**🚀** UV Package Manager Cheat Sheet

**What is UV?**

UV is a super-fast Python package manager and environment manager. It works like pip but is much faster!

# Installing UV

pip install uv

Check if it's installed correctly:

uv --version

**🌟** Creating a New Project Locally

## Quick Project Setup with uv init

uv init my\_cool\_project

This creates a folder with your project and sets up a virtual environment automatically!

Then:

cd my\_cool\_project

**Activate the environment:**

* Windows: .venv\Scripts\activate
* Mac/Linux: source .venv/bin/activate

**🔄** Working with a Cloned Repository

1. **Clone the repository**:
2. git clone https://github.com/username/repo-name.git
3. cd repo-name
4. **Create virtual environment:**
5. uv venv .venv
6. **Activate virtual environment**:
   * Windows: .venv\Scripts\activate
   * Mac/Linux: source .venv/bin/activate
7. **Install dependencies**:

**From requirements file:**

uv pip install -r requirements.txt

**OR directly from the project itself (if it has setup.py or pyproject.toml):**

uv pip install .

**📦** Common Package Commands

**Installing Packages with UV Add (Recommended)**

uv add package\_name

This installs the package AND saves it to pyproject.toml automatically!

**Installing Multiple Packages with UV Add**

uv add package1 package2 package3

**Installing Dev Dependencies with UV Add**

uv add -d pytest black

This adds packages to the [tool.project.optional-dependencies.dev] section in pyproject.toml

**Installing Packages with Pip (Traditional Way)**

uv pip install package\_name

**Note**: This doesn't update your pyproject.toml file!

**Installing Specific Versions**

uv pip install package\_name==1.2.3

**Installing from Requirements File**

uv pip install -r requirements.txt

**Uninstalling Packages**

# Remove a package and update pyproject.toml:

uv remove package\_name

# Remove multiple packages:

uv remove package1 package2

# Remove development dependencies:

uv remove -d pytest

# Uninstall without updating pyproject.toml (not recommended):

uv pip uninstall package\_name

## Viewing Dependency Tree

# See all dependencies and their relationships:

uv tree

# See dependencies for a specific package:

uv tree package\_name

**Listing Installed Packages**

uv pip list

## Syncing Your Environment

# Update your environment to match pyproject.toml:

uv sync

# Update with development dependencies included:

uv sync -d

# Sync and include exact versions specified:

uv sync --strict

# Sync without upgrading (only install new packages):

uv sync --no-upgrade

The uv sync command is super important! It makes sure your virtual environment matches exactly what's in your pyproject.toml file:

* It installs missing packages
* It upgrades or downgrades packages to match specified versions
* It removes packages that aren't in pyproject.toml

**Example use cases:**

* After pulling changes from GitHub, run uv sync to get any new dependencies
* After editing pyproject.toml manually, run uv sync to update your environment
* Run uv sync -d to include development dependencies too

**🏠 Virtual Environment Commands**

**Creating a Virtual Environment**

uv venv env\_name

**Activating a Virtual Environment**

Windows:

env\_name\Scripts\activate

Mac/Linux:

source env\_name/bin/activate

**⚙️ Special Installation Cases**

**Install in Development Mode (Your Own Project)**

uv pip install -e .

**Install with Extra Features**

uv pip install package\_name[extra\_feature]

**Install Development Dependencies**

uv pip install -r requirements-dev.txt

**📋 Complete Workflows**

**Creating a New Project:**

# Create new project with automatic setup

uv init my\_project

# Go to project folder

cd my\_project

# Activate environment

source .venv/bin/activate # or .venv\Scripts\activate on Windows

# Install needed packages (saved to pyproject.toml automatically)

uv add flask

uv add -d pytest

# Start coding!

**Working with a Cloned Project:**

# Clone the repository

git clone https://github.com/username/project.git

# Go to project folder

cd project

# Create virtual environment

uv venv .venv

# Activate environment

source .venv/bin/activate # or .venv\Scripts\activate on Windows

# If project has requirements.txt:

uv pip install -r requirements.txt

# OR if project has pyproject.toml:

uv sync # Install all dependencies from pyproject.toml

uv sync -d # Include development dependencies too

# OR if project has setup.py:

uv pip install .

