

Ship Python, Orchestrate AI

Professional Python in the AI Era

Michael Borck

2026-04-06

Ship Python, Orchestrate AI
Professional Python in the AI Era

Copyright © 2026 Michael Borck. All rights reserved.

Published by Michael Borck
Perth, Western Australia

ISBN: 979-8-255-16785-2

First edition, 2026.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the author, except for brief quotations in reviews and certain non-commercial uses permitted by copyright law.

This work is also available under a Creative Commons Attribution (CC BY) licence for content and MIT licence for code examples at the companion website. See below for details.

AI disclosure: This book was written using the methodology it describes. AI tools were used as thinking partners throughout the drafting, iterating, and refining process. The author reviewed, challenged, and took responsibility for every sentence and line of code.

Companion website: <https://michael-borck.github.io/ship-python-orchestrate-ai>

Source: <https://github.com/michael-borck/ship-python-orchestrate-ai>

Table of contents

Preface	1
1 Setting the Foundation	9
2 Advancing Your Workflow	19
3 Documentation and Deployment	47
4 Case Study: Building SimpleBot - A Python Development Workflow Example	71
5 Advanced Development Techniques	91
6 Project Management and Automation	103
7 Multi-Platform Distribution	111
8 Beyond Scripts: Notebooks, Dashboards, and Interactive Python	135
9 Case Study: From Notebook to Package with nbdev	145
10 Conclusion: Embracing Efficient Python Development	157
Acknowledgments	163
About the Author	165
Appendices	167
A Glossary of Python Development Terms	167
B Distribution Templates	171
C AI Tools for Python Development	195
D Python Development Workflow Checklist	203
E Introduction to Python IDEs and Editors	207
F Python Development Tools Reference	213
G Comparison of Python Environment and Package Management Tools	217
H Python Development Pipeline Scaffold Python Script	221
I Cookiecutter Template	227
J Hatch - Modern Python Project Management	233
K Using Conda for Environment Management	243
L Getting Started with venv	251
M UV - High-Performance Python Package Management	259
N Poetry - Modern Python Packaging and Dependency Management	267

Preface

Why This Book Exists

You can write Python. Now what?

The gap between writing scripts that work on your machine and shipping software that other people can use is enormous. The Python ecosystem offers hundreds of tools to bridge that gap — and that is exactly the problem. Which virtual environment tool? Which formatter? Which testing framework? How do you manage dependencies? How do you structure a project? The questions feel endless, and every answer leads to three more.

This book grew out of my own search for a sane, repeatable workflow for Python projects. I needed a pipeline that could target multiple platforms — web apps, PWAs, Docker containers, Windows, Linux, Mac — from a single codebase. I wanted something that was simple by default but could scale when needed, without bolting on complexity for edge cases that did not yet exist. The result is what you will find in these pages: a deliberate 80/20 approach. The 20% of practices that yield 80% of the benefits.

We settle on specific tools — uv, ruff, mypy, pytest — not because they are the only options, but because making a decision and committing to it is more productive than endlessly evaluating alternatives. If you are working in a large team with established conventions, you may choose differently. This book is for everyone else: solo developers, small teams, prototypers, and anyone who wants a professional workflow without the overhead.

Who This Book Is For

- Python developers ready to move beyond scripts to professional projects
- Solo developers and small teams who need a complete workflow without enterprise overhead
- Prototypers who want to ship to multiple platforms from one codebase
- Anyone overwhelmed by Python tooling choices who wants someone to just make the decisions

You should be comfortable writing Python. If you are still learning the language itself, start with *Code Python, Consult AI*. This book assumes you can write functions, use data structures, and debug basic errors. It teaches you what to do with that skill.

What This Book Is Not

This is not a Python tutorial. It does not teach the language. It teaches the professional practices around the language: project structure, version control, dependency management, testing, documentation, and deployment.

It is not an enterprise DevOps guide. It does not cover large-team workflows, complex CI/CD orchestration, or Kubernetes. It covers what a solo developer or small team needs to ship reliable software. The practices scale, but the book does not add complexity for situations that may never apply to you.

It is not a tool comparison. It does not present five options for every choice and leave you to decide. It picks one, explains why, and moves on. Simple but not simplistic — that is the guiding principle.

And it is not a book that ignores AI. But it uses AI differently from the other books in this series. *Code Python, Consult AI* teaches you to think with AI, to explore concepts and build understanding through conversation. This book teaches you to orchestrate AI, to set up the project structure, testing, and automation so that AI-generated code has somewhere safe to land. Without a proper pipeline, AI produces code you cannot test, cannot deploy, and cannot maintain. With one, AI becomes a force multiplier. The pipeline is what makes the difference.

If You Are Feeling Overwhelmed

That is the normal response to the Python ecosystem. The number of tools, frameworks, and opinions about “the right way” to do things is genuinely staggering. This book exists to cut through that noise. You do not need to evaluate every option. You need one good workflow that works, and the understanding to adapt it when your needs change.

How This Book Is Structured

The book builds a complete development pipeline in stages:

1. **Setting the Foundation** — project structure, version control, virtual environments
2. **Advancing Your Workflow** — dependency management, code quality (ruff), testing (pytest), type checking (mypy)
3. **Documentation and Deployment** — documentation, CI/CD automation
4. **Case Study** — applying everything to a real project (SimpleBot)
5. **Multi-Platform Distribution** — shipping to web, desktop, PWA, and containers from one codebase

Companion templates let you start new projects with the recommended structure immediately:

- **Cookiecutter template:** `cookiecutter gh:michael-borck/ship-python-cookiecutter`
- **GitHub template:** `ship-python-template`
- **Example project:** `ship-python-example`

Conventions Used in This Book

Code examples appear with syntax highlighting:

```
uv run pytest --cov=src
```

Terminal output and configuration content appear as plain text blocks:

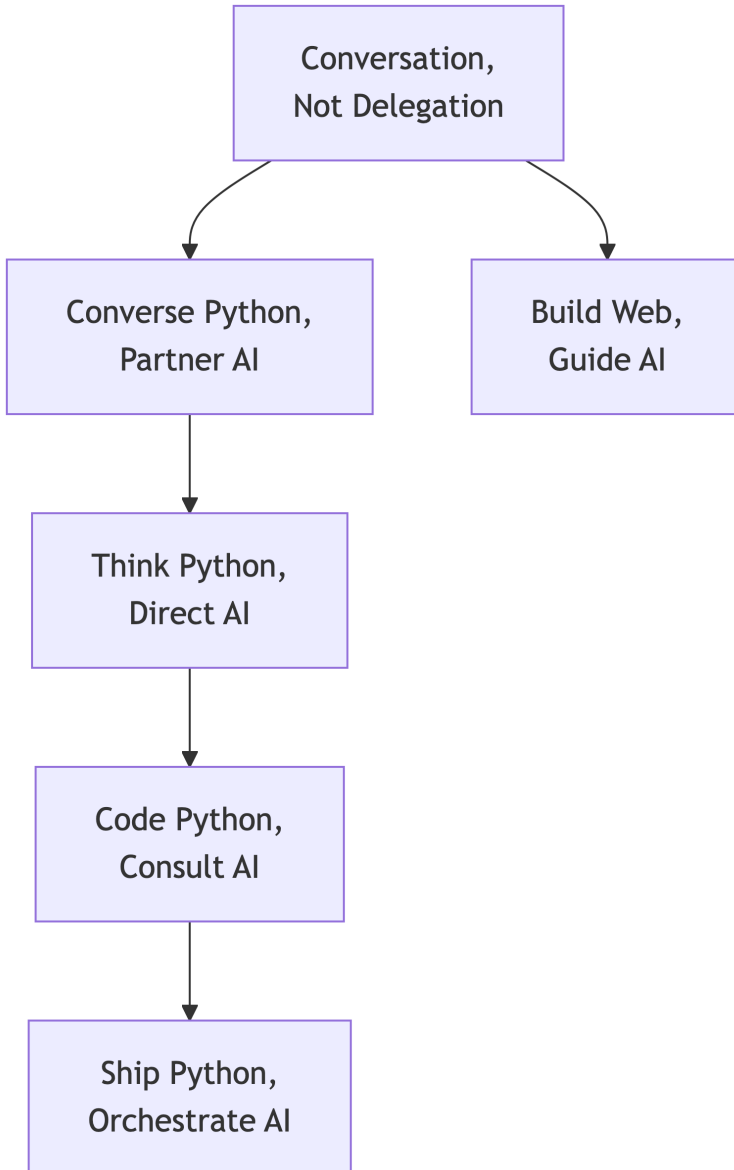
```
my_project/  
  src/  
    my_package/  
      tests/
```

Throughout the chapters you will find:

- **AI Tips** — practical advice for using AI with the practice being taught
- **Callout boxes** — important notes, tips, and warnings highlighted for reference

The Series

This book is part of a series designed to help you master modern software development in the AI era.



Conversation, Not Delegation — the general methodology for working with AI across any discipline.

Converse Python, Partner AI — intentional prompting methodology applied to software development.

Think Python, Direct AI — computational thinking for absolute beginners.

Code Python, Consult AI — focused Python fundamentals with AI integration.

Ship Python, Orchestrate AI (this book) — professional Python development practices and tooling.

Build Web, Guide AI — web development with AI as your development partner.

All titles are available at books.borck.education.

Ways to Engage with This Book

This book is available in several formats. Pick whichever fits how you work and learn.

- **Read it online.** The full book is freely available at the companion website, with dark mode, search, and navigation.
- **Read it on paper or e-reader.** Available as a paperback and ebook through Amazon KDP.
- **Converse with it.** The online edition includes a chatbot grounded in the book's content.
- **Feed it to your own AI.** The `11m.txt` file provides a clean text version of the entire book, ready to paste into ChatGPT, Claude, or any AI tool.
- **Run the code.** All code examples and templates are available on GitHub. DeepWiki provides an AI-navigable view of the repository.
- **Browse all books.** This book is part of a series. See all titles at books.borck.education.

The online version is always the most current.

Source Code & Feedback

All code examples, templates, and supplementary materials are available at:

- **This book's repository:** <https://github.com/michael-borck/ship-python-orchestrate-ai>
- **Cookiecutter template:** `cookiecutter gh:michael-borck/ship-python-cookiecutter`
- **GitHub template:** <https://github.com/michael-borck/ship-python-template>
- **Example project:** <https://github.com/michael-borck/ship-python-example>

Found an error? Have a suggestion?

- Open an issue: <https://github.com/michael-borck/ship-python-orchestrate-ai/issues>
- Email: michael@borck.me

Copyright

Ship Python, Orchestrate AI: Professional Python in the AI Era

Copyright 2026 Michael Borck. All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

First Edition, 2026

While every precaution has been taken in the preparation of this book, the author assumes no responsibility for errors or omissions, or for damages resulting from the use of information contained herein.

Cover Art Created using AI image generation, with prompts crafted by the author depicting the themes of professional Python development and human-AI collaboration.

Production Notes This book was written in Markdown using Quarto. The text is set in system fonts optimised for screen and print reading. Code examples use a monospace font for clarity.

Editorial assistance provided by Claude (Anthropic). The author reviewed and approved all content.

Online Edition A free online version of this book is available at: <https://michael-borck.github.io/ship-python-orchestrate-ai>

michael@borck.me

Chapter 1

Setting the Foundation

1.1 Python Project Structure Best Practices

A well-organised project structure is the cornerstone of maintainable Python code. Even before writing a single line of code, decisions about how to organise your files will impact how easily you can test, document, and expand your project.

The structure we recommend follows modern Python conventions, prioritizing clarity and separation of concerns:

```
my_project/
  src/                # Main source code directory
    my_package/      # Your actual Python package
      __init__.py    # Makes the directory a package
      main.py        # Core functionality
      helpers.py     # Supporting functions/classes
  tests/             # Test suite
    __init__.py
    test_main.py     # Tests for main.py
    test_helpers.py # Tests for helpers.py
  docs/              # Documentation (can start simple)
    index.md         # Main documentation page
  .gitignore         # Files to exclude from Git
  README.md          # Project overview and quick start
  requirements.txt   # Project dependencies
  pyproject.toml     # Tool configuration
```

1.1.1 Why Use the `src` Layout?

The `src` layout (placing your package inside a `src` directory rather than at the project root) provides several advantages:

1. **Enforces proper installation:** When developing, you must install your package to use it, ensuring you're testing the same way users will experience it.
2. **Prevents accidental imports:** You can't accidentally import from your project without installing it, avoiding confusing behaviours.

