Python UV: The Ultimate Guide to the Fastest Python Package Manager

Learn how to use UV, the fastest Python package manager in 2025. Discover 10x faster dependency management, virtual environments, and seamless migration from pip, Poetry, and Conda.

Jan 9, 2025 · 11 min read

Bex Tuychiev
Bex is a Top 10 AI writer on Medium and a Kaggle Master with over 15k followers. He loves writing detailed guides, tutorials, and notebooks on complex data science and machine learning topics.

TOPICS

Python

What is Python UV?

UV is a modern, high-performance Python package manager and installer written in Rust. It serves as a drop-in replacement for traditional Python package management tools like `pip`, offering significant improvements in speed, reliability, and dependency resolution.

This tool represents a new generation of Python package managers, designed to address common pain points in the Python ecosystem such as slow installation times, dependency conflicts, and environment management complexity. UV achieves this through its innovative architecture and efficient implementation, making it 10-100 times faster than traditional package managers.

Key features that make UV stand out:

- Lightning-fast package installation and dependency resolution
- Compatible with existing Python tools and workflows
- Built-in virtual environment management
- Support for modern packaging standards
- Reliable dependency locking and reproducible environments
- Memory-efficient operation, especially for large projects

Whether working on small personal projects or managing large-scale Python applications, UV provides a robust and efficient solution for package management. In this tutorial, we will cover all essential aspects of UV so you can start using it immediately.

The Difference Among UV, Poetry, PIP, Conda, and virtualenv

The first question developers often ask before switching to a new tool is "How does it compare to the one I am already using?". In the Python dependency and project management arena, these four tools are already the most common:

- PIP
- Poetry
- Conda
- virtualenv

Let's compare UV with each of these tools to help you decide if UV is the right choice for your needs before diving into the details.

UV vs. PIP and `virtualenv`

PIP and `virtualenv` have been the traditional tools for Python package management and virtual environment creation. While they get the job done, UV offers several compelling advantages:

- Speed: UV's Rust implementation makes it significantly faster than PIP for package installation and dependency resolution, often completing tasks in seconds that would take PIP minutes.
- Integrated environment management: While `virtualenv` handles only environment creation and PIP only handles package management, UV combines both functionalities in a single tool, streamlining the workflow.

UV maintains full compatibility with PIP's ecosystem while addressing its key limitations. It can use the same `requirements.txt` files and package indexes, making migration seamless. The key differences are:

- **Performance**: UV's parallel downloads and optimized dependency resolver make it 10-100x faster than PIP for large projects.
- **Memory usage**: UV uses significantly less memory than PIP during package installation and dependency resolution.
- **Error handling**: UV provides clearer error messages and better conflict resolution when dependencies clash.
- **Reproducibility**: UV's lockfile approach ensures consistent environments across different systems, something not guaranteed with basic `requirements.txt` files.

While PIP and virtualenv remain viable options, UV's modern architecture and combined functionality make it an attractive alternative for developers seeking better performance and a more streamlined workflow. The ability to drop UV into existing projects without disrupting established processes makes it particularly appealing for teams looking to gradually modernize their Python development toolchain.

UV vs. Conda

People who don't use PIP and `virtualenv` usually turn to Conda and they have good reasons:

- Conda provides a complete package management solution that handles not just Python packages but also system-level dependencies
- It excels at managing complex scientific computing environments with packages like NumPy, SciPy, and TensorFlow
- Conda environments are more isolated and reproducible across different operating systems
- It has built-in support for different Python versions and can switch between them easily
- The Anaconda distribution comes with many pre-installed scientific packages, making it convenient for data scientists

But even hardcore Conda users should consider switching to UV for a variety of reasons. UV's lightning-fast package installation and dependency resolution can dramatically speed up environment setup compared to Conda's sometimes sluggish performance. Its minimal resource footprint means less memory usage and quicker startups. UV also integrates seamlessly with existing Python packaging standards and tools, making it easier to work with the broader Python ecosystem. For projects that don't require Conda's non-Python package management, UV provides a more streamlined, efficient solution that can significantly improve development workflows.

UV vs. Poetry

I was a Conda user for almost three years but after trying out Poetry a couple of times, I never touched turtle-speed Conda again. Just when I was getting comfortable with Poetry, I came across UV and it seems to promise nearly the same things as Poetry:

- **Dependency management**: Both tools handle package dependencies and virtual environments effectively
- **Project structure**: Both provide tools for initializing and structuring Python projects
- **Lock files**: Both generate lock files to ensure reproducible environments
- **Package publishing**: Both support publishing packages to PyPI
- **Modern tooling**: Both represent modern approaches to Python project management

However, the defining feature of UV is its blazing-fast speed and minimal resource usage. While Poetry is a significant improvement over traditional tools, UV takes performance to another level because of its Rust implementation. Also, UV's compatibility with existing Python packaging means it can work alongside other tools like `pip`, offering flexibility that Poetry's more opinionated approach sometimes lacks.

