uv

Version 0.13.6 · verified 2026-11-02

One tool for Python versions, projects, lockfiles, scripts and CLI tools — the uv workflow from init to Docker and CI.

Contents

Mental model 2

Install and Python versions 2

Projects: init, add, run 3

Lockfile and sync 5

Dependency groups 6

Scripts with inline metadata 7

Tools with uvx 9

Build and publish 10

Docker and CI 11

Migrating from pip and Poetry 13

Further reading 14

Mental model

`pyproject.toml` is the project, `uv.lock` is the reproducible truth, and `.venv` is a disposable cache of the lockfile. uv replaces pip, venv, pip-tools, pipx and pyenv behind one binary, and every project command (`uv run`, `uv sync`) re-locks and re-syncs before it runs. You stop activating environments and start asking uv to run things.

Install and Python versions

Command	Does
`uv python install 3.15`	Download a managed CPython
`uv python pin 3.15`	Write `.python-version`
`uv python list`	Installed and downloadable
`uv self update`	Upgrade a standalone uv

Install uv with the standalone installer, Homebrew, or `pipx install uv`. uv downloads a missing Python

automatically when a project asks for it, so `uv python install` is optional. It looks for a `.python-version` file in the working directory and its parents.

```
curl -LsSf https://astral.sh/uv/install.sh | sh
uv python install 3.15
uv python pin 3.15    # Pinned `.python-version`
uv run python -V      # Python 3.15.2
```

> **Note:** `uv self update` works only for the standalone installer. Upgrade a Homebrew or pipx install with that tool instead.

Projects: init, add, run

Command	Does
`uv init app`	New project with `pyproject.toml`
`uv add httpx`	Add, lock and install
`uv remove httpx`	Remove, lock and uninstall

Command	Does
`uv run <cmd>`	Run inside the synced `.venv`
`uv tree`	Show the dependency tree

`uv add` writes the requirement into `[project].dependencies`, updates `uv.lock` and installs into `.venv` in one step. `uv run` locks and syncs first, so a teammate's freshly pulled change is installed before your command starts.

```
uv init demo && cd demo
uv add 'httpx>=0.28'
uv run demo          # Hello from demo!
uv run python -c "import httpx"
```

> **Gotcha:** Since uv 0.12, `uv init` creates a `src/<name>/` package with a `[build-system]` (backend `uv_build`), not a top-level `main.py`. Use `uv init --bare` or `--no-package` for the old shape.

Lockfile and sync

Flag	Lockfile stale?
(none)	Re-lock silently
`--locked`	Fail with an error
`--frozen`	Use it as is, never check
`uv lock --check`	Same check as `--locked`
`uv lock --upgrade-package x`	Bump one package

`uv.lock` is cross-platform and belongs in Git. It is “up to date” when it satisfies `pyproject.toml`; new releases on PyPI never make it stale, so upgrades happen only when you run `uv lock --upgrade` or `--upgrade-package`. `uv sync` is exact: it removes packages the lockfile does not list. `uv run` is inexact and only adds.

```
uv lock --check    # exit 1 if pyproject changed
uv sync --locked   # CI: install, never re-lock
uv lock --upgrade-package httpx
```

> **Gotcha:** `--frozen` does not compare the lockfile with `pyproject.toml`. Add a dependency, forget `uv lock`, and `uv run --frozen` runs without it — the `import` fails at runtime. Use `--locked` in CI so the drift is an error.

Dependency groups

Flag	Effect
`uv add --dev pytest`	Add to the `dev` group
`uv add --group lint ruff`	Add to a named group
`--no-dev`	Skip `dev`
`--group lint`	Include `lint` too
`--all-groups`	Include every group

Groups live in the standard `[dependency-groups]` table (PEP 735) and are never published with the package. Only `dev` is synced by default; change that with `tool.uv.default-groups`.

```
[dependency-groups]
dev = ["pytest>=9.2.0"]
lint = ["ruff>=0.17.3"]

[tool.uv]
default-groups = ["dev", "lint"]
```

> **Gotcha:** Because `uv sync` is exact, `uv sync --no-dev` also uninstalls the `lint` group you installed a minute ago. Name every group you need on each sync.

Scripts with inline metadata

Command	Does
`uv init --script f.py`	New script with a metadata block
`uv add --script f.py rich`	Declare a dependency
`uv run f.py`	Run in a throwaway env

Command	Does
`uv run --with rich f.py`	Add a dependency ad hoc
`uv lock --script f.py`	Write `f.py.lock`

A single-file script declares its own Python and dependencies in a `# /// script` comment block (PyPA inline script metadata). `uv run` builds a cached environment for it and ignores any surrounding project.

```
#!/usr/bin/env -S uv run --script
# /// script
# requires-python = ">=3.15"
# dependencies = ["httpx<1"]
# ///
import httpx

print(httpx.__version__)  # 0.28.1
```

> **Note:** The `dependencies` key must be present even when the list is empty. With the shebang, `chmod +x f.py` and run `./f.py`.