3. **Clarifies package boundaries:** Makes it explicit which code is part of your distributable package.

While simpler projects might skip this layout, adopting it early builds good habits and makes future growth easier.

💡 AI Tip: Scaffolding with AI

Ask your AI assistant to generate a project structure tailored to your needs: *“I’m building a CLI tool that processes CSV files. Generate the directory structure and initial files for a Python project using the src layout, including tests and a pyproject.toml.”* AI is excellent at generating boilerplate — the value you add is knowing whether the structure it suggests follows the conventions in this chapter.

1.1.2 Key Components Explained

- **src/my__package/:** Contains your actual Python code. The package name should be unique and descriptive.
- **tests/:** Keeps tests separate from implementation but adjacent in the repository.
- **docs/:** Houses documentation, starting simple and growing as needed.
- **.gitignore:** Tells Git which files to ignore (like virtual environments, cache files, etc.).
- **README.md:** The first document anyone will see—provide clear instructions on installation and basic usage.
- **requirements.txt:** Lists your project’s dependencies. We’ll explore more advanced dependency management techniques in Part 2.
- **pyproject.toml:** Configuration for development tools like Ruff and mypy, following modern standards.

1.1.3 Getting Started

Creating this structure is straightforward. Here’s how to initialize a basic project:

```
# Create the project directory
mkdir my_project && cd my_project

# Create the basic structure
mkdir -p src/my_package tests docs

# Initialize the Python package
touch src/my_package/__init__.py
touch src/my_package/main.py

# Create initial test files
touch tests/__init__.py
touch tests/test_main.py

# Create essential files
echo "# My Project\nA short description of my project." > README.md
touch requirements.in
touch pyproject.toml
```

```
# Initialize Git repository
git init
```

1.1.4 Applications vs. Packages: Knowing Your Project Type

Understanding whether you're building an **application** or a **package** influences structure decisions and development priorities:

Python Applications are end-user focused programs: - Web applications (Django/Flask projects) - Command-line tools and utilities - Desktop applications - Data processing scripts - Have clear entry points and user interfaces - Often include configuration files and deployment considerations

Python Packages are developer-focused libraries: - Reusable code modules (like `requests` or `pandas`) - APIs and frameworks - Plugin systems - Focus on import interfaces and documentation - Published to PyPI for others to use

Key Differences in Practice:

Aspect	Applications	Packages
Entry point	<code>main.py</code> , CLI commands	Import interfaces
Dependencies	Can pin exact versions	Should use flexible ranges
Documentation	User guides, deployment	API docs, examples
Testing focus	End-to-end workflows	Unit tests, edge cases
Configuration	Settings files, env vars	Initialization parameters

Most projects start as applications and may later extract reusable components into packages. Our recommended structure accommodates both paths—you can begin with application-focused development and naturally evolve toward package-like modularity as your codebase matures.

i Note

This chapter focuses on script-based projects, but notebooks (Jupyter, Colab) are also a legitimate way to ship Python work. For data scientists, educators, and researchers, a well-structured notebook with a sharing link *is* shipping. We cover notebooks, dashboards, and interactive tools in Chapter 8.

Practical example: A data analysis script (application) might extract its core algorithms into a separate analytics package, while keeping the command-line interface and configuration handling in the main application.

This structure promotes maintainability and follows Python's conventions. It might seem like overkill for tiny scripts, but as your project grows, you'll appreciate having this organisation from the start.

In the next section, we'll build on this foundation by implementing version control best practices.

1.2 Version Control Fundamentals

Version control is an essential part of modern software development, and Git has become the de facto standard. Even for small solo projects, proper version control offers invaluable benefits for tracking changes, experimenting safely, and maintaining a clear history of your work.

1.2.1 Setting Up Git

If you haven't set up Git yet, here's how to get started:

```
# Configure your identity (use your actual name and email)
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"

# Initialize Git in your project (if not done already)
git init

# Create a .gitignore file to exclude unnecessary files
```

A good `.gitignore` file is essential for Python projects. Here's a simplified version to start with:

```
# Virtual environments
.venv/
venv/
env/

# Python cache files
__pycache__/
*.py[cod]
*$py.class
.pytest_cache/

# Distribution / packaging
dist/
build/
*.egg-info/

# Local development settings
.env
.vscode/
.idea/

# Coverage reports
htmlcov/
.coverage

# Generated documentation
site/
```

1.2.2 Basic Git Workflow

For beginners, a simple Git workflow is sufficient:

1. **Make changes** to your code
2. **Stage changes** you want to commit
3. **Commit** with a meaningful message
4. **Push** to a remote repository (like GitHub)

Here's what this looks like in practice:

```
# Check what files you've changed
git status

# Stage specific files (or use git add . for all changes)
git add src/my_package/main.py tests/test_main.py

# Commit changes with a descriptive message
git commit -m "Add user authentication function and tests"

# Push to a remote repository (if using GitHub or similar)
git push origin main
```

1.2.3 Effective Commit Messages

Good commit messages are vital for understanding project history. Follow these simple guidelines:

1. Use the imperative mood (“Add feature” not “Added feature”)
2. Keep the first line under 50 characters as a summary
3. When needed, add more details after a blank line
4. Explain *why* a change was made, not just *what* changed

Example of a good commit message:

```
Add password validation function

- Implements minimum length of 8 characters
- Requires at least one special character
- Fixes #42 (weak password vulnerability)
```

1.2.4 Branching for Features and Fixes

As your project grows, a branching workflow helps manage different streams of work:

```
# Create a new branch for a feature
git checkout -b feature/user-profiles

# Make changes, commit, and push to the branch
git add .
git commit -m "Add user profile page"
git push origin feature/user-profiles

# When ready, merge back to main (after review)
```

```
git checkout main
git merge feature/user-profiles
```

For team projects, consider using pull/merge requests on platforms like GitHub or GitLab rather than direct merges to the main branch. This enables code review and discussion before changes are incorporated.

1.2.5 Integrating with GitHub or GitLab

Hosting your repository on GitHub, GitLab, or similar services provides:

1. A backup of your code
2. Collaboration tools (issues, pull requests)
3. Integration with CI/CD services
4. Visibility for your project

To connect your local repository to GitHub:

```
# After creating a repository on GitHub
git remote add origin https://github.com/yourusername/my_project.git
git branch -M main
git push -u origin main
```

1.2.6 Git Best Practices for Beginners

1. **Commit frequently:** Small, focused commits are easier to understand and review
2. **Never commit sensitive data:** Passwords, API keys, etc. should never enter your repository
3. **Pull before pushing:** Always integrate others' changes before pushing your own
4. **Use meaningful branch names:** Names like `feature/user-login` or `fix/validation-bug` explain the purpose

AI Tip: Git and AI Work Well Together

AI assistants can help you write commit messages, generate `.gitignore` files for specific frameworks, and explain Git commands you are unsure about. Try: *“I just added user authentication with Flask-Login. Write a commit message following conventional commit format.”* But always review what AI generates — it does not know your project’s commit history or conventions unless you provide them.

Version control may seem like an overhead for very small projects, but establishing these habits early will pay dividends as your projects grow in size and complexity. It’s much easier to start with good practices than to retrofit them later.

In the next section, we’ll set up a virtual environment and explore basic dependency management to isolate your project and manage its requirements.

1.3 Virtual Environments and Basic Dependencies

Python’s flexibility with packages and imports is powerful, but can quickly lead to conflicts between projects. Virtual environments solve this problem by creating isolated spaces for each project’s dependencies.

1.3.1 Understanding Virtual Environments

A virtual environment is an isolated copy of Python with its own packages, separate from your system Python installation. This isolation ensures:

- Different projects can use different versions of the same package
- Installing a package for one project won't affect others
- Your development environment closely matches production

1.3.2 Setting Up a Virtual Environment with venv

Python comes with `venv` built in, making it the simplest way to create virtual environments:

```
# Create a virtual environment named ".venv" in your project
python -m venv .venv

# Activate the environment (the command differs by platform)
# On Windows:
.venv\Scripts\activate
# On macOS/Linux:
source .venv/bin/activate

# Your prompt should change to indicate the active environment
(venv) $
```

Once activated, any packages you install will be confined to this environment. When you're done working on the project, you can deactivate the environment:

```
deactivate
```

Tip

Using `.venv` as the environment name (with the leading dot) makes it hidden in many file browsers, reducing clutter. Make sure `.venv/` is in your `.gitignore` file — you never want to commit this directory.

1.3.3 Basic Dependency Management

With your virtual environment active, you can install packages using `pip`:

```
# Install a specific package
pip install requests

# Install multiple packages
pip install pytest black
```

When working on a team project or deploying to production, you'll need to track and share these dependencies. For basic projects, you can manually maintain a `requirements.txt` file with the packages you need:

```
# Create or add packages to your requirements.txt file
echo "requests" >> requirements.txt
```

```
echo "pytest" >> requirements.txt

# Install from your requirements file
pip install -r requirements.txt
```

This approach works well for simple projects, especially when you're just getting started. However, as we'll see in Part 2, there are limitations to this basic method:

- It doesn't handle indirect dependencies (dependencies of your dependencies) automatically
- It doesn't distinguish between your project's requirements and development tools
- It doesn't provide version locking for reproducible environments

i Looking Ahead

In Part 2, we'll explore more robust dependency management with tools like `pip-tools` and `uv`, which solve these limitations by creating proper "lock files" while maintaining a clean list of direct dependencies. We'll also see how these tools help ensure deterministic builds — a crucial feature as your projects grow in complexity.

1.3.4 Practical Example: Setting Up a New Project

Let's combine what we've learned so far with a practical example. Here's how to set up a new project with good practices:

```
# Create project structure
mkdir -p my_project/src/my_package my_project/tests
cd my_project

# Initialize Git repository
git init
echo "*.pyc\n__pycache__/\n.venv/\n*.egg-info/" > .gitignore

# Create basic files
echo "# My Project\n\nA description of my project." > README.md
touch src/my_package/__init__.py
touch src/my_package/main.py
touch tests/__init__.py
touch tests/test_main.py
touch requirements.in

# Create and activate virtual environment
python -m venv .venv
source .venv/bin/activate # On Windows: .venv\Scripts\activate

# Install initial dependencies
pip install pytest
echo "pytest" > requirements.txt

# Initial Git commit
git add .
```

```
git commit -m "Initial project setup"
```

1.4 Jumpstarting Your Projects with Templates

Now that we've covered the essential foundation for Python development, you might be wondering how to apply these practices efficiently when starting new projects. Rather than recreating this structure manually each time, we offer two approaches to jumpstart your projects:

1.4.1 Simple Scaffolding Script

For those who prefer a transparent, straightforward approach, we've created a simple bash script that creates the basic project structure we've discussed:

```
# Download the script
REPO="michael-borck/ship-python-template"
URL="https://raw.githubusercontent.com"
curl -O $URL/$REPO/main/scripts/scaffold_python_project.sh
chmod +x scaffold_python_project.sh

# Create a new project
./scaffold_python_project.sh my_project
```

This script creates a minimal but well-structured Python project with: - The recommended `src` layout - Basic test setup - Simple `pyproject.toml` configuration - Version control initialization - Placeholder documentation

The script is intentionally simple and readable, allowing you to understand exactly what's happening and modify it for your specific needs. This approach is ideal for learning or for smaller projects where you want maximum visibility into the setup process.