Here is a table summarizing the differences we've just covered:

Feature	UV	PIP + virtualenv	Conda	Poetry
Implementation	Rust	Python	Python	Python
Speed	10-100x faster than pip	Baseline	Slower than pip	Faster than pip
Memory Usage	Very efficient	Higher	High	Moderate
Environment Management	Built-in	Separate tools needed	Built-in	Built-in
Dependency Resolution	Fast, modern resolver	Basic	Comprehensive	Modern resolver
Non-Python Packages	No	No	Yes	No
Lock Files	Yes	No (basic requirements.txt)	Yes	Yes
Project Structure	Yes	No	No	Yes
Package Publishing	Yes	Yes (with twine)	Yes	Yes
Compatibility	Works with existing pip ecosystem	Standard Python tool	Own ecosystem	More opinionated approach
Error Handling	Clear error messages	Basic	Good	Good
Resource Footprint	Minimal	Moderate	Heavy	Moderate
Scientific Computing Focus	No	No	Yes	No
Cross-platform Consistency	Yes	Limited	Excellent	Good

If you think UV is worth switching, then read on.

Getting Started With UV For Python Projects

In this section, we cover how to start a project from scratch using UV. We will discuss how to migrate from existing projects to UV in a later section.

Installing UV

UV can be installed system-wide using cURL on macOS and Linux:

```
$ curl -LsSf https://astral.sh/uv/install.sh | sudo sh
```

Explain code POWERED BY datalab

And with Powershell on Windows (make sure you run Powershell with administrator privileges):

```
$ powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps` i
```

Explain code POWERED BY datalab

UV is available via Homebrew as well:

```
$ brew install uv
```

Explain code POWERED BY datalab

A PIP install is supported but it is not recommended:

```
$ pip install uv  # Make sure you have a virtual environment activated
```

Explain code POWERED BY datalab

Afterward, you can verify the installation by running `uv version`:

```
$ uv version
uv 0.4.25 (97eb6ab4a 2024-10-21)
```

Explain code POWERED BY datalab

Initializing a new project

Working on projects is the core part of the UV experience. You start by initializing an empty project using the `uv init` command:

```
$ uv init explore-uv
Initialized project explore-uv at /Users/bexgboost/projects/explore-uv
```

Explain code POWERED BY datalab

The command will immediately create a new `explore-uv` directory with the following contents:

```
$ cd explore-uv
$ tree -a
.
├── .gitignore
├── .python-version
├── README.md
├── hello.py
└── pyproject.toml
```

Explain code POWERED BY datalab

Git is automatically initialized and main git-related files like `.gitignore` and an empty `README.md` are generated. `.python-version` file contains the Python version used for the project, while `pyproject.toml` serves as the main configuration file for project metadata and dependencies. A sample `hello.py` file is also created to help you get started quickly.

You can learn more about creating projects from the UV documentation.

Adding initial dependencies to the project

UV combines the environment creation and dependency installation into a single command - `uv add`:

```
$ uv add scikit-learn xgboost
Using CPython 3.9.20 interpreter at: /opt/homebrew/opt/python@3.9/bin/python3.9
Creating virtual environment at: .venv
Resolved 6 packages in 1.78s
⠙ Preparing packages... (2/5)
Prepared 5 packages in 1m 23s
Installed 5 packages in 45ms
 + joblib==1.4.2
 + numpy==2.0.2
 + scikit-learn==1.5.2
 ...
```

Explain code POWERED BY datalab

The first time you run the `add` command, UV creates a new virtual environment in the current working directory and installs the specified dependencies. On subsequent runs, UV will reuse the existing virtual environment and only install or update the newly requested packages, ensuring efficient dependency management.

Another important process that happens for every `add` command is resolving dependencies. UV uses a modern dependency resolver that analyzes the entire dependency graph to find a compatible set of package versions that satisfy all requirements. This helps prevent version conflicts and ensures reproducible environments. The resolver considers factors like version constraints, Python version compatibility, and platform-specific requirements to determine the optimal set of packages to install.

