Python Tools: UV

New Python Package Manager

uv replaces venv and pip

- **Speed Comparison**: uv is 10-100x faster than pip!
- Written in Rust
- uv replaces: venv, pip, pipx, and Python version management
- **Manages Python versions** automatically
- The tool that is used by AI MCP tools.

Installation

```
# Install uv
pip3 install uv

# UV examples
# Create virtual environment
uv venv

# Install packages (much faster!)
uv pip install pandas numpy requests
```

Depending on the system, the `pip` command may be used instead.

- Make sure `python` or `python3` is in your path.
- When the pip/pip3 command is not available, try `python -m pip` or `python3 -m pip` instead.

Where the UV is Located

macOS/Linux:

```
~/.local/bin
```

Windows:

```
%APPDATA%\uv\bin
```

Be sure that `~/.local/bin` is in the path.

You can also use a symlink to copy the link to the path (not recommended, but useful sometimes).

```
# In the uv directory
sudo ln -s ~/.local/bin/uv /usr/local/bin/uv
```

Problems That UV Solves

- Manage multiple Python versions — `uv python`
- Simplify virtual envs — `uv env`
- Avoid per-project package conflicts — `uv run`
- Do one-off, temporary installs (like `npx`) — `uvx`
- Globally install & manage CLI tools (like `npm -g`) — `uv tool install <pkg>` / `uv tool run <cmd>`
- Manage syncing environments - `uv sync`

Quick Examples

```
# Install and pick a Python version
uv python install 3.12
uv python pin 3.12

# Create & activate an env (auto-activation in shells supported)
uv env create
uv run python -V

# One-off run (no permanent install)
uv run ruff --version

# Install a global CLI tool
uv tool install ruff
uv tool run ruff check .
```

Manage multiple Python versions — `uv python`

UV automatically manages Python versions!

- **Downloads** Python versions as needed
- **Switches** between versions per project
- **No need** for pyenv, conda, or manual installs

💡 Think of it as: **Python version vending machine**

Installing Python Versions

```
# List available Python versions
uv python list

# Install specific Python version
uv python install 3.12
uv python install 3.11
uv python install 3.10

# List installed versions
uv python list --only-installed
```

UV downloads and manages Python for you!

Installed Directory

With `uv python`, Python is installed in the `~/share/uv/python` directory.

- With the `uv python find` command, we can find the exact location of the Python.

```
chos5@NKU023R7042 temp> uv python install 3.12
Installed Python 3.12.10 in 3.01s
 + cpython-3.12.10-macos-aarch64-none

> uv python find 3.12
~/.local/share/uv/python/cpython-3.12.10-macos-aarch64-none/bin/python3.12
```

We can use this version of Python globally with a symbolic link.

```
# In the uv directory
sudo ln -s $(uv python find 3.12) /usr/local/bin/python312
```

Simplify virtual envs — `uv env`

- Creates isolated Python environments (like `python -m venv`)
- Same activation method: `source <env>/bin/activate`
- **Key advantage: 10–30× faster** than Python's built-in `venv`

```
# Speed comparison (approximate)
time python -m venv test_env    # ~2–3 seconds
time uv venv test_env           # ~0.1–0.2 seconds
```

Feature	uv venv	Python venv
Speed	⚡ Ultra-fast (Rust)	🐌 Slower (Python)
Python Discovery	Auto-finds Python versions	Requires explicit path
Built-in	No - External tool	Yes - Built into Python3
Dependencies	Requires uv installation	No extra installation

uv venv vs Python venv

uv venv (Smart Discovery)

```
# Automatically finds Python 3.11
# If Python 3.11 isn't in uv's cache,
# it's auto-installed before the venv is created.
uv venv --python 3.11 # .venv directory
my venv myenv         # myenv directory

# Works with python, python3, py, etc.
uv venv myenv --python 3.12
```

Python venv (Manual Path)

```
# Need exact Python executable
python3.11 -m venv myenv

# Or full path on some systems
/usr/bin/python3.11 -m venv myenv
```

Directory Structure: Almost Identical

`uv venv` creates `.venv/` directory, but we can specify any directory with `uv venv DIRECTORY`.

```
.venv/
├── bin/            # Scripts (Linux/macOS)
├── Scripts/        # Scripts (Windows)
├── lib/            # Python packages
├── include/        # Header files
└── pyvenv.cfg      # Configuration
```

Both create the same structure with the same files!

Avoid per-project package conflicts — `uv run`

Creating Projects

With UV, creating and using an isolated project with venv is straightforward.

```
# Method 1: Specify during project creation
uv init my-project --python 3.12
cd my-project

# Method 2: Set Python version for existing project
cd existing-project
echo "3.11" > .python-version
uv venv  # Uses Python 3.11

# Method 3: Direct venv creation
uv venv --python 3.10
```

`uv init` command

It creates a project skeleton.