1.4.2 Cookiecutter Template (For More Comprehensive Setup)

For more complex projects or when you want a more feature-rich starting point, we also provide a cookiecutter template that implements the full development pipeline described throughout this book:

```
# Install cookiecutter
pip install cookiecutter

# Create a new project from the template
cookiecutter gh:michael-borck/ship-python-cookiecutter
```

The cookiecutter template offers more customisation options and includes: - All the foundational structure from the simple script - Comprehensive tool configurations - Optional documentation setup with MkDocs - CI/CD workflow configurations - Advanced dependency management - Security scanning integration

This approach is covered in detail in Appendix C and is recommended when you're ready to adopt more advanced practices or when working with larger teams.

1.4.3 GitHub Repository Templates (For No-Installation Simplicity)

For the ultimate in simplicity, we also provide a GitHub repository template that requires no local tool installation. GitHub templates offer a frictionless way to create new projects with the same structure and files:

1. Visit the template repository at <https://github.com/michael-borck/ship-python-template>
2. Click the “Use this template” button
3. Name your new repository and create it
4. Clone your new repository locally

```
git clone https://github.com/yourusername/your-new-project.git
cd your-new-project
```

While GitHub templates don’t offer the same parameterization as cookiecutter (file contents remain exactly as they were in the template), they provide the lowest barrier to entry for getting started with a well-structured project. After creating your repository from the template, you can manually customise file contents like project name, author information, and other details.

The GitHub template includes: - The recommended `src` layout - Basic test structure - `.gitignore` and `pyproject.toml` configuration - Documentation structure - Example code and tests

This approach is ideal for quickly starting new projects when you don’t want to install additional tools or when you’re introducing others to Python best practices with minimal setup overhead.

All these options—the simple script, the cookiecutter template—embody, and GitHub repository templates embody our philosophy of “Simple but not Simplistic.” Choose the option that best fits your current needs and comfort level. As your projects grow in complexity, you can gradually adopt more sophisticated practices while maintaining the solid foundation established here.

In Part 2, we’ll build on this foundation by exploring robust dependency management, code quality tools, testing strategies, and type checking — the next layers in our Python development pipeline.

Chapter 2

Advancing Your Workflow

2.1 Robust Dependency Management with `pip-tools` and `uv`

As your projects grow in complexity or involve more developers, the basic `pip freeze > requirements.txt` approach starts to show limitations. You need a dependency management system that gives you more control and ensures truly reproducible environments.

2.1.1 The Problem with `pip freeze`

While `pip freeze` is convenient, it has several drawbacks:

1. **No distinction between direct and indirect dependencies:** You can't easily tell which packages you explicitly need versus those that were installed as dependencies of other packages.
2. **Maintenance challenges:** When you want to update a package, you may need to regenerate the entire requirements file, potentially changing packages you didn't intend to update.
3. **No environment synchronization:** Installing from a requirements.txt file adds packages but doesn't remove packages that are no longer needed.
4. **No explicit dependency specification:** You can't easily specify version ranges (e.g., "I need any Django 4.x version") or extras.

Let's explore two powerful solutions: `pip-tools` and `uv`.

2.1.2 Solution 1: `pip-tools`

`pip-tools` introduces a two-file approach to dependency management:

1. **requirements.in:** A manually maintained list of your direct dependencies, potentially with version constraints.
2. **requirements.txt:** A generated lock file containing exact versions of all dependencies (direct and indirect).

2.1.2.1 Getting Started with pip-tools

```
# Install pip-tools in your virtual environment
pip install pip-tools

# Create a requirements.in file with your direct dependencies
cat > requirements.in << EOF
requests>=2.25.0 # Use any version 2.25.0 or newer
flask==2.0.1     # Use exactly this version
pandas          # Use any version
EOF

# Compile the lock file
pip-compile requirements.in

# Install the exact dependencies
pip-sync requirements.txt
```

The generated `requirements.txt` will contain exact versions of your specified packages plus all their dependencies, including hashes for security.

2.1.2.2 Managing Development Dependencies

For a cleaner setup, you can separate production and development dependencies:

```
# Create requirements-dev.in
cat > requirements-dev.in << EOF
-c requirements.txt # Constraint: use same versions as in
requirements.txt
pytest>=7.0.0
pytest-cov
ruff
mypy
EOF

# Compile development dependencies
pip-compile requirements-dev.in -o requirements-dev.txt

# Install all dependencies (both prod and dev)
pip-sync requirements.txt requirements-dev.txt
```

2.1.2.3 Updating Dependencies

When you need to update packages:

```
# Update all packages to their latest allowed versions
pip-compile --upgrade requirements.in

# Update a specific package
pip-compile --upgrade-package requests requirements.in

# After updating, sync your environment
pip-sync requirements.txt
```

2.1.3 Solution 2: uv

uv is a newer, Rust-based tool that provides significant speed improvements while maintaining compatibility with existing Python packaging standards. It combines environment management, package installation, and dependency resolution in one tool.

2.1.3.1 Getting Started with uv

```
# Install uv (globally with pipx or in your current environment)
pipx install uv
# Or: pip install uv

# Create a virtual environment (if needed)
uv venv

# Activate the environment as usual
source .venv/bin/activate # On Windows: .venv\Scripts\activate

# Create the same requirements.in file as above
cat > requirements.in << EOF
requests>=2.25.0
flask==2.0.1
pandas
EOF

# Compile the lock file
uv pip compile requirements.in -o requirements.txt

# Install dependencies
uv pip sync requirements.txt
```

2.1.3.2 Key Advantages of uv

1. **Speed:** uv is significantly faster than standard pip and pip-tools, especially for large dependency trees.
2. **Global caching:** uv implements efficient caching, reducing redundant downloads across projects.
3. **Consolidated tooling:** Acts as a replacement for multiple tools (pip, pip-tools, virtualenv) with a consistent interface.
4. **Enhanced dependency resolution:** Often provides clearer error messages for dependency conflicts.

2.1.3.3 Managing Dependencies with uv

uv supports the same workflow as pip-tools but with different commands:

```
# For development dependencies
cat > requirements-dev.in << EOF
```

```

-c requirements.txt
pytest>=7.0.0
pytest-cov
ruff
mypy
EOF

# Compile dev dependencies
uv pip compile requirements-dev.in -o requirements-dev.txt

# Install all dependencies
uv pip sync requirements.txt requirements-dev.txt

# Update a specific package
uv pip compile --upgrade-package requests requirements.in

```

2.1.4 Choosing Between pip-tools and uv

Both tools solve the core problem of creating reproducible environments, but with different tradeoffs:

Factor	pip-tools	uv
Speed	Good	Excellent (often 10x+ faster)
Installation	Simple Python package	External tool (but simple to install)
Maturity	Well-established	Newer but rapidly maturing
Functionality	Focused on dependency locking	Broader tool combining multiple functions
Learning curve	Minimal	Minimal (designed for compatibility)

For beginners or smaller projects, pip-tools offers a gentle introduction to proper dependency management with minimal new concepts. For larger projects or when speed becomes important, uv provides significant benefits with a similar workflow.

2.1.5 Best Practices for Either Approach

Regardless of which tool you choose:

1. **Commit both .in and .txt files** to version control. The .in files represent your intent, while the .txt files ensure reproducibility.
2. **Use constraints carefully.** Start with loose constraints (just package names) and add version constraints only when needed.
3. **Regularly update dependencies** to get security fixes, using `--upgrade` or `--upgrade-package`.
4. **Always use `pip-sync` or `uv pip sync`** instead of `pip install -r requirements.txt` to ensure your environment exactly matches the lock file.

💡 AI Tip: Dependency Decisions

When choosing between packages that serve similar purposes, ask your AI assistant for a comparison: *“Compare requests vs httpx for making HTTP calls in a Python 3.11 project. I need async support and type hints.”* AI can surface trade-offs quickly, but verify its claims about package popularity and maintenance status — check PyPI download stats and GitHub activity yourself.

In the next section, we’ll explore how to maintain code quality through automated formatting and linting with Ruff, taking your workflow to the next professional level.

2.2 Code Quality Tools with Ruff

Writing code that works is only part of the development process. Code should also be readable, maintainable, and free from common errors. This is where code quality tools come in, helping you enforce consistent style and catch potential issues early.

2.2.1 The Evolution of Python Code Quality Tools

Traditionally, Python developers used multiple specialized tools:

- **Black** for code formatting
- **isort** for import sorting
- **Flake8** for linting (style checks)
- **Pylint** for deeper static analysis

While effective, maintaining configuration for all these tools was cumbersome. Enter Ruff – a modern, Rust-based tool that combines formatting and linting in one incredibly fast package.

2.2.2 Why Ruff?

Ruff offers several compelling advantages:

1. **Speed:** Often 10-100x faster than traditional Python linters
2. **Consolidation:** Replaces multiple tools with one consistent interface
3. **Compatibility:** Implements rules from established tools (Flake8, Black, isort, etc.)
4. **Configuration:** Single configuration in your `pyproject.toml` file
5. **Automatic fixing:** Can automatically fix many issues it identifies

2.2.3 Getting Started with Ruff

First, install Ruff in your virtual environment:

```
# If using pip
pip install ruff

# If using uv
uv pip install ruff
```

2.2.4 Basic Configuration

Configure Ruff in your `pyproject.toml` file:

```
[tool.ruff]
# Enable pycodestyle, Pyflakes, isort, and more
select = ["E", "F", "I"]
ignore = []

# Allow lines to be as long as 100 characters
line-length = 100

# Assume Python 3.10
target-version = "py310"

[tool.ruff.format]
# Formats code similar to Black (this is the default)
quote-style = "double"
indent-style = "space"
line-ending = "auto"
```

This configuration enables: - E rules from pycodestyle (PEP 8 style guide) - F rules from Pyflakes (logical and syntax error detection) - I rules for import sorting (like isort)

2.2.5 Using Ruff in Your Workflow

Ruff provides two main commands:

```
# Check code for issues without changing it
ruff check .

# Format code (similar to Black)
ruff format .
```

To automatically fix issues that Ruff can solve:

```
# Fix all auto-fixable issues
ruff check --fix .
```

2.2.6 Hands-on: Setting Up Ruff Step-by-Step

Let's walk through a practical example that demonstrates Ruff's impact on code quality. Starting with some intentionally messy Python code:

```
# example.py - Before Ruff
import sys,os
from pathlib import Path
import json

def calculate_average(numbers:list)->float:
    return sum(numbers)/len(numbers)

if __name__=='__main__':
    data=[1,2,3,4,5]
    result=calculate_average(data)
    print(f'Average: {result}')
    unused_var = 42
```

This code has several quality issues: - Multiple imports on one line - Inconsistent spacing around operators - Missing spaces in type hints - Unused imports and variables - Inconsistent string quote styles

First, add Ruff to your project:

```
# Add Ruff as a development dependency
uv add --dev ruff
```

Now configure Ruff in your `pyproject.toml`:

```
[tool.ruff]
target-version = "py39"
line-length = 88

[tool.ruff.lint]
# Enable essential rule sets
select = ["E", "F", "I", "W", "B"]
ignore = ["E501"] # Line length handled by formatter

[tool.ruff.format]
quote-style = "double"
```

Run Ruff to identify issues:

```
uv run ruff check example.py
```

This will show output like:

```
example.py:2:1: E401 Multiple imports on one line
example.py:2:8: F401 `sys` imported but unused
example.py:4:1: F401 `json` imported but unused
example.py:15:5: F841 Local variable `unused_var` is assigned to but
never used
```

Apply automatic fixes:

```
uv run ruff check --fix example.py
uv run ruff format example.py
```

After running both commands, your code becomes:

```
# example.py - After Ruff
import os
from pathlib import Path

def calculate_average(numbers: list) -> float:
    return sum(numbers) / len(numbers)

if __name__ == "__main__":
    data = [1, 2, 3, 4, 5]
    result = calculate_average(data)
    print(f"Average: {result}")
```

Notice the improvements: - Unused imports automatically removed - Imports properly sorted and formatted - Consistent spacing around operators and type hints - Proper string quote style - Clean, readable formatting

2.2.7 Integrating Ruff with Pre-commit Hooks

To automatically apply these fixes before each commit, add this to your `.pre-commit-config.yaml`:

```
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.1.11
    hooks:
      - id: ruff
        args: [--fix]
      - id: ruff-format
```

Install and activate the hooks:

```
uv add --dev pre-commit
uv run pre-commit install
```

Now Ruff will automatically clean up your code before each commit, ensuring consistent quality across your entire project.

