Python Dependency Management with uv: Complete Guide

Introduction: The Python Dependency Problem

Python's flexibility comes with a significant challenge: **dependency management**. Unlike languages with built-in package managers, Python's ecosystem has evolved multiple competing solutions, each with trade-offs that create friction for developers, especially beginners.

Core Challenges

Version Conflicts: Different projects require different versions of the same package, leading to "dependency hell" where satisfying one project breaks another.

Environment Isolation: Without proper isolation, installing packages globally can corrupt your system Python or cause conflicts between projects.

Slow Operations: Traditional tools like pip can be painfully slow for dependency resolution and installation, especially with complex dependency trees.

Complex Workflows: Setting up reproducible environments often requires multiple tools and commands, creating barriers for newcomers.

Traditional Tools: Strengths and Limitations

pip (Built-in Package Installer)

Purpose: Basic package installation and management

```bash
pip install requests numpy
pip freeze > requirements.txt
```

Limitations: No environment isolation, slow dependency resolution, poor conflict handling, no lock files for reproducible builds.

pipx (Isolated Application Installer)

Purpose: Install Python CLI tools in isolated environments

```bash
pipx install black
pipx install jupyter
```

Limitations: Only for applications, not libraries; doesn't solve project-level dependency management.

Poetry (Modern Dependency Manager)

Purpose: Project-centric dependency management with lock files

```bash
poetry init
poetry add requests
poetry install
```

Limitations: Slow dependency resolution, complex for simple scripts, heavyweight for quick tasks.

Conda (Data Science Ecosystem)

Purpose: Package and environment management, especially for data science

```bash
conda create -n myenv python=3.11
conda activate myenv
conda install numpy pandas
```

Limitations: Large installations, slow operations, mixing conda and pip can cause issues.

uv: The Modern Solution

uv is a next-generation Python package manager written in Rust that addresses most traditional pain points:

Key Advantages

1. **Speed**: 10-100x faster than pip for most operations
2. **Simplicity**: Single tool for multiple use cases
3. **Inline Dependencies**: Declare dependencies directly in scripts
4. **Automatic Environment Management**: Creates isolated environments automatically
5. **Cross-platform**: Works consistently across Windows, macOS, and Linux
6. **Drop-in Replacement**: Compatible with existing pip workflows

Environment Setup Guide

1. Installing uv

Windows (PowerShell):

```
powershell
```

```
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"
```

macOS/Linux:

```
bash
```

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

Alternative (using pip):

bash

```
pip install uv
```

Verify installation:

bash

```
uv --version
```

2. Python Environment Setup

Install Python versions with uv:

bash

```
# Install latest Python
uv python install

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

# List available versions
uv python list
```

Create project environment:

bash

```bash
# Create new project
uv init my-project
cd my-project

# Or initialize in existing directory
uv init
```

3. Jupyter Notebook Setup

Install Jupyter with uv:

bash

```bash
# Global installation for system-wide access
uv tool install jupyter

# Or install in project environment
uv add jupyter
```

Launch Jupyter:

bash

```
# From anywhere (if installed as tool)
jupyter notebook

# Or from project directory
uv run jupyter notebook
```

4. VSCode/Cursor Integration

Install Python extension in VSCode/Cursor, then:

1. **Select Python Interpreter**:
   - Open Command Palette (`Ctrl+Shift+P`)
   - Type "Python: Select Interpreter"
   - Choose the uv-managed environment (usually in `.venv`)
2. **Integrated Terminal Setup**:

   bash

   ```
   # VSCode will automatically activate the environment
   # Or manually activate:
   uv shell
   ```

3. **Jupyter Integration**:
   - Install Jupyter extension in VSCode

- Create `.ipynb` files that automatically use your project environment
- Or use "Python: Create Jupyter Notebook" command

5. Complete Development Setup

Recommended workflow for new projects:

```bash
# 1. Create project directory
mkdir my-ai-scripts && cd my-ai-scripts

# 2. Initialize uv project
uv init

# 3. Add common dependencies
uv add requests pandas numpy matplotlib

# 4. Add development tools
uv add --dev jupyter black ruff

# 5. Open in editor
code .  # or cursor .
```

