Python Tools: venv, pip & uv

Classic Python Package Manager

- venv creates isolated Python environments
- pip installs packages
- uv is a fast, modern replacement for pip tools with better dependency resolution

Installing Python

Before using venv/pip/uv, you need Python installed

- For Mac users: Python is already installed on your Mac, but it is almost always an old version; so it is recommended to install a newer version of Python for developing applications.

Method 1: Official Website (Recommended for Beginners)

Visit: https://python.org/downloads/

- **Latest stable version** - Always recommended
- **Includes pip** - Package manager comes bundled
- **Adds to PATH** - Check "Add Python to PATH" during installation

For students: Download the latest version!

Method 2: Package Managers (Advanced)

Windows (using Chocolatey)

```
choco install python
```

macOS (using Homebrew)

```
brew install python
```

Ubuntu/Debian Linux

```
sudo apt update
sudo apt install python3 python3-pip python3-venv
```

Method 3: Windows Store (Windows Only)

Microsoft Store → Search "Python"

- Easy one-click installation
- Automatic updates
- No PATH configuration needed
- **Good for:** Windows beginners

Verify Installation

Check if Python is installed:

```
python --version
# Should output: Python 3.11.5 (or similar)

# On some systems, use:
python3 --version
```

Check if pip is installed:

```
pip --version
# Should output: pip 23.2.1 (or similar)

# On some systems, use:
pip3 --version
```

If neither of them is available:

```
python –m pip

python3 –m pip
```

If the commands work → You're ready to go! 🎉

venv (virtual environment)

- **Built-in Python tool** for creating isolated environments
- Each project gets its own Python packages
- Prevents dependency conflicts between projects
- **Think of it as:** separate toolboxes for each project

Why Use Virtual Environments?

Without venv:

- Project A needs Django 3.2
- Project B needs Django 4.1
- **Conflict!**

With venv:

- Project A: `venv_A` → Django 3.2
- Project B: `venv_B` → Django 4.1
- **No conflicts!**

Essential Commands

```
# Create virtual environment
python -m venv myproject_env

# Activate (Windows)
myproject_env\Scripts\activate
# Activate (macOS/Linux)
source myproject_env/bin/activate

# Install packages
pip install requests pandas
# Deactivate
deactivate
```

pip (Python Package Installer)

Each package installation is isolated.

```
# Install packages
pip install requests pandas numpy
# Install from requirements file
pip install -r requirements.txt
# List installed packages
pip list
# Show package info
pip show pandas
# Save dependencies
pip freeze > requirements.txt
```

pipx (Install and Run Python Applications)

```
pipx install black
pipx install cookiecutter
```

- **Installs into:** Isolated environments per tool
- **Purpose:** Command-line applications system-wide
- `pipx` is similar to `npm -g`, but without dependency conflict.

The Dependency Conflict Problem

With pip (risky):

```
pip install black   # Needs click==8.0
pip install flake8  # Needs click==7.0
# ❌ Conflict! One breaks the other
```

- All tools share the same site-packages
- Last install wins → version conflicts

With pipx (safe):

```
pipx install black   # Own venv → click==8.0
pipx install flake8  # Own venv → click==7.0
# ✅ No conflicts!
```

- Each tool gets its virtual environment
- Dependencies never overlap

How pipx Works

1. Creates a venv for each tool (~/.local/pipx/venvs/<tool>)
2. Installs tool + dependencies inside that venv
3. Symlinks executable into ~/.local/bin
4. $PATH makes it runnable anywhere

📦 pipx vs venv/pip (Node.js Analogy)

pipx

- Installs **Python CLI tools** globally
- Each tool in its **own isolated venv**
- Can run anywhere without activation

💡 Like:

```
npm install -g <tool>
```

> unlike npm, pipx has no shared dependency conflicts

venv + pip

- Create a project-specific environment
- Install dependencies inside it
- Must activate before use

💡 Like:

```
npm install <tool>
```

> similar to venv+pip, node.js dependencies live in that project's node_modules

Adding Multiple Packages

```
# Using pip
pip install -r requirements.txt
```

The requirements.txt example:

```
# requirements.txt
requests==2.31.0
numpy>=1.26.0
pandas
flask~=3.0

- == → exact version
- >= → minimum version
- ~= → compatible version (e.g., 3.0.x)
- No operator → latest version available
```

Freezing Python Dependencies

What is Freezing?

- "Freezing" means capturing the exact versions of packages installed in your environment.
- This ensures consistency when sharing your project or deploying it.
- The output of `pip freeze` is a list of packages and their versions.

Why Freeze Dependencies?

- Guarantees your app uses the identical package versions.
- Avoids "It works on my machine" problems.
- Critical for reproducible deployments and collaboration.

Using `pip freeze`

1. Activate your virtual environment.
2. Run:

```
pip freeze > requirements.txt
```

3. This writes all installed package versions to the `requirements.txt` file.

Installing from a Freeze File

- To recreate the environment elsewhere:

```
pip install –r requirements.txt
```

- This installs the exact versions recorded.

Distinction: Explicit vs All Dependencies

- `pip freeze` lists *all* installed packages, including sub-dependencies.
- Sometimes, you want to list only explicitly installed packages.

Explicit vs All Dependencies Example

- Example `requirements.in` (explicit packages):

```
Django
requests
```

- Example freeze output (includes dependencies):

```
Django==4.1.7
asgiref==3.6.0
sqlparse==0.4.3
requests==2.31.0
urllib3==1.26.18
```

Best Practice

1. `requirements.in`

- This file lists only explicit packages you care about, one per line:

```
Django
requests
```

You edit this file by hand or using a script.

```
echo "flask" >> requirements.in
```

2. Generate All Dependencies

- Use `pip-compile requirements.in` (from `pip-tools`) to create a comprehensive `requirements.txt` that includes every package needed, with exact versions — both explicit and implicit.

```
pip-compile requirements.in
```

- The resulting `requirements.txt` is meant for **installation and sharing**.

3. Why Two Files?

File	What goes in it?	How to use?
`requirements.in`	Only packages you write by name	Edit this file directly
`requirements.txt`	All dependencies (freeze), pinned	Install via `pip install -r requirements.txt`

You should:

- Keep `requirements.in` under version control, as it documents the explicit choices about your project's dependencies.
- Use `requirements.txt` to ensure deterministic, reproducible installs.

Typical Workflow

1. Add a new package to `requirements.in`:

   ```
   echo "flask" >> requirements.in
   ```

2. Re-run:

   ```
   pip-compile requirements.in
   ```

   This updates `requirements.txt` with Flask and all new dependencies.

3. Install everything:

   ```
   pip install -r requirements.txt
   ```

What if You Only Use `requirements.in`

- If you try to install directly from `requirements.in`:

```
pip install -r requirements.in
```

- You'll **miss version pinning for all sub-dependencies**, risking inconsistent environments over time or across machines.

Summary

Command	Purpose
`pip freeze > requirements.txt`	Freeze all installed packages
`pip install -r requirements.txt`	Install from freeze file
`pip-compile requirements.in`	Generate freeze from explicit list