2.2.8 Real-world Configuration Example

Here's a more comprehensive configuration that balances strictness with practicality:

```
[tool.ruff]
# Target Python version
target-version = "py39"
# Line length
line-length = 88

# Enable a comprehensive set of rules
select = [
    "E", # pycodestyle errors
```

```

    "F", # pyflakes
    "I", # isort
    "W", # pycodestyle warnings
    "C90", # mccabe complexity
    "N", # pep8-naming
    "B", # flake8-bugbear
    "UP", # pyupgrade
    "D", # pydocstyle
]

# Ignore specific rules
ignore = [
    "E203", # Whitespace before ':' (handled by formatter)
    "D100", # Missing docstring in public module
    "D104", # Missing docstring in public package
]

# Exclude certain files/directories from checking
exclude = [
    ".git",
    ".venv",
    "__pycache__",
    "build",
    "dist",
]

[tool.ruff.pydocstyle]
# Use Google-style docstrings
convention = "google"

[tool.ruff.mccabe]
# Maximum McCabe complexity allowed
max-complexity = 10

[tool.ruff.format]
# Formatting options (black-compatible by default)
quote-style = "double"

```

2.2.9 Integrating Ruff into Your Editor

Ruff provides editor integrations for:

- VS Code (via the Ruff extension)
- PyCharm (via third-party plugin)
- Vim/Neovim
- Emacs

For example, in VS Code, install the Ruff extension and add to your settings.json:

```

{
  "editor.formatOnSave": true,
  "editor.codeActionsOnSave": {

```

```

    "source.fixAll.ruff": true,
    "source.organizeImports.ruff": true
  }
}

```

This configuration automatically formats code and fixes issues whenever you save a file.

2.2.10 Gradually Adopting Ruff

If you're working with an existing codebase, you can adopt Ruff gradually:

1. **Start with formatting only:** Begin with `ruff format` to establish consistent formatting
2. **Add basic linting:** Enable a few rule sets like E, F, and I
3. **Gradually increase strictness:** Add more rule sets as your team adjusts
4. **Use per-file ignores:** For specific issues in specific files

```

[tool.ruff.per-file-ignores]
"tests/*" = ["D103"] # Ignore missing docstrings in tests
"__init__.py" = ["F401"] # Ignore unused imports in __init__.py

```

2.2.11 Enforcing Code Quality in CI

Add Ruff to your CI pipeline to ensure code quality standards are maintained:

```

# In your GitHub Actions workflow (.github/workflows/ci.yml)
- name: Check formatting with Ruff
  run: ruff format --check .

- name: Lint with Ruff
  run: ruff check .

```

The `--check` flag on `ruff format` makes it exit with an error if files would be reformatted, instead of actually changing them.

2.2.12 Beyond Ruff: When to Consider Other Tools

While Ruff covers a wide range of code quality checks, some specific needs might require additional tools:

- **mypy** for static type checking (covered in a later section)
- **bandit** for security-focused checks
- **vulture** for finding dead code

However, Ruff's rule set continues to expand, potentially reducing the need for these additional tools over time.

AI Tip: Understanding Linting Errors

When Ruff flags an error you do not understand, paste the rule code into your AI assistant: “*What does Ruff rule B006 mean, and how do I fix it in this code?*” AI is particularly good at explaining why a pattern is problematic and suggesting the

idiomatic alternative. This turns linting errors into learning opportunities.

By incorporating Ruff into your workflow, you'll catch many common errors before they reach production and maintain a consistent, readable codebase. In the next section, we'll explore how to ensure your code works as expected through automated testing with pytest.

2.3 Automated Testing with pytest

Testing is a crucial aspect of software development that ensures your code works as intended and continues to work as you make changes. Python's testing ecosystem offers numerous frameworks, but pytest has emerged as the most popular and powerful choice for most projects.

2.3.1 Why Testing Matters

Automated tests provide several key benefits:

1. **Verification:** Confirm that your code works as expected
2. **Regression prevention:** Catch when changes break existing functionality
3. **Documentation:** Tests demonstrate how code is meant to be used
4. **Refactoring confidence:** Change code structure while ensuring behaviour remains correct
5. **Design feedback:** Difficult-to-test code often indicates design problems

2.3.2 Getting Started with pytest

Add pytest as a development dependency to your project:

```
# Using uv (recommended for our toolchain)
uv add --dev pytest pytest-cov

# Or using pip-tools, add to requirements-dev.in:
# pytest>=7.0.0
# pytest-cov
```

2.3.3 Setting Up a Testing Project Structure

Create a proper test directory structure in your project:

```
# From your project root
mkdir -p tests
touch tests/__init__.py
touch tests/conftest.py # pytest configuration file
```

Your project structure should look like:

```
my-project/
  src/
    my_package/
      __init__.py
      calculations.py
```

```
tests/
  __init__.py
  conftest.py
  test_calculations.py
pyproject.toml
```

2.3.4 Writing Your First Test

Let's assume you have a simple function in `src/my_package/calculations.py`:

```
def add(a, b):
    """Add two numbers and return the result."""
    return a + b
```

Create a test file in `tests/test_calculations.py`:

```
from my_package.calculations import add

def test_add():
    # Test basic addition
    assert add(1, 2) == 3

    # Test with negative numbers
    assert add(-1, 1) == 0
    assert add(-1, -1) == -2

    # Test with floating point
    assert add(1.5, 2.5) == 4.0
```

2.3.5 Running Tests

Run all tests from your project root:

```
# Run all tests
pytest

# Run with more detail
pytest -v

# Run a specific test file
pytest tests/test_calculations.py

# Run a specific test function
pytest tests/test_calculations.py::test_add
```

2.3.6 pytest Features That Make Testing Easier

pytest has several features that make it superior to Python's built-in unittest framework:

2.3.6.1 1. Simple Assertions

Instead of methods like `assertEqual` or `assertTrue`, pytest lets you use Python's built-in `assert` statement, making tests more readable.

```
# With pytest
assert result == expected

# Instead of unittest's
self.assertEqual(result, expected)
```

2.3.6.2 2. Fixtures

Fixtures are a powerful way to set up preconditions for your tests:

```
import pytest
from my_package.database import Database

@pytest.fixture
def db():
    """Provide a clean database instance for tests."""
    db = Database(":memory:") # Use in-memory SQLite
    db.create_tables()
    yield db
    db.close() # Cleanup happens after the test

def test_save_record(db):
    # The db fixture is automatically provided
    record = {"id": 1, "name": "Test"}
    db.save(record)
    assert db.get(1) == record
```

2.3.6.3 3. Parameterized Tests

Test multiple inputs without repetitive code:

```
import pytest
from my_package.calculations import add

@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3),
    (-1, 1, 0),
    (0, 0, 0),
    (1.5, 2.5, 4.0),
])

def test_add_parametrized(a, b, expected):
    assert add(a, b) == expected
```

2.3.6.4 4. Marks for Test organisation

organise tests with marks:

```

@pytest.mark.slow
def test_complex_calculation():
    # This test takes a long time
    ...

# Run only tests marked as 'slow'
# pytest -m slow

@pytest.mark.skip(reason="Feature not implemented yet")
def test_future_feature():
    ...

@pytest.mark.xfail(reason="Known bug #123")
def test_buggy_function():
    ...

```

2.3.7 Test Coverage

Track which parts of your code are tested using `pytest-cov`:

```

# Run tests with coverage report
pytest --cov=src/my_package

# Generate HTML report for detailed analysis
pytest --cov=src/my_package --cov-report=html
# Then open htmlcov/index.html in your browser

```

A coverage report helps identify untested code:

```

----- coverage: platform linux, python 3.9.5-final-0 -----
Name                               Stmts  Miss  Cover
-----
src/my_package/__init__.py          1      0  100%
src/my_package/calculations.py     10      2   80%
src/my_package/models.py           45     15   67%
-----
TOTAL                               56     17   70%

```

2.3.8 Configuring pytest for Your Project

Set up `pytest` configuration in your `pyproject.toml` to customise default behaviour:

```

[tool.pytest.ini_options]
# Test discovery paths
testpaths = ["tests"]

# Default options (applied to every pytest run)
addopts = [
    "--cov=src",          # Enable coverage for src directory
    "--cov-report=term-missing", # Show missing lines in terminal
    "--cov-report=html",  # Generate HTML coverage report
    "--strict-markers",   # Require all markers to be defined

```

```

    "--disable-warnings",      # Suppress warnings for cleaner output
]

# Define custom test markers
markers = [
    "slow: marks tests as slow (deselect with '-m \"not slow\"')",
    "integration: marks tests as integration tests",
    "unit: marks tests as unit tests",
]

# Minimum coverage percentage (tests fail if below this)
# addopts = ["--cov=src", "--cov-fail-under=80"]

```

This configuration provides several benefits:

1. **Automatic coverage:** Every test run includes coverage reporting
2. **Clean output:** Suppresses unnecessary warnings while still showing errors
3. **Test organisation:** Markers help categorize and selectively run tests
4. **Consistent behaviour:** Same settings for all developers

With this configuration, running `uv run pytest` automatically: - Discovers tests in the `tests/` directory - Calculates code coverage for your `src/` directory - Generates both terminal and HTML coverage reports - Applies your chosen settings consistently

2.3.9 Testing Best Practices

1. **Write tests as you develop:** Don't wait until the end
2. **Name tests clearly:** Include the function name and scenario being tested
3. **One assertion per test:** Focus each test on a single behaviour
4. **Test edge cases:** Empty input, boundary values, error conditions
5. **Avoid test interdependence:** Tests should work independently
6. **Mock external dependencies:** APIs, databases, file systems
7. **Keep tests fast:** Slow tests get run less often

2.3.10 Common Testing Patterns

2.3.10.1 Testing Exceptions

Verify that your code raises the right exceptions:

```

import pytest
from my_package.validate import validate_username

def test_validate_username_too_short():
    with pytest.raises(ValueError) as excinfo:
        validate_username("ab") # Too short
    assert "Username must be at least 3 characters" in
        str(excinfo.value)

```

2.3.10.2 Testing with Temporary Files

Test file operations safely:

```
def test_save_to_file(tmp_path):
    # tmp_path is a built-in pytest fixture
    file_path = tmp_path / "test.txt"

    # Test file writing
    save_to_file(file_path, "test content")

    # Verify content
    assert file_path.read_text() == "test content"
```

2.3.10.3 Mocking

Isolate your code from external dependencies using the `pytest-mock` plugin:

```
def test_fetch_user_data(mockeer):
    # Mock the API call
    mock_response = mockeer.patch('requests.get')
    mock_response.return_value.json.return_value = {"id": 1, "name":
    "Test User"}

    # Test our function
    from my_package.api import get_user
    user = get_user(1)

    # Verify results
    assert user['name'] == "Test User"

mock_response.assert_called_once_with('https://api.example.com/users/1')
```

2.3.11 Testing Strategy

As your project grows, organise tests into different categories:

1. **Unit tests:** Test individual functions/classes in isolation
2. **Integration tests:** Test interactions between components
3. **Functional tests:** Test entire features from a user perspective
4. **End-to-end (E2E) tests:** Test the full application as a user would interact with it

Most projects should have a pyramid shape: many unit tests, fewer integration tests, and even fewer functional/E2E tests.

AI Tip: Generating Tests

AI excels at generating test cases, especially edge cases you might miss. Try: *“Write pytest tests for this function, including edge cases for empty input, None values, and boundary conditions.”* Then paste your function. The key is reviewing what AI generates — it often writes tests that pass trivially or miss the actual business logic. Use AI-generated tests as a starting point, not a finished product.

