Managing environments and dependencies with uv



Mohamed Echout

Senior Technical Trainer

@MEchout | www.mohamedechout.com

How uv manages virtual environments

The traditional venv ritual

Every Python project starts with the same five steps

PowerShell

```
$ python -m venv .venv    # Create the env
$ .\.venv\Scripts\Activate.ps1    # Activate it
$ pip install requests    # Now you can install
$ python my_script.py    # Now you can run
$ deactivate    # Don't forget on the way out
```



What's actually in .venv?

```
.venv/
  Scripts/
       python.exe
       pip.exe
       Activate.ps1
  Lib/
       site-packages/
              requests/
              httpx/
  pyvenv.cfg
```

Same format whether:

- **pip**
- **poetry**
- **uv** created it

uv Python venv format



An isolated Python

A copy or symlink of the interpreter you pinned



`site-packages`

Where installed packages actually live



`pyvenv.cfg`

Tells Python this is a **`venv`** and which one

uv's model: uv run does it for you

After running

```
$ uv run main.py
```



uv does

- Reads `.python-version`, finds the right Python
- Creates `.venv` if it doesn't exist
- Syncs packages from `uv.lock` if needed
- Runs `main.py` in that environment

Same task, two workflows



Traditional

- `$ python -m venv .venv`
- `$ .\.venv\Scripts\Activate.ps1`
- `(.venv) $ pip install requests`
- `(.venv) $ python main.py`
- `(.venv) $ deactivate`



With uv

- `$ uv add requests`
- `$ uv run main.py`

uv use cases



Long REPL sessions



Tools that expect a **venv**



Standalone non-project **envs**

Demo

Creating and using environments with uv venv and uv run

Demo

Adding, updating, and removing dependencies

Demo

Working with dependency groups and optional extras

Demo

Migrating a pip plus requirements.txt project to uv