Tools with uvx

Command	Does
`uvx ruff check`	Run a tool in a temp env
`uvx ruff@0.17.3 check`	Pin a version
`uvx --from httpie http`	Command name differs from package
`uv tool install ruff`	Put `ruff` on `PATH`
`uv tool upgrade --all`	Upgrade installed tools

`uvx` is an alias for `uv tool run`: it replaces `pipx run`, and `uv tool install` replaces `pipx install`. Each tool gets its own isolated environment.

```
uvx ruff@0.17.3 --version   # ruff 0.17.3
uvx --from httpie http --version
```

```
uv tool install ruff
uv tool update-shell        # add bin dir to PATH
```

> **Gotcha:** `uvx pytest` or `uvx mypy` cannot import your project's dependencies, because the tool runs in its own env. Use `uv run pytest` for tools that must see the project.

Build and publish

Command	Does
`uv build`	sdist and wheel into `dist/`
`uv version --bump minor`	Edit `[project].version`
`uv publish`	Upload `dist/*` to PyPI
`uv build --no-sources`	Build as PyPI users will

`uv publish` reads a token from `--token` or `UV_PUBLISH_TOKEN`, or uses PyPI trusted publishing from

CI with no stored secret. Add `--index <name>` to publish to a private index defined in `pyproject.toml`.

```
uv version --bump minor   # demo 0.1.0 => 0.2.0
uv build --no-sources
uv publish --dry-run
```

> **Tip:** Build with `--no-sources` before release. `tool.uv.sources` (local paths, Git pins) applies only during development, so a wheel that builds with them can fail for everyone else.

Docker and CI

Setting	Why
`COPY --from=ghcr.io/astral-sh/uv:0.13.6`	Pinned uv binary
`uv sync --locked`	Fail on a stale lockfile
`--no-install-project`	Cache the dependency layer

Setting	Why
`UV_COMPILE_BYTECODE=1`	Faster container start
`uv cache prune --ci`	Smaller CI cache

Install dependencies in their own layer from `uv.lock` alone (`--frozen`, since `pyproject.toml` is not there yet), then copy the source and sync again with `--locked`. Add `.venv` to `.dockerignore`. In GitHub Actions, `astral-sh/setup-uv` with `enable-cache: true` persists the cache keyed on the lockfile.

```
FROM python:3.15-slim
COPY --from=ghcr.io/astral-sh/uv:0.13.6 /uv /bin/
WORKDIR /app
ENV UV_COMPILE_BYTECODE=1 UV_LINK_MODE=copy
RUN --mount=type=cache,target=/root/.cache/uv \
  --mount=type=bind,source=uv.lock,target=uv.lock
\
  uv sync --frozen --no-install-project --no-dev
COPY . .
```

```
RUN --mount=type=cache,target=/root/.cache/uv \
  uv sync --locked --no-dev
CMD ["/app/.venv/bin/demo"]
```

> **Warning:** `COPY --from=...uv:latest` changes under you on every build. Pin the tag (or digest) so a uv release cannot break a deploy.

Migrating from pip and Poetry

From	To
`pip install -r requirements.txt`	`uv add -r requirements.txt`
`pip install x` in a venv	`uv pip install x`
`pip-compile`	`uv pip compile` or `uv lock`
`[tool.poetry.dependencies]`	`[project].dependencies`
CI needs `requirements.txt`	`uv export --format requirements.txt`

`uv pip` is a drop-in, faster pip that needs no project, but it requires a virtual environment unless you pass `--system`. For the project workflow, `uv init --bare` then `uv add -r requirements.txt` moves your requirements into `pyproject.toml` and locks them.

```
uv init --bare
uv add -r requirements.txt
uv add --dev -r requirements-dev.txt
uv export --format requirements.txt \
  --no-hashes --no-dev > requirements.txt
```

> **Gotcha:** uv reads dependencies only from the standard `[project]` and `[dependency-groups]` tables. A Poetry-only `[tool.poetry]` section is ignored and `uv lock` resolves an empty project — move the requirements first.

Further reading

- uv documentation

- Locking and syncing
- Managing dependencies
- Running scripts
- Using uv in Docker
- PEP 735 — Dependency groups
- Inline script metadata specification