💡 GUI and End-to-End Testing with Playwright

When your project has a web frontend or a desktop interface (such as an Electron app), consider adding end-to-end tests using Playwright. Playwright lets you automate real browser interactions—clicking buttons, filling forms, verifying that pages render correctly—giving you confidence that your application works as users will actually experience it.

We cover Playwright in detail in the Multi-Platform Distribution chapter, where we show how to test web deployments, PWAs, and Electron desktop applications.

2.3.12 Continuous Testing

Make testing a habitual part of your workflow:

1. **Run relevant tests as you code:** Many editors integrate with pytest
2. **Run full test suite before committing:** Use pre-commit hooks
3. **Run tests in CI:** Catch issues that might only appear in different environments

By incorporating comprehensive testing into your development process, you'll catch bugs earlier, ship with more confidence, and build a more maintainable codebase.

In the next section, we'll explore static type checking with mypy, which can help catch a whole new category of errors before your code even runs.

2.4 Type Checking with mypy

Python is dynamically typed, which provides flexibility but can also lead to type-related errors that only appear at runtime. Static type checking with mypy adds an extra layer of verification, catching many potential issues before your code executes.

2.4.1 Understanding Type Hints

Python 3.5+ supports type hints, which are annotations indicating what types of values functions expect and return:

```
def greeting(name: str) -> str:
    return f"Hello, {name}!"
```

These annotations don't change how Python runs—they're ignored by the interpreter at runtime. However, tools like mypy can analyse them statically to catch potential type errors.

2.4.2 Getting Started with mypy

First, install mypy in your development environment:

```
pip install mypy
```

Let's check a simple example:

```
# example.py
def double(x: int) -> int:
    return x * 2
```

```
# This is fine
result = double(5)

# This would fail at runtime
double("hello")
```

Run mypy to check:

```
mypy example.py
```

Output:

```
example.py:8: error: Argument 1 to "double" has incompatible type "str";
expected "int"
```

mypy caught the type mismatch without running the code!

2.4.3 Configuring mypy

Configure mypy in your `pyproject.toml` file for a consistent experience:

```
[tool.mypy]
python_version = "3.9"
warn_return_any = true
warn_unused_configs = true
disallow_untyped_defs = false
disallow_incomplete_defs = false
```

Start with a lenient configuration and gradually increase strictness:

```
# Starting configuration: permissive but helpful
[tool.mypy]
python_version = "3.9"
warn_return_any = true
check_untyped_defs = true
disallow_untyped_defs = false

# Intermediate configuration: more rigorous
[tool.mypy]
python_version = "3.9"
warn_return_any = true
disallow_incomplete_defs = true
disallow_untyped_defs = false
check_untyped_defs = true

# Strict configuration: full typing required
[tool.mypy]
python_version = "3.9"
disallow_untyped_defs = true
disallow_incomplete_defs = true
no_implicit_optional = true
warn_redundant_casts = true
```

```
warn_unused_ignores = true
warn_return_any = true
warn_unreachable = true
```

2.4.4 Gradual Typing

One major advantage of Python's type system is gradual typing—you can add types incrementally:

1. Start with critical or error-prone modules
2. Add types to public interfaces first
3. Increase type coverage over time

2.4.5 Essential Type Annotations

2.4.5.1 Basic Types

```
# Variables
name: str = "Alice"
age: int = 30
height: float = 1.75
is_active: bool = True

# Lists, sets, and dictionaries
names: list[str] = ["Alice", "Bob"]
unique_ids: set[int] = {1, 2, 3}
user_scores: dict[str, int] = {"Alice": 100, "Bob": 85}
```

2.4.5.2 Function Annotations

```
def calculate_total(prices: list[float], tax_rate: float = 0.0) -> float:
    """Calculate the total price including tax."""
    subtotal = sum(prices)
    return subtotal * (1 + tax_rate)
```

2.4.5.3 Class Annotations

```
from typing import Optional

class User:
    def __init__(self, name: str, email: str, age: Optional[int] = None):
        self.name: str = name
        self.email: str = email
        self.age: Optional[int] = age

    def is_adult(self) -> bool:
        """Check if user is an adult."""
        return self.age is not None and self.age >= 18
```

2.4.6 Advanced Type Hints

2.4.6.1 Union Types

Use Union to indicate multiple possible types (use the | operator in Python 3.10+):

```
from typing import Union

# Python 3.9 and earlier
def process_input(data: Union[str, list[str]]) -> str:
    if isinstance(data, list):
        return ", ".join(data)
    return data

# Python 3.10+
def process_input(data: str | list[str]) -> str:
    if isinstance(data, list):
        return ", ".join(data)
    return data
```

2.4.6.2 Optional and None

Optional[T] is equivalent to Union[T, None] or T | None:

```
from typing import Optional

def find_user(user_id: int) -> Optional[dict]:
    """Return user data or None if not found."""
    # Implementation...
```

2.4.6.3 Type Aliases

Create aliases for complex types:

```
from typing import Dict, List, Tuple

# Complex type
TransactionRecord = Tuple[str, float, str, Dict[str, str]]

# More readable with alias
def process_transactions(transactions: List[TransactionRecord]) -> float:
    total = 0.0
    for _, amount, _, _ in transactions:
        total += amount
    return total
```

2.4.6.4 Callable

Type hint for functions:

```

from typing import Callable

def apply_function(func: Callable[[int], str], value: int) -> str:
    """Apply a function that converts int to str."""
    return func(value)

```

2.4.7 Common Challenges and Solutions

2.4.7.1 Working with Third-Party Libraries

Not all libraries provide type hints. For popular packages, you can often find stub files:

```
pip install types-requests
```

For others, you can silence mypy warnings selectively:

```
import untyped_library # type: ignore
```

2.4.7.2 Dealing with Dynamic Features

Python's dynamic features can be challenging to type. Use `Any` when necessary:

```

from typing import Any, Dict

def parse_config(config: Dict[str, Any]) -> Dict[str, Any]:
    """Parse configuration with unknown structure."""
    # Implementation...

```

2.4.8 Integration with Your Workflow

2.4.8.1 Running mypy

```

# Check a specific file
mypy src/my_package/module.py

# Check the entire package
mypy src/my_package/

# Use multiple processes for faster checking
mypy -p my_package --python-version 3.9 --multiprocessing

```

2.4.8.2 Integrating with CI/CD

Add mypy to your continuous integration workflow:

```

# GitHub Actions example
- name: Type check with mypy
  run: mypy src/

```

2.4.8.3 Editor Integration

Most Python-friendly editors support mypy:

- VS Code: Use the Pylance extension
- PyCharm: Has built-in type checking
- vim/neovim: Use ALE or similar plugins

2.4.9 The Broader Type Checking Landscape

While mypy remains the most widely adopted and beginner-friendly type checker, Python’s type checking ecosystem is rapidly evolving. Other notable options include:

- **pyright/pylance**: Microsoft’s fast, strict type checker that powers VS Code’s Python extension
- **basedmypy**: A mypy fork with stricter defaults and additional features
- **basedpyright**: An even more aggressive fork of pyright
- **ty**: Astral’s upcoming type checker (from the makers of ruff and uv), with an alpha preview expected by PyCon 2025

For learning and establishing good type annotation habits, mypy provides an excellent foundation with extensive documentation and community support. As your expertise grows, you can explore these alternatives to find the right balance of speed, strictness, and features for your projects.

2.4.10 Benefits of Type Checking

1. **Catch errors early**: Find type-related bugs before running code
2. **Improved IDE experience**: Better code completion and refactoring
3. **Self-documenting code**: Types serve as documentation
4. **Safer refactoring**: Change code with more confidence
5. **Gradual adoption**: Add types where they provide the most value

2.4.11 When to Use Type Hints

Type hints are particularly valuable for:

- Functions with complex parameters or return values
- Public APIs used by others
- Areas with frequent bugs
- Critical code paths
- Large codebases with multiple contributors

Type checking isn’t an all-or-nothing proposition. Even partial type coverage can significantly improve code quality and catch common errors. Start small, focus on interfaces, and expand your type coverage as your team becomes comfortable with the system.

AI Tip: Adding Type Hints to Existing Code

AI is excellent at adding type hints to untyped code. Paste a function and ask: “Add type hints to this function, including the return type.” AI can also help you understand complex type annotations like `Optional[dict[str, list[int]]]`. For larger codebases, use AI to type-annotate one module at a time rather than trying to do everything at once.

2.5 Security Analysis with Bandit

Software security is a critical concern in modern development, yet it's often overlooked until problems arise. Bandit is a tool designed to find common security issues in Python code through static analysis.

2.5.1 Understanding Security Static Analysis

Unlike functional testing or linting, security-focused static analysis looks specifically for patterns and practices that could lead to security vulnerabilities:

- Injection vulnerabilities
- Use of insecure functions
- Hardcoded credentials
- Insecure cryptography
- And many other security issues

2.5.2 Getting Started with Bandit

First, install Bandit in your virtual environment:

```
pip install bandit
```

Run a basic scan:

```
# Scan a specific file
bandit -r src/my_package/main.py

# Scan your entire codebase
bandit -r src/
```

2.5.3 Security Issues Bandit Can Detect

Bandit identifies a wide range of security concerns, including:

2.5.3.1 1. Hardcoded Secrets

```
# Bandit will flag this
def connect_to_database():
    password = "super_secret_password" # Hardcoded secret
    return Database("user", password)
```

2.5.3.2 2. SQL Injection

```
# Vulnerable to SQL injection
def get_user(username):
    query = f"SELECT * FROM users WHERE username = '{username}'"
    return db.execute(query)

# Safer approach
def get_user_safe(username):
    query = "SELECT * FROM users WHERE username = %s"
```

```
return db.execute(query, (username,))
```

2.5.3.3 3. Shell Injection

```
# Vulnerable to command injection
def run_command(user_input):
    return os.system(f"ls {user_input}") # User could inject commands

# Safer approach
import subprocess
def run_command_safe(user_input):
    return subprocess.run(["ls", user_input], capture_output=True,
                          text=True)
```

2.5.3.4 4. Insecure Cryptography

```
# Using weak hash algorithms
import hashlib
def hash_password(password):
    return hashlib.md5(password.encode()).hexdigest() # MD5 is insecure
```

2.5.3.5 5. Unsafe Deserialization

```
# Insecure deserialization
import pickle
def load_user_preferences(data):
    return pickle.loads(data) # Pickle can execute arbitrary code
```

2.5.4 Configuring Bandit

You can configure Bandit using a `.bandit` file or your `pyproject.toml`:

```
[tool.bandit]
exclude_dirs = ["tests", "docs"]
skips = ["B311"] # Skip random warning
targets = ["src"]
```

The most critical findings are categorized with high severity and confidence levels:

```
# Only report high-severity issues
bandit -r src/ -iii -ll
```

2.5.5 Integrating Bandit in Your Workflow

2.5.5.1 Add Bandit to CI/CD

Add security scanning to your continuous integration pipeline:

```
# GitHub Actions example
- name: Security check with Bandit
```

```
run: bandit -r src/ -f json -o bandit-results.json

# Optional: convert results to GitHub Security format
# (requires additional tools or post-processing)
```

2.5.5.2 Pre-commit Hook

Configure a pre-commit hook to run Bandit before commits:

```
# In .pre-commit-config.yaml
- repo: https://github.com/PyCQA/bandit
  rev: 1.7.5
  hooks:
    - id: bandit
      args: ["-r", "src"]
```

2.5.6 Responding to Security Findings

When Bandit identifies security issues:

1. **Understand the risk:** Read the detailed explanation to understand the potential vulnerability
2. **Fix high-severity issues immediately:** These represent significant security risks
3. **Document deliberate exceptions:** If a finding is a false positive, document why and use an inline ignore comment
4. **Review regularly:** Security standards evolve, so regular scanning is essential

2.5.7 False Positives

Like any static analysis tool, Bandit can produce false positives. You can exclude specific findings:

```
# In code, to ignore a specific line
import pickle # nosec

# For a whole file
# nosec

# Or configure globally in pyproject.toml
```

By incorporating security scanning with Bandit, you add an essential layer of protection against common security vulnerabilities, helping to ensure that your code is not just functional but also secure.

