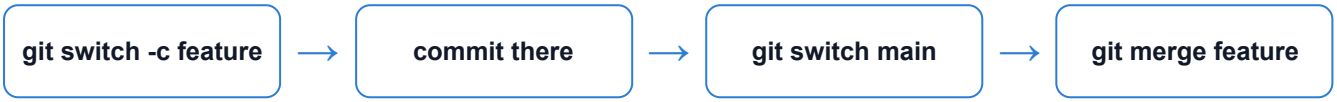
BE SLOW WITH THIS ONE COMMAND

`git reset --hard` deletes work you have not committed. It does not ask, and you cannot get it back. Everything else in the table can be undone. If a tutorial tells you to run it, commit first.

GIT AND GITHUB · ESSENTIAL

git: branches and merges

A branch is a copy where you can try something without breaking what already works. You do not need one on day one. You will want one soon.




```
git switch -c add-telegram   # create a branch and move onto it
git switch main              # back to the main line
git branch                   # which branches exist? (* marks yours)
git merge add-telegram       # bring the branch's work into main
git branch -d add-telegram   # tidy up once merged
```

EXAMPLE

```
$ git switch -c add-telegram # main stays safe while you work here
Switched to a new branch 'add-telegram'
```

VERIFIED — WHAT A REAL CONFLICT LOOKS LIKE IN YOUR FILE

```
<<<<<<< HEAD
model = "gemma3"          <- what main says
=======
model = "llama3.2"        <- what the branch says
>>>>>>> feature-x

# Delete the markers, leave the version you want, then:
#   git add cfg.py && git commit
```

What to remember

- main should always work. Try new things on a branch, named after what it does.
- Changing branch changes the files on your computer. Commit before you switch.
- If two branches changed the same lines, git stops and asks you to choose. That is a conflict.
- A conflict is not an error. git is refusing to guess for you.

GIT AND GITHUB · ESSENTIAL

.gitignore that actually protects you

One file tells git what to ignore. Write it once, at the start, and you will never share a key or commit a huge folder of packages by mistake.

```
# .gitignore — put this in every project in this course
.env                # secrets. The important line.
.venv/              # installed packages — rebuildable with uv sync
__pycache__/        # compiled Python
*.pyc
.DS_Store           # macOS folder clutter
memory/             # your agent's written memory, if it is private
*.log
```

EXAMPLE

```
$ git status
Untracked files:
.env <— STOP. Do not commit.

Add .env to .gitignore, then run git status again.
The .env is gone from the list. Now it is safe to push.
```

What to remember

- A `/` at the end means folder. A `*` at the start matches any name that ends that way.
- .gitignore only works on files git is not already following. If you already committed the file, stop following it first.
- Commit the .gitignore file itself. Everyone working on the project needs it.
- `uv init` writes a starter .gitignore. Open it and add `.env`. That is the most important line.

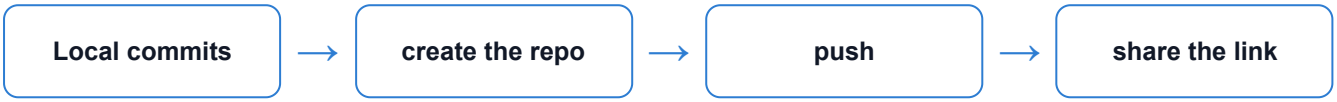
VERIFIED — A TEN-SECOND CHECK THAT PREVENTS THE WORST MISTAKE

I put `.env` in .gitignore, made a real .env with a fake key, then ran `git add .` and committed. `git ls-files` showed only .gitignore and agent.py. git never saw the .env. **Run `git status` before your first push. Every time.**

GIT AND GITHUB · ESSENTIAL

GitHub: push, clone, pull

git lives on your computer. GitHub is a copy that other people can see. It is your backup, your portfolio, and how you send your work to someone.




```
# the easy way, with GitHub's own CLI
gh auth login                                  # once per machine
gh repo create my-agent --private --source=. --push

# the manual way (after making the repo on github.com)
git remote add origin https://github.com/you/my-agent.git
git push -u origin main      # first push sets the default
git push                     # every push after that

git clone <url>              # get a copy of someone else's repo
git pull                     # bring down changes you do not have yet
```

EXAMPLE

```
$ gh repo create my-agent --private --source=. --push
Created repository you/my-agent on GitHub
pushed to https://github.com/you/my-agent

From now on, one word is enough: git push
```

What to remember

- Start private. Make it public when you are happy with it.
- Before your first push, run `git status` and check that .env is not there.
- If you use two computers, run `git pull` before you start. It prevents many conflicts.
- Push often. A commit on your own computer is not a backup.

