perplexity

Managing Python Dependency Challenges: The uv Revolution

Python dependency management has long been a source of frustration for developers at all experience levels. This report explores the complex challenges of Python dependency management, examines how the new uv tool offers comprehensive solutions, and provides a beginner-friendly guide to using uv for standalone scripts. As AI-generated code becomes more common, tools like uv that automate dependency management become increasingly valuable, allowing developers to focus on code rather than configuration.

The Python Dependency Management Challenge

Python's popularity stems from its simplicity and vast ecosystem of packages. However, this abundance creates significant dependency management challenges that can derail productivity and cause frustration.

The Core Problems

The dependency management challenge in Python has multiple dimensions, all contributing to what developers often call "dependency hell":

Complexity of Dependency Chains: When you install a package like requests, you're actually installing numerous nested dependencies. As your project grows, these interdependencies create a complex web that becomes increasingly difficult to manage[1]. Each dependency may have its own version requirements for shared libraries, leading to conflicts.

Version Incompatibilities: Different packages often require different versions of the same dependency. For example, numpy >=1.22.0 requires python >=3.8, making it impossible to use that version with Python 3.7[2]. These conflicts become more common as projects grow.

Reproducibility Issues: Without proper dependency management, code that works on one machine fails on another. This leads to the dreaded "but it works on my machine" problem and creates a brittle development environment where developers become afraid to update anything[3].

Environment Isolation: Python lacks built-in isolation between projects, requiring developers to create and manage virtual environments manually. This adds cognitive overhead and technical complexity[4].

Missing Features: Traditional Python package management lacks features common in other ecosystems, such as automatically defining lock files, parallel dependency installation, and creating standalone executables[4].

The Human Factor

The most challenging aspect of dependency management isn't technical but human. As Dominick M. notes, "The more libraries you rely on, the more you're at the mercy" of their compatibility and maintenance[1]. This creates a situation where developers spend more time troubleshooting environments than actually writing code.

Traditional Solutions and Their Limitations

Several tools have emerged to address these challenges, each with their own strengths and limitations:

pip and requirements.txt

The standard approach uses pip with a requirements.txt file to list dependencies. While familiar, this approach:

- Doesn't handle transitive dependencies well
- Lacks built-in lock file capability for exact reproducibility
- Requires separate virtual environment management
- Can be slow for large dependency sets

pipx

Pipx focuses on isolating Python applications:

"pipx is a tool to help you install and run end-user applications written in Python. It's roughly similar to macOS's brew, JavaScript's npx, and Linux's apt." [5]

While excellent for installing command-line tools, pipx isn't designed for project-level dependency management.

Poetry

Poetry offers more comprehensive dependency management:

"I use Poetry extensively for dependency management and also to publish and releasing new versions of various Python libraries." [6]

Poetry provides lock files and better dependency resolution but has a steeper learning curve and slower performance compared to newer tools.

Conda

Conda takes a different approach:

"An environment is a directory that contains a specific collection of packages that you have installed. For example, you may have one environment with NumPy 1.7 and its dependencies, and another environment with NumPy 1.6 for legacy testing." [7]

Conda is powerful but can be resource-intensive and uses its own package repository system that differs from PyPI.

Introducing uv: The New Standard for Python Management

Uv represents a significant advancement in Python tooling, addressing many longstanding issues with dependency management.

What Makes uv Different?

Uv is a high-speed package and project manager for Python written in Rust. It integrates multiple functionalities into a single tool, offering a comprehensive solution for managing Python projects[8]. Key features include:

- **Speed**: Written in Rust, uv provides significantly faster dependency resolution and installation
- **Integrated tooling**: Combines virtual environment management, dependency resolution, script execution, and more
- **Python version management**: Can install and manage different Python versions automatically
- **Project management**: Handles the full lifecycle of a Python project
- **Script execution**: Runs standalone scripts with automatic dependency management

Setting Up uv: A Complete Guide

Installation

Uv can be installed through multiple methods:

Standalone installer (recommended)

```
# For macOS/Linux
$ curl -LsSf https://astral.sh/uv/install.sh | sh

# For Windows PowerShell
$ powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Package managers