2.6 Finding Dead Code with Vulture

As projects evolve, code can become obsolete but remain in the codebase, creating maintenance burdens and confusion. Vulture is a static analysis tool that identifies unused code – functions, classes, and variables that are defined but never used.

2.6.1 The Problem of Dead Code

Dead code creates several issues:

1. **Maintenance overhead:** Every line of code needs maintenance
2. **Cognitive load:** Developers need to understand code that serves no purpose
3. **False security:** Tests might pass while dead code goes unchecked
4. **Misleading documentation:** Dead code can appear in documentation generators

2.6.2 Getting Started with Vulture

Install Vulture in your virtual environment:

```
pip install vulture
```

Run a basic scan:

```
# Scan a specific file
vulture src/my_package/main.py

# Scan your entire codebase
vulture src/
```

2.6.3 What Vulture Detects

Vulture identifies:

2.6.3.1 1. Unused Variables

```
def process_data(data):
    result = [] # Defined but never used
    for item in data:
        processed = transform(item) # Unused variable
        data.append(item * 2)
    return data
```

2.6.3.2 2. Unused Functions

```
def calculate_average(numbers):
    """Calculate the average of a list of numbers."""
    if not numbers:
        return 0
    return sum(numbers) / len(numbers)

# If this function is never called anywhere, Vulture will flag it
```

2.6.3.3 3. Unused Classes

```
class LegacyFormatter:
    """Format data using the legacy method."""
    def __init__(self, data):
```

```

        self.data = data

    def format(self):
        return json.dumps(self.data)

# If this class is never instantiated, Vulture will flag it

```

2.6.3.4 4. Unused Imports

```

import os
import sys # If sys is imported but never used
import json
from datetime import datetime, timedelta # If timedelta is never used

```

2.6.4 Handling False Positives

Vulture can sometimes flag code that's actually used but in ways it can't detect. Common cases include:

- Classes used through reflection
- Functions called in templates
- Code used in an importable public API

You can create a whitelist file to suppress these reports:

```

# whitelist.py
# unused_function # vulture:ignore

```

Run Vulture with the whitelist:

```

vulture src/ whitelist.py

```

2.6.5 Configuration and Integration

Add Vulture to your workflow:

2.6.5.1 Command Line Options

```

# Set minimum confidence (default is 60%)
vulture --min-confidence 80 src/

# Exclude test files
vulture src/ --exclude "test_*.py"

```

2.6.5.2 CI Integration

```

# GitHub Actions example
- name: Find dead code with Vulture
  run: vulture src/ --min-confidence 80

```

2.6.6 Best Practices for Dead Code Removal

1. **Verify before removing:** Confirm the code is truly unused
2. **Use version control:** Remove code through proper commits with explanations
3. **Update documentation:** Ensure documentation reflects the changes
4. **Run tests:** Confirm nothing breaks when the code is removed
5. **Look for patterns:** Clusters of dead code often indicate larger architectural issues

2.6.7 When to Run Vulture

- Before major refactoring
- During codebase cleanup
- As part of regular maintenance
- When preparing for a significant release
- When onboarding new team members (helps them focus on what matters)

Regularly checking for and removing dead code keeps your codebase lean and maintainable. It also provides insights into how your application has evolved and may highlight areas where design improvements could be made.

With these additional security and code quality tools in place, your Python development workflow is now even more robust. Let's move on to Part 3, where we'll explore documentation and deployment options.

Chapter 3

Documentation and Deployment

3.1 Documentation Options: From pydoc to MkDocs

Documentation is often neglected in software development, yet it's crucial for ensuring others (including your future self) can understand and use your code effectively. Python offers a spectrum of documentation options, from simple built-in tools to sophisticated documentation generators.

3.1.1 Starting Simple with Docstrings

The foundation of Python documentation is the humble docstring - a string literal that appears as the first statement in a module, function, class, or method:

```
def calculate_discount(price: float, discount_percent: float) -> float:
    """Calculate the discounted price.

    Args:
        price: The original price
        discount_percent: The discount percentage (0-100)

    Returns:
        The price after discount

    Raises:
        ValueError: If discount_percent is negative or greater than 100
    """
    if not 0 <= discount_percent <= 100:
        raise ValueError("Discount percentage must be between 0 and 100")

    discount = price * (discount_percent / 100)
    return price - discount
```

Docstrings become particularly useful when following a consistent format. Common

conventions include:

- **Google style** (shown above)
- **NumPy style** (similar to Google style but with different section headers)
- **reStructuredText** (used by Sphinx)

3.1.2 Viewing Documentation with pydoc

Python's built-in pydoc module provides a simple way to access documentation:

```
# View module documentation in the terminal
python -m pydoc my_package.module

# Start an HTTP server to browse documentation
python -m pydoc -b
```

You can also generate basic HTML documentation:

```
# Create HTML for a specific module
python -m pydoc -w my_package.module

# Create HTML for an entire package
mkdir -p docs/html
python -m pydoc -w my_package
mv my_package*.html docs/html/
```

While simple, this approach has limitations: - Minimal styling - No cross-linking between documents - Limited navigation options

For beginner projects, however, it provides a fast way to make documentation available with zero dependencies.

3.1.3 Simple Script for Basic Documentation Site

For slightly more organised documentation than plain pydoc, you can create a simple script that: 1. Generates pydoc HTML for all modules 2. Creates a basic index.html linking to them

Here's a minimal example script (build_docs.py):

```
import os
import importlib
import pkgutil
import pydoc

def generate_docs(package_name, output_dir="docs/api"):
    """Generate HTML documentation for all modules in a package."""
    # Ensure output directory exists
    os.makedirs(output_dir, exist_ok=True)

    # Import the package
    package = importlib.import_module(package_name)

    # Track all modules for index page
    modules = []
```

```

# Walk through all modules in the package
for _, modname, ispkg in pkgutil.walk_packages(package.__path__,
package_name + '.'):
    try:
        # Generate HTML documentation
        html_path = os.path.join(output_dir, modname + '.html')
        with open(html_path, 'w') as f:
            pydoc_output =
pydoc.HTMLDoc().document(importlib.import_module(modname))
            f.write(pydoc_output)

        modules.append((modname, os.path.basename(html_path)))
        print(f"Generated documentation for {modname}")
    except ImportError as e:
        print(f"Error importing {modname}: {e}")

# Create index.html
index_path = os.path.join(output_dir, 'index.html')
with open(index_path, 'w') as f:
    f.write("<html><head><title>API
Documentation</title></head><body>\n")
    f.write("<h1>API Documentation</h1>\n<ul>\n")

    for modname, html_file in sorted(modules):
        f.write(f'<li><a href="{html_file}">{modname}</a></li>\n')

    f.write("</ul></body></html>")

print(f"Index created at {index_path}")

if __name__ == "__main__":
    # Change 'my_package' to your actual package name
    generate_docs('my_package')

```

This script generates slightly more organised documentation than raw pydoc but still leverages built-in tools.

3.1.4 Moving to MkDocs for Comprehensive Documentation

When your project grows and needs more sophisticated documentation, MkDocs provides an excellent balance of simplicity and features. MkDocs generates a static site from Markdown files, making it easy to write and maintain documentation.

3.1.4.1 Getting Started with MkDocs

First, install MkDocs and a theme:

```
pip install mkdocs mkdocs-material
```

Initialize a new documentation project:

```
mkdocs new .
```

This creates a `mkdocs.yml` configuration file and a `docs/` directory with an `index.md` file.

3.1.4.2 Basic Configuration

Edit `mkdocs.yml`:

```
site_name: My Project
theme:
  name: material
  palette:
    primary: indigo
    accent: indigo
nav:
  - Home: index.md
  - User Guide:
    - Installation: user-guide/installation.md
    - Getting Started: user-guide/getting-started.md
  - API Reference: api-reference.md
  - Contributing: contributing.md
```

3.1.4.3 Creating Documentation Content

MkDocs uses Markdown files for content. Create `docs/user-guide/installation.md`:

```
# Installation

## Prerequisites

- Python 3.8 or later
- pip package manager

## Installation Steps

1. Install from PyPI:

    pip install my-package

2. Verify installation:

    python -c "import my_package; print(my_package.__version__)"
```

3.1.4.4 Testing Documentation Locally

Preview your documentation while writing:

```
mkdocs serve
```

This starts a development server at `http://127.0.0.1:8000` that automatically refreshes when you update files.

3.1.4.5 Building and Deploying Documentation

Generate static HTML files:

```
mkdocs build
```

This creates a `site/` directory with the HTML documentation site.

For GitHub projects, you can publish to GitHub Pages:

```
mkdocs gh-deploy
```

3.1.5 Hosting Documentation with GitHub Pages

GitHub Pages provides a simple, free hosting solution for your project documentation that integrates seamlessly with your GitHub repository.

3.1.5.1 Setting Up GitHub Pages

There are two main approaches to hosting documentation on GitHub Pages:

1. **Repository site:** Serves content from a dedicated branch (typically `gh-pages`)
2. **User/organisation site:** Serves content from a special repository named `username.github.io`

For most Python projects, the repository site approach works best:

1. Go to your repository on GitHub
2. Navigate to Settings → Pages
3. Under “Source”, select your branch (either `main` or `gh-pages`)
4. Choose the folder that contains your documentation (`/` or `/docs`)
5. Click Save

Your documentation will be published at `https://username.github.io/repository-name/`.

3.1.5.2 Automating Documentation Deployment

MkDocs has built-in support for GitHub Pages deployment:

```
# Build and deploy documentation to GitHub Pages
mkdocs gh-deploy
```

This command: 1. Builds your documentation into the `site/` directory 2. Creates or updates the `gh-pages` branch 3. Pushes the built site to that branch 4. GitHub automatically serves the content

For a fully automated workflow, integrate this into your GitHub Actions CI pipeline:

```
name: Deploy Documentation

on:
  push:
    branches:
      - main
  paths:
    - 'docs/**'
```

```

- 'mkdocs.yml'

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install mkdocs mkdocs-material mkdocstrings[python]
      - name: Deploy documentation
        run: mkdocs gh-deploy --force

```

This workflow automatically deploys your documentation whenever you push changes to documentation files on the main branch.

3.1.5.3 GitHub Pages with pydoc

Even if you're using the simpler pydoc approach, you can still host the generated HTML on GitHub Pages:

1. Create a docs/ folder in your repository
2. Generate HTML documentation with pydoc:

```
python -m pydoc -w src/my_package/*.py
mv *.html docs/
```

3. Add a simple docs/index.html that links to your module documentation
4. Configure GitHub Pages to serve from the docs/ folder of your main branch

3.1.5.4 Custom Domains

For more established projects, you can use your own domain:

1. Purchase a domain from a registrar
2. Add a CNAME file to your documentation with your domain name
3. Configure your DNS settings according to GitHub's instructions
4. Enable HTTPS in GitHub Pages settings

By hosting your documentation on GitHub Pages, you make it easily accessible to users and maintainable alongside your codebase. It's a natural extension of the Git-based workflow we've established.

3.1.5.5 Enhancing MkDocs

MkDocs supports numerous plugins and extensions:

- **Code highlighting:** Built-in support for syntax highlighting

- **Admonitions:** Create warning, note, and info boxes
- **Search:** Built-in search functionality
- **Table of contents:** Automatic generation of section navigation

Example of enhanced configuration:

```
site_name: My Project
theme:
  name: material
  features:
    - navigation.instant
    - navigation.tracking
    - navigation.expand
    - navigation.indexes
    - content.code.annotate
markdown_extensions:
  - admonition
  - pymdownx.highlight
  - pymdownx.superfences
  - toc:
      permalink: true
plugins:
  - search
  - mkdocstrings:
      handlers:
        python:
          selection:
            docstring_style: google
```

3.1.6 Integrating API Documentation

MkDocs alone is great for manual documentation, but you can also integrate auto-generated API documentation:

3.1.6.1 Using mkdocstrings

Install mkdocstrings to include docstrings from your code:

```
pip install mkdocstrings[python]
```

Update mkdocs.yml:

```
plugins:
  - search
  - mkdocstrings:
      handlers:
        python:
          selection:
            docstring_style: google
```

Then in your docs/api-reference.md:

```
# API Reference
```

```

## Module my_package.core

This module contains core functionality.

::: my_package.core
  options:
    show_source: false

```

This automatically generates documentation from docstrings in your `my_package.core` module.