- Don't forget to make a venv: If you run uv venv in a directory with a .python-version (or in its parent), uv uses that Python version for the venv.

```
uv init my-project --python 3.12
cd my-project
uv venv
```

```
my-project/
├── .python-version     # Specifies Python version
├── pyproject.toml      # Project configuration & dependencies
├── README.md
└── .venv/             # Virtual environment (created when needed)
    ├── bin/
    ├── lib/
    └── pyvenv.cfg
```

Different Projects, Different Python

```
# Legacy project (needs older Python)
uv init legacy-app --python 3.9
cd legacy-app
uv add "django==3.2"  # Old Django version

# Modern project (latest Python)
uv init modern-app --python 3.12
cd modern-app
uv add "fastapi"  # Latest FastAPI

# Data science project (stable Python)
uv init data-project --python 3.11
cd data-project
uv add "pandas" "jupyter"
```

Python Version Priority (How UV Chooses)

1. **Command line**: `--python 3.12`
2. **`.python-version` file**: Project-specific
3. **`pyproject.toml`**: `requires-python = ">=3.11"`
4. **System default**: Whatever's available

Best Practice: Always specify in `.python-version`

`uv add` vs `pip`

Similarities

- Install Python packages from PyPI
- Support version specifiers (`requests==2.31.0`)
- Can install multiple packages at once
- Use installed packages in your project immediately

Differences

Feature	`uv add`	`pip`
Dependency Mgmt	Updates `pyproject.toml` & `uv.lock` automatically	No lock file by default
Speed	Much faster (parallel, prebuilt wheels)	Slower
Isolation	Works with `uv` venvs seamlessly	Requires manual `venv` handling
One-off Install	`uv run <pkg>` without install	No direct equivalent
Cache	Shared global cache for re-use	Builds per venv

Example

```
# uv
uv add requests
uv remove requests                  # Remove dependency

# pip
pip install requests
pip uninstall requests
```

The pyproject.toml is updated automatically.

```
[project]
name = "my-project"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.12"
dependencies = [
    "requests>=2.32.4",
]
```

Avoid per-project package conflicts — `uv run`

```
uv run python script.py

# uv add pytest
# uv add jupyter

uv run pytest tests/
uv run jupyter notebook
```

- **Purpose:** Run commands in **current project's environment**
- **Like:** Activated virtual environment
- **Think of it as:** "Run this in my current project"
- **Available:** Only within project directory

How `uv run python` Works

1. **Finds or creates** a project-specific virtual environment based on `pyproject.toml` and `uv.lock`.
2. **Temporarily activates** this isolated environment automatically.
3. **Runs your Python command or script inside** this managed environment with exact dependencies.
4. **Ends the session** without manual activation or cleanup—the environment persists for reuse.

Key benefit: Seamlessly manages environment syncing and activation on-the-fly, eliminating manual steps and avoiding dependency conflicts.

- **Purpose:** Run commands in **current project's environment**
- **Like:** Activated virtual environment
- **Think of it as:** "Run this in my current project"
- **Available:** Only within project directory

1. Create a project and add a dependency

```
uv init my-project
cd my-project
uv add requests
```

2. Create a script that uses the dependency

```
echo 'import requests; print("Works!")' > script.py
```

3. Compare the two approaches:

```
python script.py           # Might fail (no requests available)
> ModuleNotFoundError: No module named 'requests'

uv run python script.py    # Works (uses project environment)
# PAHT/my-project/.venv/bin/python script.py
> Works!
```

Do one-off, temporary installs (like `npx`) — `uvx`

Think of `uvx` like Node.js's `npx` but for Python tools!

- **Run Python tools** without permanent installation
- **Isolated execution** - no conflicts with your system
- **Automatic cleanup** - tools are cached but don't clutter your environment
- **Fast execution** - reuses cached installations

💡 **Perfect for:** Testing tools, one-time scripts, CI/CD pipelines

`uvx` vs Traditional Installation

Traditional Way (Permanent Install)

```
# Install tool globally (clutters system)
pip install black
black my_file.py

# Or in a virtual environment (manual setup)
python -m venv temp_env
source temp_env/bin/activate
pip install black
black my_file.py
deactivate
rm -rf temp_env  # Manual cleanup
```

With `uvx` (Temporary)

```
# One command - automatic isolation & cleanup
uvx black my_file.py
```

How `uvx` Works Behind the Scenes

1. **Check cache**: Is the tool already downloaded?
2. **Create isolated environment**: Temporary virtual environment
3. **Install tool**: Only if not cached
4. **Execute command**: Run the tool with your arguments
5. **Keep cache**: Tool stays cached for future use
6. **Clean isolation**: Environment is temporary

Usage Examples

```
# 1. Format a Python file
echo "x=1;y=2;print(x+y)" > messy.py
uvx black messy.py
cat messy.py

# 2. Create a simple web server
echo "<h1>Hello World</h1>" > index.html
uvx http.server 8000
# Visit http://localhost:8000

# 3. Analyze a requirements file
echo "requests==2.25.0\ndjango==3.0" > requirements.txt
uvx safety check -r requirements.txt
```

Common `uvx` Patterns

Testing Different Versions

```
# Test with different Black versions
uvx black==22.0.0 my_file.py
uvx black==23.0.0 my_file.py
uvx black==24.0.0 my_file.py
```

One-off Scripts

```
# Run a script that needs specific packages
uvx --with pandas --with matplotlib python analyze_data.py
```