Standalone Script Tutorial

Basic Inline Dependencies

Create `fetch_weather.py`:

```
python
```

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

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

def get_weather(city):
    """Fetch weather data for a city."""
    url = f"https://wttr.in/{city}?format=j1"
    response = requests.get(url)
    return response.json()

def display_weather(data):
    """Display weather in a nice table."""
    console = Console()
    table = Table(title="Weather Information")

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

```
    current = data['current_condition'][0]
    table.add_row("Temperature", f"{current['temp_C']}°C")
    table.add_row("Feels Like", f"{current['FeelsLikeC']}°C")
    table.add_row("Humidity", f"{current['humidity']}%")
    table.add_row("Description", current['weatherDesc'][0]['value'])

    console.print(table)

if __name__ == "__main__":
    city = input("Enter city name: ")
    weather_data = get_weather(city)
    display_weather(weather_data)
```

Run the script:

```
bash

uv run fetch_weather.py
```

uv will automatically:

1. Create a temporary environment
2. Install `requests` and `rich`
3. Execute the script
4. Cache the environment for future runs

Data Analysis Script Example

Create `analyze_data.py`:

```
python
```

```python
# /// script
# requires-python = ">=3.9"
# dependencies = [
#     "pandas",
#     "matplotlib",
#     "seaborn",
#     "numpy",
# ]
# ///

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from pathlib import Path

def create_sample_data():
    """Generate sample sales data."""
    np.random.seed(42)
    dates = pd.date_range('2024-01-01', periods=100, freq='D')
    sales = np.random.normal(1000, 200, 100) + np.sin(np.arange(100) * 0.1) * 100

    data = pd.DataFrame({
        'date': dates,
        'sales': np.maximum(sales, 0),  # Ensure positive values
        'month': dates.month
```

```
    })
    return data

def analyze_sales(df):
    """Perform basic sales analysis."""
    print("Sales Analysis Report")
    print("=" * 30)
    print(f"Total Sales: ${df['sales'].sum():,.2f}")
    print(f"Average Daily Sales: ${df['sales'].mean():,.2f}")
    print(f"Best Day: {df.loc[df['sales'].idxmax(), 'date'].strftime('%Y-%m-%d')}")

    # Monthly analysis
    monthly = df.groupby('month')['sales'].agg(['sum', 'mean']).round(2)
    print("\nMonthly Summary:")
    print(monthly)

def create_visualizations(df):
    """Create sales visualizations."""
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))
    fig.suptitle('Sales Analysis Dashboard', fontsize=16)

    # Daily sales trend
    axes[0, 0].plot(df['date'], df['sales'])
    axes[0, 0].set_title('Daily Sales Trend')
    axes[0, 0].set_xlabel('Date')
    axes[0, 0].set_ylabel('Sales ($)')
```

```python
    # Sales distribution
    axes[0, 1].hist(df['sales'], bins=20, alpha=0.7)
    axes[0, 1].set_title('Sales Distribution')
    axes[0, 1].set_xlabel('Sales ($)')
    axes[0, 1].set_ylabel('Frequency')

    # Monthly boxplot
    df.boxplot(column='sales', by='month', ax=axes[1, 0])
    axes[1, 0].set_title('Monthly Sales Distribution')

    # Monthly trend
    monthly_avg = df.groupby('month')['sales'].mean()
    axes[1, 1].bar(monthly_avg.index, monthly_avg.values)
    axes[1, 1].set_title('Average Monthly Sales')
    axes[1, 1].set_xlabel('Month')
    axes[1, 1].set_ylabel('Average Sales ($)')

    plt.tight_layout()
    plt.savefig('sales_analysis.png', dpi=300, bbox_inches='tight')
    plt.show()

    print(f"\nVisualization saved as 'sales_analysis.png'")

if __name__ == "__main__":
    # Generate or load data
    data = create_sample_data()
```

```
    # Perform analysis
    analyze_sales(data)

    # Create visualizations
    create_visualizations(data)
```