UV also updates the `pyproject.toml` and `uv.lock` files after each add command. Here is what the TOML file looks like after installing Scikit-learn and XGBoost:

```
$ cat pyproject.toml
[project]
name = "explore-uv"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.9"
dependencies = [
    "scikit-learn>=1.5.2",
    "xgboost>=2.0.3",
]
```

Explain code POWERED BY datalab

To remove a dependency from the environment and the `pyproject.toml` file, you can use the `uv remove` command. It will uninstall the package and all its child-dependencies:

```
$ uv remove scikit-learn
```

Explain code POWERED BY datalab

We will discuss dependency management in more detail in a later section.

Running Python scripts with UV

Once you install the necessary dependencies, you can start working on your Python scripts as usual. UV provides a few different ways to run Python code:

To run a Python script directly, you can use the `uv run` command followed by your script name instead of the usual `python script.py` syntax:

```
$ uv run hello.py
```

Explain code POWERED BY datalab

The `run` command ensures that the script is executed inside the virtual environment UV created for the project.

Managing Python Versions in UV

Managing Python versions is a key part of dependency management. UV provides simple commands to control which Python version you use. Let's explore how to use the `uv python` command.

Listing existing Python versions

Since it is common for systems to have Python already installed, UV can discover these existing installations by default. For example, to list all the Python versions UV detects on your system, run the following command:

```
$ uv python list --only-installed
cpython-3.13.0-macos-aarch64-none    /opt/homebrew/opt/python@3.13/bin/python3.1
cpython-3.12.7-macos-aarch64-none    /opt/homebrew/opt/python@3.12/bin/python3.1
cpython-3.12.5-macos-aarch64-none    /Users/bexgboost/miniforge3/bin/python3.12
cpython-3.12.5-macos-aarch64-none    /Users/bexgboost/miniforge3/bin/python3 ->
cpython-3.12.5-macos-aarch64-none    /Users/bexgboost/miniforge3/bin/python -> p
cpython-3.11.10-macos-aarch64-none   /opt/homebrew/opt/python@3.11/bin/python3.1
cpython-3.11.7-macos-aarch64-none    /Users/bexgboost/.local/share/uv/python/cpy
cpython-3.10.15-macos-aarch64-none   /opt/homebrew/opt/python@3.10/bin/python3.1
cpython-3.9.20-macos-aarch64-none    /opt/homebrew/opt/python@3.9/bin/python3.9
cpython-3.9.6-macos-aarch64-none     /Library/Developer/CommandLineTools/usr/bin
```

Explain code POWERED BY datalab

The command correctly detected my Conda and Brew-installed Python versions.

Changing Python versions for the current project

You can switch Python versions for your current UV project at any point as long as the new version satisfies the specifications in your `pyproject.toml` file. For example, the following file requires Python versions 3.9 and above:

```
...
requires-python = ">=3.9"
```

Explain code POWERED BY datalab

This means you can change the Python version in `.python-version` file to any version above, like 3.11.7. Afterwards, call `uv sync`.

The command first checks against existing Python installations. If the requested version isn't found, UV downloads and installs it inside the `/Users/username/.local/share/uv/python` path. UV also creates a new virtual environment inside the project directory, replacing the old one.

This new environment doesn't have the dependencies listed in your `pyproject.toml` file, so you have to install them with the following command:

```
$ uv pip install -e .
```

Explain code POWERED BY datalab

Note that sometimes `uv` commands may raise `Permission Denied` errors. In those cases, be sure to use the `sudo` command if you are on macOS or Linux or run your command prompt with administrator privileges if you are on Windows. Even better solution would be to change the ownership of the UV home directory to the user:

```
$ sudo chown -R $USER ~/.local/share/uv  # macOS or Linux
```


To learn more about managing Python versions with UV, refer to the documentation.

What Are UV Tools And How to Use Them?

Some Python packages are exposed as command-line tools like `black` for code formatting, `flake8` for linting, `pytest` for testing, `mypy` for type checking, etc. UV provides two special interfaces to manage these packages:

1. Using `uv tool run` :

```
$ uv tool run black hello.py
```

Explain code POWERED BY datalab

2. Using the shorter and more convenient `uvx` command:

```
$ uvx black hello.py
```

Explain code POWERED BY datalab

When these commands are run, UV creates a temporary virtual environment in its cache. The requested tool is installed and run from there. In other words, you can use command-line tools without installing them in the project's virtual environment, resulting in faster execution and cleaner project dependencies.

Key points about the tool run interfaces:

- Works with any Python package that provides command-line tools like flake8, mypy, black or pytest
- Cached environments are automatically cleaned up when clearing UV's cache
- New cached environments are created on-demand when needed
- Perfect for occasional use of development tools

Read the UV Tools section of the documentation to learn more about these interfaces.