GIT AND GITHUB · ESSENTIAL

GitHub: pull requests and handing in your work

A pull request is a branch asking to join main, with a place to talk about it. It is also the clearest way to show someone what you built.

```
git switch -c add-telegram
# ... commit your work ...
git push -u origin add-telegram
gh pr create --fill          # title and body from your commit messages
gh pr view --web             # open it in the browser
```

EXAMPLE

Test your own submission before you send it:

1. git clone your own repository into a new folder
2. uv sync
3. run the one command from your README

If it works without you touching anything, it is ready.

What to remember

- A pull request shows only what changed. The reviewer does not read the whole project.
- `--fill` uses your commit notes as the title. One more reason to write them well.
- Your repository link is your submission. The README is the first thing people read.
- A README with one working command looks finished. Two short paragraphs are enough: what it does, and how to run it.

WHAT I LOOK FOR WHEN YOU HAND IN A PROJECT

Can I download it, run `uv sync`, follow one command from your README, and see it work — without asking you any question? If yes, your project is finished.

PYTHON ESSENTIALS · REFERENCE

Type hints and docstrings

In most Python courses, these are just good manners. Here they are important: the model reads the types and the description to choose a tool. Write them carelessly and the agent picks the wrong tool.

```
def get_weather(city: str) -> str:
    """Get the current weather for a city.

    Args:
        city: The city name, e.g. "Casablanca".
    """
    return "22C and sunny"
```

EXAMPLE

Two tools. The model can only read the description:

```
get_weather(city: str)
"Get today's weather for a city." -> the model uses this one

helper(x)
"Helper function." -> the model never uses it
```

What to remember

- `city: str` says what goes in. `-> str` says what comes out.
- Python does not check these. But the model, your editor and other people all read them.
- The docstring is the text in triple quotes at the top of the function. The model reads it.
- Types you will use most: `str`, `int`, `float`, `bool`, `list[str]`, `dict`.

WHY THIS PAGE IS IN A TOOLS BOOK

In Workshop 1 you write a function, and the model decides to call it using only this description. If the model never picks your tool, read your docstring again. The problem is usually there, not in the model.

PYTHON ESSENTIALS · REFERENCE

Dicts, f-strings, and returning data

Three small pieces of Python that appear on almost every page of this course.

```
# a dict — keys and values. Messages to a model are dicts.
message = {"role": "user", "content": "hello"}
print(message["role"])            # 'user' — errors if the key is missing
print(message.get("name"))        # None if missing — safer

# an f-string — put values inside text
city = "Rabat"
print(f"Weather in {city}: 22C")

# returning structured data from a tool
return {"status": "ok", "saved": text}
```

EXAMPLE

```
message = {"role": "user", "content": "hello"}

message["name"] -> KeyError: 'name' (your program stops)
message.get("name") -> None (your program continues)

Use .get() for anything that came from outside your program.
```

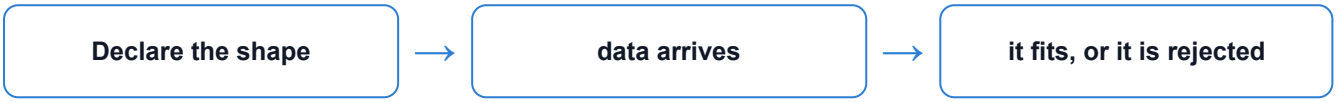
What to remember

- `d["key"]` gives an error if the key is missing. `d.get("key")` gives None instead.
- Use `.get()` for anything that comes from outside your program.
- An f-string needs the letter `f` before the quote. Without it, you print the braces.
- `if __name__ == "__main__":` means "run this only when I start this file directly".