Run the analysis:

```
bash

uv run analyze_data.py
```

Web Scraping Script Example

Create `scrape_news.py`:

```
python
```

```python
# /// script
# requires-python = ">=3.8"
# dependencies = [
#     "requests",
#     "beautifulsoup4",
#     "lxml",
#     "rich",
# ]
# ///

import requests
from bs4 import BeautifulSoup
from rich.console import Console
from rich.table import Table
from rich.progress import track
from urllib.parse import urljoin
import time

def scrape_hacker_news():
    """Scrape top stories from Hacker News."""
    url = "https://news.ycombinator.com/"
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')

    stories = []
    story_links = soup.find_all('span', class_='titleline')
```

```
    for i, story in enumerate(track(story_links[:10], description="Fetching stori
        try:
            link = story.find('a')
            if link:
                title = link.text.strip()
                href = link.get('href')

                # Handle relative URLs
                if href.startswith('item?'):
                    href = urljoin(url, href)

                stories.append({
                    'rank': i + 1,
                    'title': title[:80] + '...' if len(title) > 80 else title,
                    'url': href
                })
            time.sleep(0.1)  # Be respectful
        except Exception as e:
            print(f"Error processing story {i}: {e}")
            continue

    return stories

def display_stories(stories):
    """Display stories in a formatted table."""
    console = Console()
```

```
    table = Table(title="🔥 Top Hacker News Stories")

    table.add_column("Rank", style="cyan", width=6)
    table.add_column("Title", style="white")
    table.add_column("URL", style="blue", max_width=50)

    for story in stories:
        table.add_row(
            str(story['rank']),
            story['title'],
            story['url']
        )

    console.print(table)

if __name__ == "__main__":
    console = Console()

    with console.status("[bold green]Scraping Hacker News..."):
        stories = scrape_hacker_news()

    if stories:
        display_stories(stories)
    else:
        console.print("[red]Failed to fetch stories[/red]")
```

Run the scraper:

```bash
uv run scrape_news.py
```

Advanced uv Features

Project Management

Initialize with specific Python version:

```bash
uv init --python 3.11 my-project
```

Add dependencies with version constraints:

```bash
uv add "requests>=2.28"
uv add "pandas~=2.0"
uv add --dev pytest black ruff
```

Lock dependencies:

```bash
uv lock
```

Script Management

Run scripts with specific Python version:

```bash
uv run --python 3.11 script.py
```

Install script dependencies globally:

```bash
uv tool install --from git+https://github.com/user/repo script-name
```

Environment Commands

bash

```bash
# Activate shell
uv shell

# Run commands in environment
uv run python
uv run pytest
uv run jupyter notebook

# Sync environment (install/update dependencies)
uv sync

# Export requirements
uv export --format requirements-txt > requirements.txt
```

Best Practices for AI-Generated Scripts

1. Always Use Inline Dependencies

Include the `# /// script` block in every standalone script to ensure portability.

2. Version Pin Critical Dependencies

```python
# /// script
# dependencies = [
#     "requests==2.31.0",  # Pin for stability
#     "pandas>=2.0,<3.0",  # Allow minor updates
# ]
# ///
```

3. Handle Errors Gracefully

```python
try:
    import requests
except ImportError:
    print("Run with: uv run script.py")
    exit(1)
```

4. Use Rich for Better Output

Always include `rich` for better terminal output in user-facing scripts.

5. Project Structure for Multiple Scripts

```
ai-scripts/
├── pyproject.toml          # uv project file
├── scripts/
│   ├── data_analysis.py
│   ├── web_scraper.py
│   └── api_client.py
├── shared/
│   └── utils.py           # Common utilities
└── README.md
```

Troubleshooting Common Issues

Script Won't Run

bash

```
# Check uv installation
uv --version

# Verify Python availability
uv python list

# Run with verbose output
uv run -v script.py
```

Dependencies Not Installing

bash

```
# Clear cache
uv cache clean

# Force reinstall
uv sync --reinstall
```

Environment Issues

bash

```
# Remove and recreate environment
rm -rf .venv
uv sync
```

Conclusion

uv represents a significant step forward in Python dependency management, particularly for AI-assisted development workflows. Its speed, simplicity, and inline dependency features make it ideal for:

- **Rapid prototyping** with AI-generated scripts
- **Educational environments** where setup friction must be minimized
- **Data science workflows** requiring quick experimentation

- **Production scripts** needing reliable dependency management

By adopting uv early in your Python journey, you avoid the complexity and frustration that has historically made Python dependency management challenging, allowing you to focus on writing and running code rather than managing environments.

The inline script format is particularly powerful for AI-generated code, as it makes scripts fully self-contained and immediately executable, removing barriers between idea and implementation.