What Are Lock Files in UV?

Lock files (`uv.lock`) are an essential part of dependency management in UV. When you run `uv add` commands to install dependencies, UV automatically generates and updates a `uv.lock` file. This lock file serves several critical purposes:

- It records the exact versions of all dependencies and their sub-dependencies that were installed.
- It ensures reproducible builds by "locking" dependency versions across different environments.
- It helps prevent "dependency hell" by maintaining consistent package versions.
- It speeds up installations since UV can use the locked versions instead of resolving dependencies again.

UV manages the lock file automatically - you don't need to manually edit it. The lock file should be committed to version control to ensure all developers use the same dependency versions.

The Difference Between Lock Files and `requirements.txt`

While both lock files and `requirements.txt` serve to track dependencies, they have distinct purposes and use cases. Lock files contain detailed information about exact package versions and their complete dependency tree, ensuring consistent environments across development. Requirements.txt files are simpler, typically listing only direct dependencies, and are widely supported across Python tools.

Lock files are essential for development to maintain reproducible builds and prevent dependency conflicts. Requirements.txt files are better suited for deployment scenarios or when sharing code with users who may not use UV. They're also necessary for compatibility with tools and services that don't support UV's lock file format.

You can maintain both files by using UV's lock file for development while generating a `requirements.txt` for deployment. To generate a `requirements.txt` from a UV lock file, use the following command:

```
$ uv export -o requirements.txt
```

Explain code POWERED BY datalab

This creates the text file with pinned versions based on your lock file, making it easy to share your project's dependencies in a standard format while still benefiting from UV's advanced dependency management during development.

You can learn more about managing lock files from the documentation.

Advanced Dependency Management With UV

In this section, we will explore more sophisticated methods to manage dependencies in UV. We will learn how to update dependencies, make them optional, or make them part of a dependency group.

Updating dependencies

In long-term projects, it is common to update the packages you are using to bring the most up-to-date features to the table. Or sometimes, a package you are using introduces breaking changes and you want to ensure that version doesn't get installed accidentally in your environment. The `add` command can be used again in these and any other scenario where you need to change the constraints or versions of existing dependencies.

1. Installing the latest version of a package:

```
$ uv add requests
```

Explain code POWERED BY datalab

2. Installing a specific version:

```
$ uv add requests=2.1.2
```

Explain code POWERED BY datalab

3. Change the bounds of a package's constraints:

```
$ uv add 'requests<3.0.0'
```

Explain code POWERED BY datalab

4. Make a dependency platform-specific:

```
$ uv add 'requests; sys_platform="linux"'
```

Explain code POWERED BY datalab

Adding optional dependencies

Optional dependencies are packages that are not required for the core functionality of your project but may be needed for specific features. For example, Pandas has an `excel` extra and a `plot` extra to avoid the installation of Excel parsers and matplotlib unless someone explicitly requires them. Optionals are usually installed with the `pip install pandas[plot, excel]` syntax.

With UV, this syntax is slightly different. First, ensure that the core Pandas package is installed:

```
$ uv add pandas
```

Explain code POWERED BY datalab

Then, add its optional dependencies:

```
$ uv add pandas --optional plot excel
```

Explain code POWERED BY datalab

Once they are resolved, they will be listed in your `pyproject.toml` with the following format:

```
[project]
name = "explore-uv"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.9"
dependencies = [
    "pandas>=2.2.3",
    "requests>=2.32.3",
]
[project.optional-dependencies]
plot = [
    "excel>=1.0.1",
    ...
]
```

Explain code POWERED BY datalab

Dependency groups

Dependency groups allow you to organize your dependencies into logical groups, such as development dependencies, test dependencies, or documentation dependencies. This is useful for keeping your production dependencies separate from your development dependencies.

To add a dependency to a specific group, use the `--group` flag:

```
$ uv add --group group_name package_name
```

Explain code POWERED BY datalab

Then, users will be able to control which groups to install using the `--group`, `--only-group`, and `--no-group` tags.