PYDANTIC AND FASTAPI · REFERENCE

pydantic: declaring a shape

You do not need to learn pydantic now. You only need to recognise it. Workshop 3 uses it to check every request before your agent sees it.




```
from pydantic import BaseModel, Field

class ChatRequest(BaseModel):
    message: str = Field(min_length=1, max_length=4000)

ChatRequest(message="hello")     # fine
ChatRequest(message="")          # raises ValidationError
```

EXAMPLE

```
ChatRequest(message="hello") -> fine
ChatRequest(message="") -> ValidationError

The second one never reaches your agent, so you never pay for a model
call, and you never debug a strange empty answer.
```

What to remember

- A model describes what good input looks like. The type hints become real checks.
- It refuses bad input at the entrance, not deep inside your program.
- You already have it. It comes with ollama, and FastAPI is built on it.
- Useful rules: `min_length`, `max_length`, and `ge` / `le` for numbers.

WHY AN AGENTS COURSE CARES ABOUT THIS

If your agent accepts a message of 200,000 characters, it will try to send it to the model. One line of pydantic turns a strange failure into a clear message.

PYDANTIC AND FASTAPI · REFERENCE

pydantic: reading the error

At first these errors look like noise. They are not. They tell you which field failed, which rule it broke, and what you actually sent.

EXAMPLE

```
Read the four lines as four questions:

Which model failed? ChatRequest
Which field? message
Which rule? at least 1 character
What did I send? input_value=''

Now you know exactly what to fix.
```

VERIFIED — THE REAL OUTPUT, IN FULL

```
1 validation error for ChatRequest
message
  String should have at least 1 character [type=string_too_short, input_value='',
input_type=str]
    For further information visit https://errors.pydantic.dev/2.13/v/string_too_short
```

What to remember

- Line 1: how many problems there are, and in which model.
- Line 2: which field failed. Here it is `message`.
- Line 3: the rule, then the name of the rule, then `input_value` — what you sent.
- Line 4: a link to a page about that rule. It is worth opening.

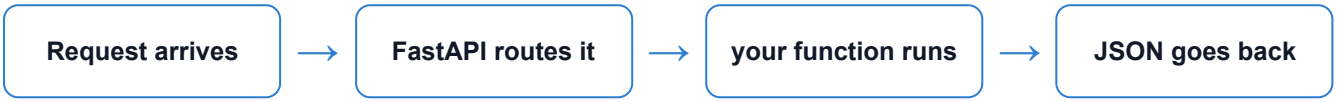
READ IT IN THIS ORDER, EVERY TIME

Which model? Which field? Which rule? What did I send? Four questions, four lines. Read it this way and it stops looking like a wall of text.

PYDANTIC AND FASTAPI · REFERENCE

FastAPI: a function with an address

FastAPI turns a Python function into a web address. That is the whole idea. Workshop 3 spends a full session on it. This page is just so it looks familiar.




```
from fastapi import FastAPI

app = FastAPI()

@app.get("/health")
def health():
    return {"status": "ok"}

# run it:
#   uv run uvicorn main:app --reload
# then open http://127.0.0.1:8000/docs
```

EXAMPLE

```
$ uv run uvicorn main:app --reload
Uvicorn running on http://127.0.0.1:8000

Open http://127.0.0.1:8000/health in your browser:
{"status":"ok"}

Your Python function now has a web address.
```

What to remember

- The line above the function is the address. `@app.get("/health")` means: when someone asks for /health, run this function.
- Return a dict. FastAPI turns it into JSON for you.
- `main:app` means "the thing called app, inside the file main.py". The colon confuses everyone the first time.
- `--reload` restarts the server when you save a file. Use it only while developing.

PYDANTIC AND FASTAPI · REFERENCE

FastAPI: bodies, docs, and ports

Two more things and you can read all of Workshop 3: how data arrives in a request, and why the free documentation page is such a good testing tool.

```
@app.post("/chat")
def chat(request: ChatRequest):        # a pydantic model as the argument
    return {"reply": agent.run(request.message)}
```

EXAMPLE

```
Open http://127.0.0.1:8000/docs

You see a page with your /chat route on it. Click "Try it out",
type a message in the box, and press Execute.

You just tested your own API without writing a client.
```

What to remember

- Use a pydantic model as the argument. FastAPI then checks every request for you, and writes the documentation too. That is why Workshop 3 uses both together.
- GET asks for something. POST sends something. A chat message is a POST.
- Open `/docs` and you get a page where you can call your own API by filling a form. It is made from your code, so it is never old.
- `localhost:8000` means your own computer, door number 8000. Only you can reach it.