Quick Utilities

```
# Quick file serving
uvx http.server 3000
# Quick code formatting
uvx autopep8 --in-place *.py
```

Useful UVX Tools

Code Formatting & Linting

```
# Format Python code with Black
uvx black .
# Lint code with Ruff
uvx ruff check .
# Sort imports with isort
uvx isort .
```

Web Development

```
# Create a Django project without installing Django
uvx django-admin startproject mysite
# Run HTTP server for static files
uvx http.server 8000
```

Package Management Tools

```
# Check package vulnerabilities
uvx safety check
# Analyze dependencies
uvx pipdeptree
# Check outdated packages
uvx pip-review
```

Data Science & Analysis

```
# Quick data analysis
uvx pandas-profiling data.csv
# Convert notebooks
uvx nbconvert notebook.ipynb --to html
# Run Jupyter lab temporarily
uvx jupyterlab
```

`uvx` vs `npx` Comparison

Feature	`uvx` (Python)	`npx` (Node.js)
Purpose	Run Python packages	Run npm packages
Installation	Temporary execution	Temporary execution
Caching	Yes (reuses cached)	Yes (reuses cached)
Isolation	Automatic venv	Node modules
Cleanup	Automatic	Automatic

Globally install & manage CLI tools (like `npm -g`) — `uv tool install <pkg>` / `uv tool run <cmd>`

Don't install everything globally - be selective

- **Use `uv tool`** for tools you'll use repeatedly
  - **Use `uv tool run`** for experimentation
- **Use `uv run`** for project-specific commands

uv tool (Persistent Tools)

```
uv tool install black
uv tool uninstall black
```

- **Purpose:** Install tools **permanently** in isolated environments
- **Like:** `pipx` - each tool gets its own space
- **Think of it as:** "Add this to my workshop toolbox"
- **Available:** Everywhere, across all projects

How UV Organizes Tools

Tools are installed in `~/.local/share/uv/tools`.

```
~/.local/share/uv/tools/
├── black/                        # Each tool gets its directory
│   ├── bin/
│   │   └── black                # Executable
│   ├── lib/
│   │   └── python3.11/
│   │       └── site-packages/
│   └── pyvenv.cfg
└── cookiecutter/
    ├── bin/
    ├── lib/
    └── pyvenv.cfg
```

notice that `~/.local/bin/black` is linked to `~/.local/share/uv/tools/black`

~/.local/bin/black (Direct execution) vs uv tool run black (UV-managed execution)

They are the same in most cases: functionally identical: The `~/.local/bin/black` is a **wrapper script created by UV** that does essentially the same thing as `uv tool run black`. Both:

- Use the same isolated environment
- Have the same dependencies
- Run the same version of Black

When They Might Differ

1. Environment Resolution

```
# UV tool run: Always uses UV's managed environment
uv tool run black my_file.py

# Direct call: Uses the wrapper, but could theoretically be overridden
~/.local/bin/black my_file.py
```

2. Path Issues

```
# If ~/.local/bin is not in PATH:
~/.local/bin/black my_file.py  # Works (full path)
black my_file.py               # Might not work
uv tool run black my_file.py   # Always works
```

3. Tool Management

```
# UV knows about the tool
uv tool list                    # Shows black
uv tool upgrade black          # Updates black

# Direct execution doesn't involve UV's management layer
```

When to Use `uvx` vs `uv tool install`

Use `uvx` for:

- **One-time tasks**: Formatting, testing, analysis
- **Trying tools**: Before deciding to install permanently
- **CI/CD scripts**: Temporary tool usage
- **Different tool versions**: Testing multiple versions

Use `uv tool install` for:

- **Daily tools**: Tools you use regularly
- **CLI utilities**: Commands you want in PATH
- **Development workflow**: Part of your regular setup

Manage syncing environments - `uv sync`

(Old way) Using requirements.txt and freeze

```
# Using uv (fast & modern)
uv add -r requirements.txt
uv pip freeze > requirements.txt

# Using pip (classic way)
pip install -r requirements.txt
pip freeze > requirements.txt
```

(New Way) UV Lock Files Like package-lock.json

```
# Generate lock file (exact versions)
uv lock

# Install from lock file (reproducible builds)
uv sync

# Update dependencies
uv lock --upgrade
```

`uv.lock` ensures everyone gets **exact same versions**

`uv lock` and `pyproject.toml`

File	Purpose	Who Should Have It	Action	Edit?
pyproject.toml	Declare broad project dependencies	Everyone	Add/remove dependencies	Yes
uv.lock	Lock exact dependency versions and environment	Everyone (shared)	Ensure reproducible installs	No (auto-managed)

Work Process Example

1. Develop and specify dependencies in `pyproject.toml`.
2. Run commands `uv add <package>` or manually edit `pyproject.toml`.
3. Run `uv sync` or `uv lock` to update `uv.lock` with resolved versions.
4. Commit both files (`pyproject.toml` and `uv.lock`) to your version control.
5. Other developers or CI systems run `uv sync` to install the exact locked environment.
6. Avoid using `pip install -r` when using uv with lockfiles; use `uv sync` instead.