Switching From PIP and Virtualenv to UV

Migrating from PIP and virtualenv to UV is straightforward since UV maintains compatibility with existing Python packaging standards. Here's a step-by-step guide to make the transition smooth:

1. Converting an existing virtualenv project

If you have an existing project using `virtualenv` and pip, start by generating a `requirements.txt` file from your current environment if you haven't already:

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

Explain code POWERED BY datalab

Then, create a new UV project in the same directory:

```
$ uv init .
```

Explain code POWERED BY datalab

Finally, install the dependencies from your requirements file:

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

Explain code POWERED BY datalab

2. Replacing common pip/virtualenv commands

Here's a quick reference for replacing common pip and virtualenv commands with their UV equivalents:

pip/virtualenv command	UV equivalent
python -m venv .venv	uv venv
pip install package	uv add package
pip install -r requirements.txt	uv pip install -r requirements.txt
pip uninstall package	uv remove package
pip freeze	uv pip freeze
pip list	uv pip list

After migration, you can safely remove your old virtualenv directory and start using UV's virtual environment management. The transition is typically seamless, and you can always fall back to `pip` commands through UV's `pip` compatibility layer if needed.

Conclusion

UV represents a significant advancement in Python package management, offering a modern, fast, and efficient alternative to traditional tools. Its key advantages include:

- Blazing fast performance with 10-100x speed improvements over pip
- Seamless integration with existing Python packaging standards
- Built-in virtual environment management
- Efficient dependency resolution and lock file support
- Low memory footprint and resource usage

Whether you're starting a new project or migrating an existing one, UV provides a robust solution that can significantly improve your Python development workflow. Its compatibility with existing tools makes it an easy choice for developers looking to modernize their toolchain without disrupting their current processes.

As the Python ecosystem continues to evolve, tools like UV demonstrate how modern technologies like Rust can enhance the development experience while maintaining the simplicity and accessibility that Python developers value.

If you like to learn more about dependency management or Python in general, refer to these additional sources:

- Python Tutorial for Beginners
- PIP Python Tutorial: Definitive Guide
- Developing Python Packages Course
- Top 9 Anaconda Alternatives for Python Environment Management
- Data Scientist in Python | Learn Python for Data Science
- Python Poetry: Modern And Efficient Python Environment And Dependency Management

Python UV FAQs

Is UV faster than pip for Python package management?

Yes, UV is significantly faster than pip, offering 10-100x speed improvements for package installation and dependency resolution. This performance boost is achieved through UV's Rust implementation and optimized parallel download capabilities. For large projects especially, tasks that take minutes with pip can be completed in seconds with UV.

Can I use UV with existing pip requirements.txt files?

What are the advantages of UV over Poetry and Conda?

Does UV support virtual environments?

AUTHOR
Bex Tuychiev

I am a data science content creator with over 2 years of experience and one of the largest followings on Medium. I like to write detailed articles on AI and ML with a bit of a sarcastic style because you've got to do something to make them a bit less dull. I have produced over 130 articles and a DataCamp course to boot, with another one in the making. My content has been seen by over 5 million pairs of eyes, 20k of whom became followers on both Medium and LinkedIn.

TOPICS

Python

Top DataCamp Courses

TRACK

Python Data Fundamentals

0 min

Grow your data skills, discover how to manipulate and visualize data, and apply advanced analytics to make data-driven decisions.

See Details → Start Course

COURSE

Intermediate Python

4 hr 1.3M

Level up your data science skills by creating visualizations using M

See Details →

See More →

Related

BLOG

10 Python Packages to Add to Your Data Science Stack in 2022

BLOG

Top 9 Anaconda Alternatives for Python Environment...

TUTORIAL

Python Poetry: Modern And Efficient Python Environment...

See More →

Courses

Data Science Roadmap

DATA COURSES

Python Courses

R Courses

SQL Courses

Power BI Courses

Tableau Courses

Alteryx Courses

Azure Courses

AWS Courses

Google Sheets Courses

Excel Courses

AI Courses

Data Analysis Courses

Data Visualization Courses

Machine Learning Courses

Data Engineering Courses

Probability & Statistics Courses

DATALAB

Get Started

Pricing

Security

Documentation

CERTIFICATION

Certifications

Data Scientist

Data Analyst

Data Engineer

SQL Associate

Power BI Data Analyst

Tableau Certified Data Analyst

Azure Fundamentals

AI Fundamentals

RESOURCES

Resource Center

Upcoming Events

Blog

Code-Alongs

Tutorials

Docs

Open Source

RDocumentation

Book a Demo with DataCamp for Business

Data Portfolio

PLANS

Pricing

For Students

For Business

For Universities

Discounts, Promos & Sales

Expense DataCamp

DataCamp Donates

FOR BUSINESS

Business Pricing

Teams Plan

Data & AI Unlimited Plan

Customer Stories

Partner Program

ABOUT

About Us

Learner Stories

Careers

Become an Instructor

Press

Leadership

Contact Us

DataCamp Español

DataCamp Português

DataCamp Deutsch

DataCamp Français

SUPPORT

Help Center

Become an Affiliate