"ADDRESS ALREADY IN USE"

Two programs cannot use the same port. This happens often in Workshop 3 when an earlier lesson is still running. Stop it with Ctrl-C, or start the new one on another port with `--port 8010`.

SERVING: FASTAPI, UVICORN, NGINX · REFERENCE

The three jobs: FastAPI, Uvicorn, Nginx

People mix these three up, because a request passes through all of them. They are not three choices. They are three different jobs, in three positions.

EXAMPLE

```
https://myagent.com/chat -> Nginx (port 443, removes encryption)
-> Uvicorn (127.0.0.1:8000)
-> FastAPI runs your chat()
```

What to remember

- **Nginx handles traffic.** It faces the internet. It never runs your Python.
- **Uvicorn runs your app.** It owns the port, speaks HTTP, and calls your code.
- **FastAPI is your app.** It chooses which function runs and what the answer is.
- Look at the arrows, not only the boxes. The language changes at every step: HTTPS, then normal HTTP on your own machine, then ASGI — which is just a function call.

THE SHORT VERSION

Nginx is the doorman. Uvicorn is the engine that runs your program. FastAPI is the program. You always need the last two. You need the doorman only when strangers can reach your building.

SERVING: FASTAPI, UVICORN, NGINX · REFERENCE

Why can't Nginx talk to FastAPI directly?

Everyone asks this. The answer is not "that is the habit". The answer is that these two cannot connect at all. Something has to run the Python.

EXAMPLE

```
Try it: point Nginx at your main.py file instead of at Uvicorn.

Nginx cannot run Python, so nothing happens.
Point it at 127.0.0.1:8000 with no Uvicorn running, and you get:
502 Bad Gateway

Nothing is listening on that port. Uvicorn is what listens.
```

VERIFIED — FASTAPI ANSWERING A REQUEST WITH NO SERVER AND NO PORT

```
# call the app the way Uvicorn does: app(scope, receive, send)
{'type': 'http.response.start', 'status': 200}
{'type': 'http.response.body', 'body': b'{"status":"ok"}'}

# A 200 response, no socket opened, nothing listening.
# That is the proof: FastAPI is a callable, not a server.
```

What to remember

- **FastAPI is not a server.** It never opens a port. There is nothing for Nginx to connect to.
- **Nginx cannot run Python.** It cannot import your file or keep your objects in memory. It only passes HTTP to a program that can.
- **They speak different languages.** Nginx speaks HTTP. FastAPI speaks ASGI, which is Python function calls. Uvicorn translates between them. That is its whole job.
- So the chain is real: Nginx → **HTTP** → Uvicorn → **ASGI** → FastAPI. Take Uvicorn away and nothing fills the gap.

SERVING: FASTAPI, UVICORN, NGINX · REFERENCE

ASGI: the contract in the middle

ASGI is an agreement. It lets any ASGI server run any ASGI framework. Uvicorn does not need to know what FastAPI is, and FastAPI does not need to know that Uvicorn is in front of it.

```
# Every ASGI application is one callable with three arguments
async def app(scope, receive, send):
    ...

#   scope    — a dict about this connection: type, method, path, headers
#   receive  — await it to get incoming data (the request body, events)
#   send     — await it to send data back (status, headers, body)
```

EXAMPLE

Stop Uvicorn. Start a different ASGI server instead:

```
hypercorn main:app
```

Your FastAPI code does not change by one letter. It still works.
That is what an agreement between two programs buys you.

What to remember

- This matters for agents: your function spends most of its time *waiting* for the model. While it waits, the same process can serve other people.
- `async def` does not make your code faster. It makes waiting cost almost nothing.

WSGI (older) vs ASGI (modern)

Standard	What it means
WSGI	One request keeps one worker busy until it finishes. No WebSockets. Flask and older Django use it.
ASGI	One worker can handle many requests while they wait for a database, an API or a model. It also supports WebSockets and long connections.

