02

Setting Up Python

How a Python project is built — coming from Node

Chapter 2 · the tools, explained from scratch (with the npm words you already know)

02 The journey — five things to set up

'Setting up Python correctly' really means understanding five things. We'll take them one at a time, using the Node words you already know as a map.

Follow the arrows → by the end you can set up a project properly.

02

The one weird thing: no node_modules

Where do installed packages actually go?

NODE

'npm install' puts packages in a node_modules folder INSIDE each project. Every project keeps its own copy — automatic, and separate.

PYTHON

By default, packages install into ONE shared place used by that whole Python — NOT inside your project. There is no per-project folder.

The problem this creates: two projects that need DIFFERENT versions of the same package clash — nothing keeps them apart. Fixing that is what the next slide is about.

02

Which Python? You have several

Pick your own — never build on the one that came with the Mac

Already on your Mac

A 'system' Python, maybe a Homebrew one, and others. They're for the operating system — messing with them can break things.

What to do instead

Install your OWN Python (3.12 or 3.13) and point your project at that. Keep it separate from the system one.

The tool for this is pyenv — it installs and switches between Python versions (it's the nvm of Python). The modern 'uv' (two slides on) can also download and manage Python versions for you.

02

Virtual environments — the fix

A private box of packages, one per project

Your project

.venv (a private box)

Its own copy of Python + its own packages, kept apart from every other project.

It's Python's version of node_modules — a per-project folder holding just this project's packages. The one difference: you have to make it and switch it on yourself.

```
python -m venv .venv          # make the box
source .venv/bin/activate      # switch it on  (your prompt then shows (.venv) )
deactivate                    # switch it off
```

02

The #1 beginner mistake

Forgetting to switch the box on

If you forget to switch on the .venv, then 'pip install' dumps the packages into the SHARED / system Python instead — quietly polluting every other project and, worse, the operating system's Python.

The habit that saves you: before you install anything, glance at your prompt. If it doesn't show (.venv), stop and switch it on first.

02

The classic way: pip + requirements.txt

Fine to recognise, but the old way

pip

The classic installer — roughly Node's npm. It fetches and installs packages.

requirements.txt

A plain text list of the packages a project needs (a rough package.json, but just a list).

```
pip install -r requirements.txt    # install everything on the list
pip freeze > requirements.txt      # save the exact versions you have now
```

Weakness: it's only a list — no proper settings, and no 'lockfile' pinning the exact versions of everything. Modern tooling fixes that.

02

The modern way: uv + pyproject.toml

The 2026 default — fast, and does everything

uv
A fast, modern tool (written in Rust). It installs packages, makes and manages the .venv, and can even download Python — all in one, and very quick.

pyproject.toml
Python's package.json — one file that lists your dependencies and settings.

uv.lock
The lockfile — Python's package-lock.json. Pins the EXACT version of every package so installs are identical everywhere.

```
uv venv                  # make the .venv (finds/downloads Python)
uv add anthropic         # add a package (updates pyproject.toml + uv.lock)
uv sync                  # install exactly what the lockfile pins
uv run python main.py    # run inside the project — no 'activate' needed
uvx ruff check .         # run a tool once, without installing it (like npx)
```

02 The Node → Python cheat sheet

JavaScript / Node		Python (modern)
npm / pnpm	→	pip / uv
package.json	→	pyproject.toml
node_modules/	→	.venv/
package-lock.json	→	uv.lock
npx <tool>	→	uvx <tool>
eslint + prettier	→	ruff
tsc (types)	→	mypy / pyright

02

Gotcha: install name ≠ import name

What you install isn't always what you type to import

```
uv add python-dotenv   →   import dotenv
uv add beautifulsoup4  →   import bs4
uv add anthropic       →   import anthropic   (matches, this time)
```

There's no guaranteed match like npm's. When unsure, check the package's docs for the real import name.

Handy: 'python –m <tool>' runs a tool using THIS project's Python — so you never accidentally run the wrong one. That's why you see 'python –m venv', 'python –m pytest'.

02

How a good project is laid out

The 'src layout' — the mark of a serious repo

```
agent-demo/
├── pyproject.toml
├── uv.lock
├── .venv/                (git-ignored)
├── src/
│   └── agent_demo/     the package
│       ├── __init__.py   marks it a package
│       ├── client.py     a module
│       └── tools/        a sub-package
└── tests/
```

A module = one .py file.
A package = a folder of modules.

__init__.py marks a folder as a package. Often empty — its job is just 'this folder is a package'.

Putting code under src/ forces your tests to import it the same way a real user would — catching 'works on my machine' bugs.

02

Imports: by name, not by file path

No 'import ./file' like in Node

```
Absolute import  (preferred — clear and works everywhere)

from agent_demo.tools.search import web_search
```

```
Relative import  (from inside the same package; the dots mean 'here')

from .tools.search import web_search      # .  = this package
from ..client import Client               # .. = the parent package
```

You import by a dotted NAME (agent_demo.tools.search), not a file path. There's no index.js — __init__.py is the nearest thing, but it means 'this folder is a package', not 'the entry point'.

02

The quality tools you'll always see

Two of them; both worth having

ruff

The ESLint + Prettier of Python, rolled into one and very fast (Rust). It finds problems AND formats your code.

```
uvx ruff check .    (find issues)
uvx ruff format .   (tidy the code)
```

It replaced the older trio: flake8 + black + isort.

mypy / pyright

Python's tsc — the type checkers. They read your type hints and flag mismatches WITHOUT running the code:

greet(42) → error: expected str, got int

pyright (from Microsoft, powers VS Code) is stricter; mypy is the long-standing classic. Either one is a good sign.

02 Reading & judging a repo in 10 seconds

HEALTHY (open pyproject.toml first)

✔ a pyproject.toml with pinned dependencies
✔ a lockfile (uv.lock) committed
✔ the src/ layout with a real package
✔ ruff + mypy set up
✔ .env kept out of git (.gitignore)

RED FLAGS

✗ only a bare requirements.txt, or none
✗ versions unpinned — not reproducible
✗ .venv committed to git
✗ all the logic living in notebooks
✗ API keys hardcoded in the source

The teaching — in GenAI work, drifting dependencies quietly break everything. So the setup around the code tells you whether to trust it. Open pyproject.toml first; it says more than the code does.