# Add new dependencies (saved to pyproject.toml):

uv add new\_package

# Add development dependencies:

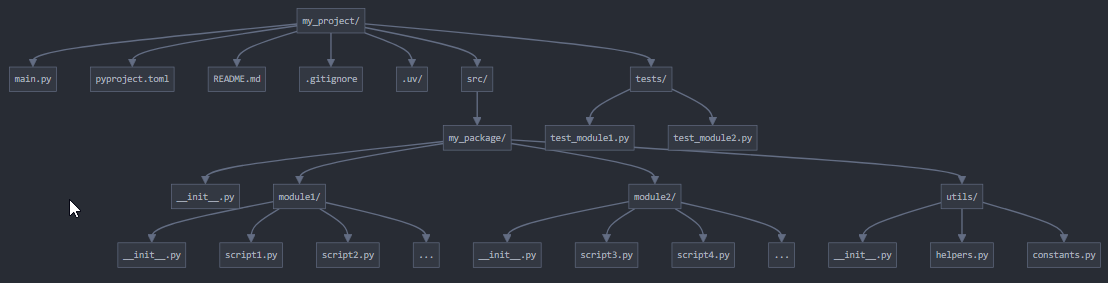
uv add -d black pytest

# Start coding!

**🚀 Tips and Tricks**

* Always activate your virtual environment before installing packages
* Use uv add instead of uv pip install to automatically save dependencies to pyproject.toml
* Use uv remove instead of uv pip uninstall to remove from pyproject.toml as well
* Use uv sync after pulling changes to update your environment to match pyproject.toml
* Use uv tree to see dependency relationships and potential conflicts
* Use uv add -d package\_name for development dependencies
* Use uv pip install -e . for your own projects you're developing
* For speed, install multiple packages in one command
* Use uv init for the fastest project setup

# Python project structure – with UV



This folder structure follows Python best practices for a moderately complex project with about 20 scripts:

**Root Directory (my\_project/)**

* **main.py**: Entry point that runs the entire application. Keeps your core logic simple by importing and orchestrating functionality from other modules.
* **pyproject.toml**: Configuration file for UV, dependencies, and project metadata.
* **.uv/**: UV virtual environment directory (managed by UV itself, not edited manually).
* **README.md**: Project documentation.
* **.gitignore**: Prevents unnecessary files from being tracked by git.

**Source Directory (src/my\_package/)**

Using the src/ layout is considered a best practice because:

1. It prevents accidental imports from the project root
2. It ensures your installed package and development environment behave identically
3. It makes the package namespace cleaner

Inside the package, I've organized code by functionality:

* **module1/**, **module2/**: Logical groupings of related scripts
* **utils/**: Common utilities shared across modules

**Tests Directory (tests/)**

Separate test files that mirror your package structure.

## Import Management

To avoid import problems, follow these guidelines:

**1. Make your package installable**

python -c "import my\_package; print(my\_package.\_\_file\_\_)" # run it in folder dir to see if the package holding all scripts is returned properly‘



If not returning the package found, there is an issue with python path, nad needs to manually be setup for the project:  
set PYTHONPATH=C:\Users\Mr Black\Desktop\ticket\_count\src

Afterwards, install in dev mode as seen bellow!!!!

2. Install in development mode

A black and grey rectangular object

AI-generated content may be incorrect.

This makes your package importable from anywhere, solving most import headaches

Modify toml file:  
A screen shot of a computer program

AI-generated content may be incorrect.

[tool.setuptools]

package-dir = { "" = "src" }

packages.find = { where = ["src"] }

### **Explanation:**

* packages.find = { where = ["src"] }: This line tells setuptools to automatically discover packages inside the src/ directory.
* package-dir = { "" = "src" }: This is still necessary to tell setuptools that the main package directory is src/.

With this setup, setuptools will automatically find all Python packages (directories containing \_\_init\_\_.py) inside src/ without needing to manually list them in packages.

1. Use absolute imports in your code

A screenshot of a computer

AI-generated content may be incorrect.

**4. Set up proper \_\_init\_\_.py files**

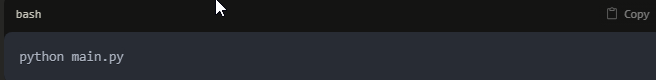
Each \_\_init\_\_.py should expose the public API of that module:

A screenshot of a computer program

AI-generated content may be incorrect.

## Running Scripts

**Running the entire project**



### Running individual scripts for testing

There are two good approaches:

**1. Make scripts runnable individually**

Add this to the end of each script:

A screenshot of a computer program

AI-generated content may be incorrect.

**2. Create test scripts**

For more formal testing, create corresponding test files in the tests/ directory and use pytest:

A screenshot of a computer

AI-generated content may be incorrect.