SERVING: FASTAPI, UVICORN, NGINX · REFERENCE

Uvicorn: running your app

Uvicorn is the one you type every day. Three options cover development. Two more cover a real server.

```
# development — on your machine only, restarts when you save
uv run uvicorn main:app --reload

# main:app  =  the object called `app`, inside the file main.py
#              ^^^^ file        ^^^ variable

# production — reachable from outside, several worker processes
uv run uvicorn main:app --host 0.0.0.0 --port 8000 --workers 4

# behind Nginx, trust the forwarded headers it sets
uv run uvicorn main:app --proxy-headers --forwarded-allow-ips="127.0.0.1"
```

EXAMPLE

```
$ uv run uvicorn main:app --reload
INFO: Uvicorn running on http://127.0.0.1:8000
INFO: Started reloader process
INFO: Application startup complete

Save a file. It restarts by itself and prints those lines again.
```

What to remember

- `--reload` is for development only. It watches your files, and it is slower.
- `--host 127.0.0.1` means *your computer only*. `--host 0.0.0.0` means anyone who can reach your machine. Do not type it by accident.
- `--workers` starts several processes. **Each one has its own memory**, so anything you saved in a global variable is no longer shared between them.
- In this course you run Uvicorn alone, with no Nginx. That is correct on your own computer, and it is why lesson 15 says this is not a real deployment.

SERVING: FASTAPI, UVICORN, NGINX · REFERENCE

Nginx: what it adds, and when you need it

Nginx does none of your work. It does everything around your work. On a public server, that turns out to be a lot.

```
# the minimal reverse-proxy config, in full
server {
    listen 80;
    server_name myagent.example.com;

    location / {
        proxy_pass http://127.0.0.1:8000;   # <- Uvicorn
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

EXAMPLE

One server, two apps, one address:

```
myagent.com/ -> 127.0.0.1:8000 (your agent)
myagent.com/notes/ -> 127.0.0.1:8010 (another app)
```

What to remember

- Those three `X-Forwarded-*` lines are the connection between the two. They tell your app who the real visitor is, because now every request comes from Nginx.

Do you actually need it?

Situation	What it means
On your own computer	No. Uvicorn alone is enough. Adding Nginx only adds confusion.
One small public app	Probably yes — for HTTPS, and for limits on speed and size.
Several apps, one server	Yes. One public door, sent to several Uvicorns by name or path.
Serving images or files	Yes. Nginx sends a file much faster than Python can.

SERVING: FASTAPI, UVICORN, NGINX · REFERENCE

The boundary in one experiment

This small detail between Nginx and Uvicorn causes real problems on real servers. So here is the result of running it, not just a claim.

EXAMPLE

You add a rule: five messages per visitor per minute.

On your computer it works.
Behind Nginx, every visitor looks like 127.0.0.1 — so after five messages from anyone, everybody is blocked.

One missing option, and a working feature blocks all your users.

VERIFIED — THE SAME REQUEST, WITH AND WITHOUT --PROXY-HEADERS

```
# a route returning request.client.host and request.url.scheme,
# called with  X-Forwarded-For: 203.0.113.9   X-Forwarded-Proto: https

uvicorn --no-proxy-headers  ->  {'client': '127.0.0.1',   'scheme': 'http'}
uvicorn --proxy-headers     ->  {'client': '203.0.113.9', 'scheme': 'https'}
```

What to remember

- Without the option, your app thinks **every** visitor is `127.0.0.1`. From where it sits, that is true: every request really does come from Nginx.
- So limits per user, logs and blocking all stop working — quietly, on the real server.
- Your app also thinks the visitor used plain `http`. Any link it builds sends people out of HTTPS.
- Only trust these lines from a proxy you control (`--forwarded-allow-ips`). Anyone can send an `X-Forwarded-For` line.

WHAT "BOUNDARY" MEANS IN PRACTICE

Each part only knows what the part in front tells it. Nginx knows the real visitor. Uvicorn knows only what Nginx wrote in the headers. FastAPI knows only what Uvicorn passed to it. Bugs live in these gaps between the parts, not inside them.

SERVING: FASTAPI, UVICORN, NGINX · REFERENCE

The request, end to end

Eight steps: four going in, four coming back. Read them slowly once, and the three boxes stop being abstract.

1
Browser sends
GET /items/5

2
Nginx terminates TLS,
forwards over HTTP

3
Uvicorn accepts it
on 127.0.0.1:8000

4
Uvicorn builds a *scope*
and calls your app

5
FastAPI matches the
route,
validates, calls your
function

6
Your function returns
a Python value

7
FastAPI turns it into
status + headers + JSON

8
Uvicorn writes HTTP
bytes;
Nginx passes them back

EXAMPLE