```
# HomeBrew (macOS/Linux)
$ brew install uv

# WinGet (Windows)
$ winget install --id=astral-sh.uv -e

# Scoop (Windows)
$ scoop install main/uv
```

```
# Using pipx (any platform)
$ pipx install uv
```

Python Environment Management with uv

Installing and Managing Python Versions

Uv simplifies Python version management by automatically installing the required Python versions as needed[9]:

```
# Install the latest Python version
$ uv python install

# Install a specific Python version
$ uv python install 3.12

# Install multiple Python versions
$ uv python install 3.11 3.12

# View available and installed Python versions
$ uv python list
```

This functionality eliminates the need for separate tools like pyenv, streamlining your workflow.

Editor and Notebook Integration

VSCode/Cursor Integration

VS Code can detect and use environments created by uv[10]. To integrate uv with VS Code:

1. Install the Python extension in VS Code
2. Create a project with `uv init`
3. Open the project folder in VS Code
4. Select the Python interpreter from the virtual environment created by uv (typically in `.venv`)

VS Code will automatically use this environment, providing proper completion and linting based on your project's dependencies.

Jupyter Notebook Integration

Uv integrates seamlessly with Jupyter for data science and exploratory programming[11]:

```
# Start Jupyter Lab with access to the project's virtual environment
$ uv run --with jupyter jupyter lab
```

For a more tailored setup with a dedicated kernel:

```
# Add ipykernel as a development dependency
$ uv add --dev ipykernel

# Create the kernel for your project
$ uv run ipython kernel install --user --env VIRTUAL_ENV $(pwd)/.venv --name=project

# Start Jupyter Lab
$ uv run --with jupyter jupyter lab
```

When creating a new notebook, select your project kernel to access all project dependencies.

Writing Standalone Scripts with uv: The Tutorial

One of uv's most powerful features is its ability to run standalone Python scripts with automatic dependency management. This is especially valuable when working with AI-generated code that may include various dependencies.

Basic Script Execution

For a script without dependencies, running with uv is simple:

```
$ uv run script.py
```

Adding Dependencies to Scripts

Uv supports inline dependency specifications[12]. Here's how to add them to your script:

```
# /// script
# requires-python = ">=3.8"
# dependencies = [
#   "requests",
#   "pandas",
# ]
# ///

import requests
import pandas as pd

# Your code here
```

You can then run this script with:

```
$ uv run script.py
```

Uv will automatically:

1. Create a virtual environment if needed
2. Install the required dependencies

3. Run the script in that environment
4. Cache the environment for faster future execution

Creating Executable Scripts

To make a script directly executable[13]:

1. Add a uv shebang at the top of your script:

```
#!/usr/bin/env -S uv run

# /// script
# requires-python = ">=3.8"
# dependencies = [
#   "requests",
# ]
# ///

import requests
# Rest of your code
```

2. Make the script executable:

```
$ chmod +x script.py
```

3. Run directly:

```
$ ./script.py
```

Step-by-Step: Building a Weather CLI Script

Let's create a simple weather script that demonstrates uv's power:

1. Create a file named `weather.py`:

```
#!/usr/bin/env -S uv run

# /// script
# requires-python = ">=3.8"
# dependencies = [
#   "requests",
#   "rich",
# ]
# ///

import requests
from rich.console import Console
from rich.table import Table

def get_weather(city):
    api_key = "demo_key"  # Replace with actual API key
    url = f"https://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&unit
    response = requests.get(url)
    return response.json()

def display_weather(weather_data):
    console = Console()

    if weather_data.get("cod") != 200:
        console.print(f"[bold red]Error: {weather_data.get('message', 'Unknown error')}[/
        return

    table = Table(title=f"Weather in {weather_data['name']}")

    table.add_column("Condition", style="cyan")
    table.add_column("Value", style="green")

    table.add_row("Temperature", f"{weather_data['main']['temp']}°C")
    table.add_row("Feels like", f"{weather_data['main']['feels_like']}°C")
    table.add_row("Humidity", f"{weather_data['main']['humidity']}%")
    table.add_row("Weather", weather_data['weather'][^0]['description'].capitalize())

    console.print(table)

if __name__ == "__main__":
    import sys

    if len(sys.argv) < 2:
        print("Please provide a city name. Example: ./weather.py London")
        sys.exit(1)

    city = sys.argv[^1]
    weather_data = get_weather(city)
    display_weather(weather_data)
```

2. Make the script executable:

```
$ chmod +x weather.py
```

3. Run the script:

```
$ ./weather.py London
```

On the first run, uv will install the dependencies (requests and rich). For subsequent runs, uv will reuse the cached environment, making execution nearly instantaneous.

Creating a CLI Tool

For more complex applications that you want to install globally, uv offers an elegant path[14]:

```
# Create a new CLI project
$ uv init --package my-cli-tool
```

```
# Change to the project directory
$ cd my-cli-tool

# Add dependencies
$ uv add requests rich

# Edit your code in src/my_cli_tool/
```

Then install globally:

```
$ uv tool install . -e
```

Now the command is available anywhere on your system:

```
$ my-cli-tool
```

Advanced Features for Production

Dependency Groups

Uv supports separating dependencies into logical groups[15]:

```
# Add a development dependency
$ uv add --dev pytest

# Add a dependency to a specific group
$ uv add --group lint ruff
```

This creates a clean separation between production, development, testing, and other dependencies.

Lock Files and Reproducibility

Uv automatically generates lock files that precisely capture your environment, ensuring reproducibility across machines and deployments[15].

Building and Publishing

Uv can also build and publish packages to PyPI, supporting the full lifecycle of Python projects.

Conclusion: The Future of Python Dependency Management

Python dependency management has been a persistent challenge, but uv represents a significant leap forward in addressing these issues. Its speed, simplicity, and comprehensive feature set make it an ideal choice for both beginners and experienced developers.

For AI-assisted development workflows, uv is particularly valuable. When AI generates code with dependencies, uv handles those dependencies automatically, allowing developers to focus on the code itself rather than environment setup. This dramatically reduces the friction in using AI-generated code, enabling faster iteration and experimentation.

Whether you're writing simple scripts, building complex applications, or teaching others to code with Python, uv simplifies the dependency management process and removes a significant source of frustration from the Python development experience.

References

- [1] Dominick M. "Chained to Complexity: Python Dependency Management." dominickm.com, 2024.
- [2] Palantir. "Python environment - Troubleshooting guide." palantir.com, 2024.
- [3] ActiveState. "How to Solve Reproducible Environments the Easy Way." activestate.com, 2025.
- [15] Astral Documentation. "Managing dependencies | uv." docs.astral.sh, 2025.
- [14] mathspp. "Using uv to build and install Python CLI apps." mathspp.com, 2025.
- [13] mathspp. "Standalone executable Python scripts with uv." mathspp.com, 2025.
- [5] PyPA. "pipx — Install and Run Python Applications in Isolated Environments." github.com/pypa/pipx.
- [8] Real Python. "Managing Python Projects With uv: An All-in-One Solution." realpython.com, 2025.

⁂

1. https://dominickm.com/chained-to-complexity-python-dependency-management/
2. https://palantir.com/docs/foundry/transforms-python/environment-troubleshooting/
3. https://www.activestate.com/blog/how-to-solve-reproducible-environments-the-easy-way/
4. https://news.ycombinator.com/item?id=20672201
5. https://github.com/pypa/pipx
6. https://prabhupant.github.io/2024/04/05/python-poetry-package-management.html
7. https://docs.conda.io/docs/using/envs.html
8. https://realpython.com/python-uv/
9. https://docs.astral.sh/uv/guides/install-python/
10. https://code.visualstudio.com/docs/python/environments
11. https://docs.astral.sh/uv/guides/integration/jupyter/
12. https://docs.astral.sh/uv/guides/scripts/
13. https://mathspp.com/blog/til/standalone-executable-python-scripts-with-uv
14. https://note.nkmk.me/en/python-uv-cli-tool/
15. https://docs.astral.sh/uv/concepts/projects/dependencies/