```
Your page shows nothing. Which step failed?

Server terminal is silent -> steps 1–3, it never arrived
You see the request, then red -> step 6, your code raised an error
Correct answer, blank page -> steps 7–8, the client is the problem

Naming the step turns "it is broken" into one place to look.
```

What to remember

- Steps 1-4 go **inward**, and become more like Python at every step. Steps 5-8 come **outward**, and become more like HTTP at every step.
- Your own code is step 6. Everything else is done for you by these three tools.
- When something breaks, find the step. "Nothing reaches my function" is steps 1-5. "The answer is wrong" is step 6. "The answer never arrives" is steps 7-8.

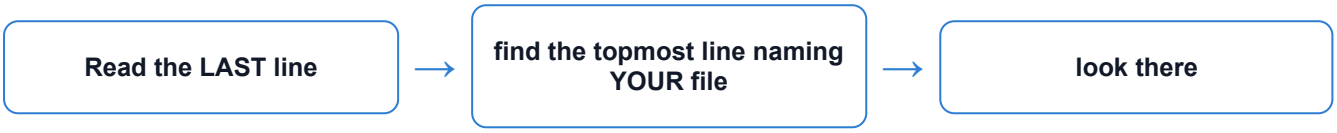
TAKE ONE PIECE AWAY

Remove Nginx: steps 2 and 8 disappear, and everything still works without HTTPS. That is your own computer. Remove Uvicorn: there is no step 3 or 4, nothing is listening, and your app never runs. That difference is the whole answer.

WHEN IT BREAKS · REFERENCE

Reading a traceback

A traceback is not blaming you. It is a map. It is printed in a strange order, so read it from the bottom.




```
Traceback (most recent call last):
  File "agent.py", line 42, in main          <- 2. your file, your line
    reply = agent.run(text)
  File "shared.py", line 17, in run
    response = self.client.chat(...)
ConnectionError: [Errno 61] Connection refused  <- 1. read this FIRST
```

EXAMPLE

```
The last line says: ConnectionError: Connection refused

So: something you tried to reach is not running.
In this course that is almost always Ollama.

$ ollama serve
Run your program again. It works.
```

What to remember

- The last line says WHAT went wrong. The lines above say HOW the program got there.
- Look upward for the first line with a file you wrote. Start looking there.
- Copy the last line into a search engine, exactly as it is. Someone had the same error.
- Change one thing, then run again. Two changes at once teach you nothing.

PRINT THE VALUE YOU ASSUMED

Do not guess what a variable holds. Print it. Most bugs are simply a value that is not what you expected — an empty string, a None, or a list instead of a dict.

WHEN IT BREAKS · REFERENCE

The errors you will actually meet

Five errors cause most of the trouble in this course. Each one takes less than a minute to fix, once you recognise it.

EXAMPLE

```
$ python agent.py
ModuleNotFoundError: No module named 'ollama'

But you installed ollama yesterday. Two possible reasons:
you are outside the project folder, or
you typed python instead of uv run python.
```

What to remember

- `IndentationError` means spaces and tabs are mixed. Set your editor to spaces only.
- `TypeError: takes 2 positional arguments but 3 were given` means you passed one argument too many.
- If the error only names files you did not write, the cause is still usually the last line you changed.

Symptom and cause

Error	What it means
ModuleNotFoundError	The package is not installed here. Run `uv add`, or check that you are inside the project folder.
FileNotFoundError	Wrong folder, or the name is spelled differently. Run `pwd`, then `ls`.
SyntaxError	A typing mistake. Usually a missing bracket, quote or colon, on that line or the line before it.
KeyError	You asked a dict for a key it does not have. Use `.get()` instead.
ConnectionError	The other program is not running. Here, that means Ollama.

WHEN IT BREAKS · REFERENCE

Troubleshooting index

The problem is on the left. The first thing to try is on the right. Come to this page before you ask anyone for help.

EXAMPLE

```
Use it like a menu. You see:
Address already in use

The table says: an older lesson is still running.
Press Ctrl-C in that terminal, or start this one with --port 8010.

Ten seconds, no searching.
```

It happens · try this first

Symptom	What it means
Nothing works, everything is missing	You are in the wrong folder. Run `pwd`.
Python cannot find a package you installed	It went to another place. Use `uv run python`, not `python`.
Red lines in the editor, but the code runs	Choose the `.venv` interpreter. Your program is fine.
Any lesson: Connection refused	`ollama serve`, then `ollama ps` to confirm.
model not found	`ollama list` and match the name exactly, tag included.
Address already in use	An older lesson is still running. Ctrl-C it, or `--port 8010`.
git: Please tell me who you are	Set user.name and user.email globally.
Push rejected	`git pull` first, then push again.
The agent ignores my tool	Read the docstring as the model sees it. Rewrite it.

WHEN IT BREAKS · REFERENCE

How to ask for help

A clear question gets an answer in two minutes. An unclear one can cost you a whole evening. The difference is four short sentences.

EXAMPLE

```
Weak: "FastAPI is not working, can you help?"

Strong: "I ran `uv run uvicorn main:app`. I expected the server to
start. I got `Address already in use`. I already closed my editor."

The second one gets an answer in one message.
```

What to remember

- **What I ran** — copy the exact command. Do not describe it.
- **What I expected** — one sentence.
- **What happened** — the full error as text, not a photo of your screen.
- **What I already tried** — so nobody repeats what you did.
- Then send the smallest example that still fails. Cutting 80 lines down to 10 often shows you the answer before you send anything.

SAY THE PROBLEM OUT LOUD

Explain it in full sentences, as if to someone who has never seen your code. Many bugs become clear in the middle of that explanation. This is not a joke. It is one of the most useful things you can do.

Cheatsheet: terminal, uv, Ollama

The commands from this book, in the order you meet them. Print it, or keep it open.

Command	What it does
`pwd · ls · ls -a`	Where am I · what is here · include hidden
`cd folder · cd .. · cd ~`	Go in · go up · go home
`cat f · head -20 f`	Print a file · print the top of a big one
`mkdir d · cp a b · mv a b · rm f`	Make · copy · move · delete (no undo)
`Ctrl-C · Ctrl-D · clear`	Stop it · end input · wipe the screen
`history | grep ollama`	Find that command you ran yesterday
`uv python install 3.12`	Install a Python of your own
`uv init --name my_agent`	Start a project
`uv add ollama · uv add --dev pytest`	Install and record · dev-only
`uv remove ollama`	Uninstall and un-record
`uv run python agent.py`	Run inside this project's environment
`uv sync`	Rebuild .venv from the lock file
`uv add --script s.py ollama · uv run s.py`	One-file script with its own deps
`uvx ruff check .`	Run a tool without installing it
`ollama serve · ollama ps`	Start the service · what is loaded
`ollama pull qwen2.5 · ollama run qwen2.5`	Download · chat in the terminal
`ollama list · ollama show m · ollama rm m`	What I have · details · delete

Cheatsheet: git and GitHub

The commands from this book, in the order you meet them. Print it, or keep it open.

Command	What it does
`git config --global user.name "..."`	Who you are (once per machine)
`git config --global init.defaultBranch main`	Match GitHub's default branch
`git init · git status`	Start tracking · what changed (run constantly)
`git add . · git commit -m "..."`	Stage · save with a message
`git log --oneline`	The history, one line each
`git diff · git diff --staged`	Unstaged changes · what I am about to commit
`git restore f · git restore --staged f`	Discard edits · un-stage them (safe)
`git commit --amend -m "..."`	Fix the last commit (before pushing)
`git revert <hash>`	Undo an old commit with a new one (safe)
`git reset --hard`	Destroy commits and changes — be slow with this
`git switch -c feature · git switch main`	New branch · back to main
`git branch · git branch -d feature`	List · delete a merged branch
`git merge feature`	Bring a branch into this one
`gh auth login`	Connect to GitHub (once)
`gh repo create x --private --source=. --push`	Create the repo and push it
`git push · git pull · git clone <url>`	Send · receive · copy a repo
`gh pr create --fill · gh pr view --web`	Open a pull request · see it

Cheatsheet: Uvicorn and Nginx

The commands from this book, in the order you meet them. Print it, or keep it open.

Command	What it does
`uv run uvicorn main:app --reload`	Dev: serve on :8000, restart on save
`uvicorn main:app --port 8010`	Use another port when 8000 is taken
`uvicorn main:app --host 0.0.0.0`	Reachable from other machines — mean it
`uvicorn main:app --workers 4`	Several processes (each with its own memory)
`uvicorn main:app --proxy-headers`	Trust X-Forwarded-* from your proxy
`--forwarded-allow-ips="127.0.0.1"`	...but only from the proxy you control
`main:app`	The object `app` inside the file main.py
`curl -N -X POST localhost:8000/chat/stream`	Watch a stream without client buffering
`curl localhost:8000/docs`	The API page FastAPI generates for you
`nginx -t`	Check the config before applying it
`nginx -s reload`	Apply a new config without dropping traffic
`proxy_pass http://127.0.0.1:8000;`	The one line that points Nginx at Uvicorn
`proxy_set_header X-Forwarded-For ...`	Tell your app who the real client is
`proxy_set_header X-Forwarded-Proto ...`	Tell your app the client used HTTPS

Glossary

Confused code usually starts as a confused word. These are the 22 that matter most.

Terminal / shell The window where you type commands.

Path The address of a file. A full path starts with / or ~.

Package Code written by someone else that you install, like ollama or fastapi.

Environment A separate set of packages for one project. Here, the .venv folder.

Lock file The exact versions you installed, so other people get the same ones.

Environment variable A setting kept outside your code, like GOOGLE_API_KEY.

Repository (repo) A folder whose history git is following.

Commit One save point, with a short note.

Staging area The changes you have chosen for your next commit.

Branch A separate line of commits. main is the one that should always work.

Remote / origin The copy on GitHub. origin is its usual name.

Push / pull / clone Send yours up · bring theirs down · make your first copy.

Pull request (PR) A branch asking to join main, with a place to discuss it.

Merge conflict Two branches changed the same lines. git stops and asks you to choose.

API A way for one program to call another program.

Endpoint / route One address in an API, such as /chat.

JSON Text that looks like a Python dict. Programs use it to exchange data.

Port / localhost A numbered door on a computer · your own computer, only you can reach it.

ASGI The agreement between an async server and a Python app: scope, receive, send.

ASGI server The program that owns the port and calls your app. Uvicorn is one.

Reverse proxy A server that takes public requests and passes them to yours. Nginx.

TLS / HTTPS The encryption behind the S in HTTPS. Usually handled by Nginx, not by your app.

Ready-to-start checklist

Tick every box before you start building. "Installed" is not the test — "I ran it and saw it work" is the test.

- [ ] uv --version prints a version.
- [ ] uv python list shows Python 3.11 or newer.
- [ ] You made a test project: uv init, uv add ollama, uv run python — all of them worked.
- [ ] ollama --version prints a version, and ollama ps runs without an error.
- [ ] ollama pull qwen2.5 finished, and ollama run qwen2.5 answered a question.
- [ ] Your editor opens the FOLDER, and its interpreter is the one inside .venv.
- [ ] git config --global user.name and user.email are both set.
- [ ] git config --global init.defaultBranch prints main.
- [ ] gh auth status says you are logged in to GitHub.
- [ ] You made one commit and pushed it to a private repo.
- [ ] Your .gitignore contains .env, .venv/ and __pycache__/ — and git status proves .env is invisible.
- [ ] You know what to do if you share a key by mistake: make a new one, then remove the old file.

THE .GITIGNORE BOX IS THE IMPORTANT ONE

If `git status` ever lists `.env`, stop and fix it before you push. That single line is the difference between a portfolio and a leaked key.