3.1.7 Documentation Best Practices

Regardless of which documentation tool you choose, follow these best practices:

1. **Start with a clear README:** Include installation, quick start, and basic examples
2. **Document as you code:** Write documentation alongside code, not as an afterthought
3. **Include examples:** Show how to use functions and classes with realistic examples
4. **Document edge cases and errors:** Explain what happens in exceptional situations
5. **Keep documentation close to code:** Use docstrings for API details
6. **Maintain a changelog:** Track major changes between versions
7. **Consider different audiences:** Write for both new users and experienced developers

3.1.8 Choosing the Right Documentation Approach

Approach	When to Use
Docstrings only	For very small, personal projects
pydoc	For simple projects with minimal documentation needs
Custom pydoc script	Small to medium projects needing basic organisation
MkDocs	Medium to large projects requiring structured, attractive documentation
Sphinx	Large, complex projects, especially with scientific or mathematical content

For most applications, the journey often progresses from simple docstrings to MkDocs as the project grows. By starting with good docstrings from the beginning, you make each subsequent step easier.

AI Tip: Documentation Generation

AI is excellent at writing docstrings and README content. Try: “*Write Google-style docstrings for all the public functions in this module*” and paste your code. For project-level documentation, ask AI to draft a README based on your

pyproject.toml and source code. The key is providing AI with your actual code — generic documentation prompts produce generic results. Always review for accuracy, especially around installation steps and API descriptions.

In the next section, we'll explore how to automate your workflow with CI/CD using GitHub Actions.

3.2 CI/CD Workflows with GitHub Actions

Continuous Integration (CI) and Continuous Deployment (CD) automate the process of testing, building, and deploying your code, ensuring quality and consistency throughout the development lifecycle. GitHub Actions provides a powerful and flexible way to implement CI/CD workflows directly within your GitHub repository.

3.2.1 Understanding CI/CD Basics

Before diving into implementation, let's understand what each component achieves:

- **Continuous Integration:** Automatically testing code changes when pushed to the repository
- **Continuous Deployment:** Automatically deploying code to testing, staging, or production environments

A robust CI/CD pipeline typically includes:

1. Running tests
2. Verifying code quality (formatting, linting)
3. Static analysis (type checking, security scanning)
4. Building documentation
5. Building and publishing packages or applications
6. Deploying to environments

3.2.2 Setting Up GitHub Actions

GitHub Actions workflows are defined using YAML files stored in the `.github/workflows/` directory of your repository. Each workflow file defines a set of jobs and steps that execute in response to specified events.

Start by creating the directory structure:

```
mkdir -p .github/workflows
```

3.2.3 Basic Python CI Workflow

Let's create a file named `.github/workflows/ci.yml`:

```
name: Python CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

```

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ["3.8", "3.9", "3.10"]

    steps:
      - uses: actions/checkout@v3

      - name: Set up Python ${ matrix.python-version }
        uses: actions/setup-python@v4
        with:
          python-version: ${ matrix.python-version }
          cache: pip

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
          pip install -r requirements-dev.txt

      - name: Check formatting with Ruff
        run: ruff format --check .

      - name: Lint with Ruff
        run: ruff check .

      - name: Type check with mypy
        run: mypy src/

      - name: Run security checks with Bandit
        run: bandit -r src/ -x tests/

      - name: Test with pytest
        run: pytest --cov=src/ --cov-report=xml

      - name: Upload coverage to Codecov
        uses: codecov/codecov-action@v3
        with:
          file: ./coverage.xml
          fail_ci_if_error: true

```

This workflow:

1. Triggers on pushes to `main` and on pull requests
2. Runs on the latest Ubuntu environment
3. Tests against multiple Python versions
4. Sets up caching to speed up dependency installation
5. Runs our full suite of quality checks and tests
6. Uploads coverage reports to Codecov (if you've set up this integration)

3.2.4 Using Dependency Caching

To speed up your workflow, GitHub Actions provides caching capabilities:

```
- name: Set up Python ${ matrix.python-version }
  uses: actions/setup-python@v4
  with:
    python-version: ${ matrix.python-version }
    cache: pip # Enable pip caching
```

For more specific control over caching:

```
- name: Cache pip packages
  uses: actions/cache@v3
  with:
    path: ~/.cache/pip
    key: ${ runner.os }}-pip-${ hashFiles('**/requirements*.txt') }}
    restore-keys: |
      ${ runner.os }}-pip-
```

3.2.5 Adapting for Different Dependency Tools

If you're using uv instead of pip, adjust your workflow:

```
- name: Install uv
  run: curl -Lsf https://astral.sh/uv/install.sh | sh

- name: Install dependencies with uv
  run: |
    uv pip sync requirements.txt requirements-dev.txt
```

3.2.6 Building and Publishing Documentation

Add a job to build documentation with MkDocs:

```
docs:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: "3.10"

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install mkdocs mkdocs-material mkdocstrings[python]

    - name: Build documentation
      run: mkdocs build --strict
```

```

- name: Deploy to GitHub Pages
  if: github.event_name == 'push' && github.ref == 'refs/heads/main'
  uses: peaceiris/actions-gh-pages@v3
  with:
    github_token: ${ secrets.GITHUB_TOKEN }
    publish_dir: ./site

```

This job builds your documentation with MkDocs and deploys it to GitHub Pages when changes are pushed to the main branch.

3.2.7 Building and Publishing Python Packages

For projects that produce packages, add a job for publication to PyPI:

```

publish:
  needs: [test, docs] # Only run if test and docs jobs pass
  runs-on: ubuntu-latest
  # Only publish on tagged releases
  if: github.event_name == 'push' && startsWith(github.ref, 'refs/tags')
  steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: "3.10"

    - name: Install build dependencies
      run: |
        python -m pip install --upgrade pip
        pip install build twine

    - name: Build package
      run: python -m build

    - name: Check package with twine
      run: twine check dist/*

    - name: Publish package
      uses: pypa/gh-action-pypi-publish@release/v1
      with:
        user: __token__
        password: ${ secrets.PYPI_API_TOKEN }

```

This job: 1. Only runs after tests and documentation have passed 2. Only triggers on tagged commits (releases) 3. Builds the package using the `build` package 4. Validates the package with `twine` 5. Publishes to PyPI using a secure token

You would need to add the `PYPI_API_TOKEN` to your repository secrets.

3.2.8 Running Tests in Multiple Environments

For applications that need to support multiple operating systems or Python versions:

```
test:
  runs-on: ${{ matrix.os }}
  strategy:
    matrix:
      os: [ubuntu-latest, windows-latest, macos-latest]
      python-version: ["3.8", "3.9", "3.10"]

  steps:
    # ... Steps as before ...
```

This configuration runs your tests on three operating systems with three Python versions each, for a total of nine environments.

3.2.9 Branch Protection and Required Checks

To ensure code quality, set up branch protection rules on GitHub:

1. Go to your repository → Settings → Branches
2. Add a rule for your main branch
3. Enable “Require status checks to pass before merging”
4. Select the checks from your CI workflow

This prevents merging pull requests until all tests pass, maintaining your code quality standards.

3.2.10 Scheduled Workflows

Run your tests on a schedule to catch issues with external dependencies:

```
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  schedule:
    - cron: '0 0 * * 0' # Weekly on Sundays at midnight
```

3.2.11 Notifications and Feedback

Configure notifications for workflow results:

```
- name: Send notification
  if: always()
  uses: rtCamp/action-slack-notify@v2
  env:
    SLACK_WEBHOOK: ${{ secrets.SLACK_WEBHOOK }}
    SLACK_TITLE: CI Result
    SLACK_MESSAGE: ${{ job.status }}
    SLACK_COLOR: ${{ job.status == 'success' && 'good' || 'danger' }}
```

This example sends notifications to Slack, but similar actions exist for other platforms.

3.2.12 A Complete CI/CD Workflow Example

Here's a comprehensive workflow example bringing together many of the concepts we've covered:

```

name: Python CI/CD Pipeline

on:
  push:
    branches: [ main ]
    tags: [ 'v*' ]
  pull_request:
    branches: [ main ]
  schedule:
    - cron: '0 0 * * 0' # Weekly on Sundays

jobs:
  quality:
    name: Code Quality
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.10"
          cache: pip

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements-dev.txt

      - name: Check formatting
        run: ruff format --check .

      - name: Lint with Ruff
        run: ruff check .

      - name: Type check
        run: mypy src/

      - name: Security scan
        run: bandit -r src/ -x tests/

      - name: Check for dead code
        run: vulture src/ --min-confidence 80

  test:

```

```
name: Test
needs: quality
runs-on: ${{ matrix.os }}
strategy:
  matrix:
    os: [ubuntu-latest]
    python-version: ["3.8", "3.9", "3.10"]
    include:
      - os: windows-latest
        python-version: "3.10"
      - os: macos-latest
        python-version: "3.10"

steps:
  - uses: actions/checkout@v3

  - name: Set up Python ${{ matrix.python-version }}
    uses: actions/setup-python@v4
    with:
      python-version: ${{ matrix.python-version }}
      cache: pip

  - name: Install dependencies
    run: |
      python -m pip install --upgrade pip
      pip install -r requirements.txt -r requirements-dev.txt

  - name: Test with pytest
    run: pytest --cov=src/ --cov-report=xml

  - name: Upload coverage
    uses: codecov/codecov-action@v3
    with:
      file: ./coverage.xml

docs:
name: Documentation
needs: quality
runs-on: ubuntu-latest
steps:
  - uses: actions/checkout@v3

  - name: Set up Python
    uses: actions/setup-python@v4
    with:
      python-version: "3.10"

  - name: Install docs dependencies
    run: |
      python -m pip install --upgrade pip
      pip install mkdocs mkdocs-material mkdocstrings[python]
```

```

- name: Build docs
  run: mkdocs build --strict

- name: Deploy docs
  if: github.event_name == 'push' && github.ref ==
'refs/heads/main'
  uses: peaceiris/actions-gh-pages@v3
  with:
    github_token: ${{ secrets.GITHUB_TOKEN }}
    publish_dir: ./site

publish:
  name: Publish Package
  needs: [test, docs]
  runs-on: ubuntu-latest
  if: github.event_name == 'push' && startsWith(github.ref,
'refs/tags')
  steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: "3.10"

    - name: Install build dependencies
      run: |
        python -m pip install --upgrade pip
        pip install build twine

    - name: Build package
      run: python -m build

    - name: Check package
      run: twine check dist/*

    - name: Publish to PyPI
      uses: pypa/gh-action-pypi-publish@release/v1
      with:
        user: __token__
        password: ${{ secrets.PYPI_API_TOKEN }}

    - name: Create GitHub Release
      uses: softprops/action-gh-release@v1
      with:
        files: dist/*
        generate_release_notes: true

```

This comprehensive workflow: 1. Checks code quality (formatting, linting, type checking, security, dead code) 2. Runs tests on multiple Python versions and operating systems 3.

Builds and deploys documentation 4. Publishes packages to PyPI on tagged releases 5. Creates GitHub releases with release notes

3.2.13 CI/CD Best Practices

1. **Keep workflows modular:** Split complex workflows into logical jobs
2. **Fail fast:** Run quick checks (like formatting) before longer ones (like testing)
3. **Cache dependencies:** Speed up workflows by caching pip packages
4. **Be selective:** Only run necessary jobs based on changed files
5. **Test thoroughly:** Include all environments your code supports
6. **Secure secrets:** Use GitHub’s secret storage for tokens and keys
7. **Monitor performance:** Watch workflow execution times and optimise slow steps

With these CI/CD practices in place, your development workflow becomes more reliable and automatic. Quality checks run on every change, documentation stays up to date, and releases happen smoothly and consistently.

AI Tip: CI/CD Workflows

GitHub Actions YAML can be fiddly to get right. Ask AI to generate workflow files: *“Write a GitHub Actions workflow that runs pytest, ruff, and mypy on push to main and on pull requests, for a Python project using uv.”* AI handles the YAML syntax and action versions well, but always test the workflow — AI may reference outdated action versions or miss environment-specific details.

In the final section, we’ll explore how to publish and distribute Python packages to make your work available to others.

3.3 Package Publishing and Distribution

When your Python project matures, you may want to share it with others through the Python Package Index (PyPI). Publishing your package makes it installable via `pip`, allowing others to easily use your work.

3.3.1 Preparing Your Package for Distribution

Before publishing, your project needs the right structure. Let’s ensure everything is ready:

3.3.1.1 1. Package Structure Review

A distributable package should have this basic structure:

```
my_project/
  src/
    my_package/
      __init__.py
      module1.py
      module2.py
  tests/
  docs/
  pyproject.toml
```

```
LICENSE
README.md
```

3.3.1.2 2. Package Configuration with pyproject.toml

Modern Python packaging uses `pyproject.toml` for configuration:

```
[build-system]
requires = ["setuptools>=61.0", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "my-package"
version = "0.1.0"
description = "A short description of my package"
readme = "README.md"
requires-python = ">=3.8"
license = {text = "MIT"}
authors = [
    {name = "Your Name", email = "your.email@example.com"}
]
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]
dependencies = [
    "requests>=2.25.0",
    "numpy>=1.20.0",
]

[project.optional-dependencies]
dev = [
    "pytest>=7.0.0",
    "pytest-cov",
    "ruff",
    "mypy",
]
doc = [
    "mkdocs",
    "mkdocs-material",
    "mkdocstrings[python]",
]

[project.urls]
"Homepage" = "https://github.com/yourusername/my-package"
"Bug Tracker" = "https://github.com/yourusername/my-package/issues"

[project.scripts]
my-command = "my_package.cli:main"
```

```
[tool.setuptools]
package-dir = {"" = "src"}
packages = ["my_package"]
```

This configuration: - Defines basic metadata (name, version, description) - Lists dependencies (both required and optional) - Sets up entry points for command-line scripts - Specifies the package location (src layout)

3.3.1.3 3. Include Essential Files

Ensure you have these files:

```
# Create a LICENSE file (example: MIT License)
cat > LICENSE << EOF
MIT License

Copyright (c) $(date +%Y) Your Name

Permission is hereby granted...
EOF

# Create a comprehensive README.md with:
# - What the package does
# - Installation instructions
# - Basic usage examples
# - Links to documentation
# - Contributing guidelines
```

3.3.2 Building Your Package

With configuration in place, you're ready to build distribution packages:

```
# Install build tools
pip install build

# Build both wheel and source distribution
python -m build
```

This creates two files in the `dist/` directory: - A source distribution (`.tar.gz`) - A wheel file (`.whl`)

Always check your distributions before publishing:

```
# Install twine
pip install twine

# Check the package
twine check dist/*
```

3.3.3 Publishing to Test PyPI

Before publishing to the real PyPI, test your package on TestPyPI:

1. Create a TestPyPI account at <https://test.pypi.org/account/register/>
2. Upload your package:

```
twine upload --repository testpypi dist/*
```

3. Test installation from TestPyPI:

```
pip install --index-url https://test.pypi.org/simple/ --extra-index-url
https://pypi.org/simple/ my-package
```

3.3.4 Publishing to PyPI

When everything works correctly on TestPyPI:

1. Create a PyPI account at <https://pypi.org/account/register/>
2. Upload your package:

```
twine upload dist/*
```

Your package is now available to the world via `pip install my-package!`

3.3.5 Automating Package Publishing

To automate publishing with GitHub Actions, add a workflow that: 1. Builds the package 2. Uploads to PyPI when you create a release tag

```
name: Publish Python Package

on:
  release:
    types: [created]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install build twine
      - name: Build and publish
        env:
          TWINE_USERNAME: ${ secrets.PYPI_USERNAME }
          TWINE_PASSWORD: ${ secrets.PYPI_PASSWORD }
        run: |
          python -m build
          twine upload dist/*
```

For better security, use API tokens instead of your PyPI password: 1. Generate a token

from your PyPI account settings 2. Add it as a GitHub repository secret 3. Use the token in your workflow:

```
- name: Build and publish
  env:
    TWINE_USERNAME: __token__
    TWINE_PASSWORD: ${ secrets.PYPI_API_TOKEN }
  run: |
    python -m build
    twine upload dist/*
```

3.3.6 Versioning Best Practices

Follow Semantic Versioning (MAJOR.MINOR.PATCH): - MAJOR: Incompatible API changes - MINOR: New functionality (backward-compatible) - PATCH: Bug fixes (backward-compatible)

Track versions in one place, usually in `__init__.py`:

```
# src/my_package/__init__.py
__version__ = "0.1.0"
```

Or with a dynamic version from your git tags using `setuptools-scm`:

```
[build-system]
requires = ["setuptools>=61.0", "wheel", "setuptools_scm[toml]>=6.2"]
build-backend = "setuptools.build_meta"

[tool.setuptools_scm]
# Uses git tags for versioning
```

3.3.7 Creating Releases

A good release process includes:

1. **Update documentation:**
 - Ensure README is current
 - Update changelog with notable changes
2. **Create a new version:**
 - Update version number
 - Create a git tag:

```
git tag -a v0.1.0 -m "Release version 0.1.0"
git push origin v0.1.0
```

3. **Monitor the CI/CD pipeline:**
 - Ensure tests pass
 - Verify package build succeeds
 - Confirm successful publication
4. **Announce the release:**
 - Create GitHub release notes
 - Post in relevant community forums
 - Update documentation site

3.3.8 Package Maintenance

Once published, maintain your package responsibly:

1. **Monitor issues** on GitHub or GitLab
2. **Respond to bug reports** promptly
3. **Review and accept contributions** from the community
4. **Regularly update dependencies** to address security issues
5. **Create new releases** when significant improvements are ready

3.3.9 Advanced Distribution Topics

As your package ecosystem grows, consider these advanced techniques:

3.3.9.1 1. Binary Extensions

For performance-critical components, you might include compiled C extensions: - Use Cython to compile Python to C - Configure with the `build-system` section in `pyproject.toml` - Build platform-specific wheels

3.3.9.2 2. Namespace Packages

For large projects split across multiple packages:

```
# src/myorg/packageone/__init__.py
# src/myorg/packagetwo/__init__.py

# Makes 'myorg' a namespace package
```

3.3.9.3 3. Conditional Dependencies

For platform-specific dependencies:

```
dependencies = [
    "requests>=2.25.0",
    "numpy>=1.20.0",
    "pywin32>=300; platform_system == 'Windows'",
]
```

3.3.9.4 4. Data Files

Include non-Python files (data, templates, etc.):

```
[tool.setuptools]
package-dir = {"" = "src"}
packages = ["my_package"]
include-package-data = true
```

Create a `MANIFEST.in` file:

```
include src/my_package/data/*.json
include src/my_package/templates/*.html
```

By following these practices, you'll create a professional, well-maintained package that others can easily discover, install, and use. Publishing your work to PyPI is not just

about sharing code—it's about participating in the Python ecosystem and contributing back to the community.

3.3.10 Modern vs. Traditional Python Packaging

Python packaging has evolved significantly over the years:

3.3.10.1 Traditional `setup.py` Approach

Historically, Python packages required a `setup.py` file:

```
# setup.py
from setuptools import setup, find_packages

setup(
    name="my-package",
    version="0.1.0",
    packages=find_packages(),
    install_requires=[
        "requests>=2.25.0",
        "numpy>=1.20.0",
    ],
)
```

This approach is still common and has advantages for: - Compatibility with older tooling - Dynamic build processes that need Python code - Complex build requirements (e.g., C extensions, custom steps)

3.3.10.2 Modern `pyproject.toml` Approach

Since PEP 517/518, packages can use `pyproject.toml` exclusively:

```
[build-system]
requires = ["setuptools>=61.0", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "my-package"
version = "0.1.0"
dependencies = [
    "requests>=2.25.0",
    "numpy>=1.20.0",
]
```

This declarative approach is recommended for new projects because it: - Provides a standardized configuration format - Supports multiple build systems (not just `setuptools`) - Simplifies dependency specification - Avoids executing Python code during installation

3.3.10.3 Which Approach Should You Use?

- For new, straightforward packages: Use `pyproject.toml` only
- For packages with complex build requirements: You may need both `pyproject.toml` and `setup.py`

- For maintaining existing packages: Consider gradually migrating to `pyproject.toml`

Many projects use a hybrid approach, with basic metadata in `pyproject.toml` and complex build logic in `setup.py`.

Chapter 4

Case Study: Building SimpleBot - A Python Development Workflow Example

This case study demonstrates how to apply the Python development pipeline practices to a real project. We'll walk through the development of SimpleBot, a lightweight wrapper for Large Language Models (LLMs) designed for educational settings.

4.1 Project Overview

SimpleBot is an educational tool that makes it easy for students to interact with Large Language Models through simple Python functions. Key features include:

- Simple API for sending prompts to LLMs
- Pre-defined personality bots (pirate, Shakespeare, emoji, etc.)
- Error handling and user-friendly messages
- Support for local LLM servers like Ollama

This project is ideal for our case study because: - It solves a real problem (making LLMs accessible in educational settings) - It's small enough to understand quickly but complex enough to demonstrate real workflow practices - It includes both pure Python and compiled Cython components

Let's see how we can develop this project using our Python development pipeline. Along the way, we will see how AI can accelerate each phase — from scaffolding the project structure to generating tests and documentation.

Prefer Notebooks?

If you develop primarily in Jupyter notebooks, see Chapter 9 for a parallel case study that builds TextKit using nbdev—same destination (published package),

different workflow.

4.2 1. Setting the Foundation

4.2.1 Project Structure

We'll set up the project using the recommended `src` layout:

```
simplebot/  
  src/  
    simplebot/  
      __init__.py  
      core.py  
      personalities.py  
  tests/  
    __init__.py  
    test_core.py  
    test_personalities.py  
  docs/  
    index.md  
    examples.md  
  .gitignore  
  README.md  
  requirements.in  
  pyproject.toml  
  LICENSE
```

4.2.2 Setting Up Version Control

First, we initialize a Git repository and create a `.gitignore` file:

```
# Initialize Git repository  
git init  
  
# Create a file named README.md with the following contents:  
# Virtual environments  
.venv/  
venv/  
env/  
  
# Python cache files  
__pycache__/  
*.py[cod]  
*$py.class  
.pytest_cache/  
  
# Distribution / packaging  
dist/  
build/  
*.egg-info/
```

```
# Cython generated files
*.c
*.so

# Local development settings
.env
.vscode/

# Coverage reports
htmlcov/
.coverage
EOF

# Initial commit
git add .gitignore
git commit -m "Initial commit: Add .gitignore"
```

4.2.3 Creating Essential Files

Let's create the basic files:

```
# Create the project structure
mkdir -p src/simplebot tests docs

# Create a file name
# SimpleBot

> LLMs made simple for students and educators

SimpleBot is a lightweight Python wrapper that simplifies interactions
with Large Language Models (LLMs) for educational settings.

## Installation

\\\`bash
pip install simplebot
\\\`

## Quick Start

\\\`python
from simplebot import get_response, pirate_bot

# Basic usage
response = get_response("Tell me about planets")
print(response)

# Use a personality bot
pirate_response = pirate_bot("Tell me about sailing ships")
print(pirate_response)
```

```
\\\`

## License

This project is licensed under the MIT License - see the LICENSE file
for details.
EOF

# Create a file named LICENSE with the following contents:
MIT License

Copyright (c) 2025 SimpleBot Authors

Permission is hereby granted, free of charge, to any person obtaining a
copy
of this software and associated documentation files (the "Software"), to
deal
in the Software without restriction, including without limitation the
rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included
in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN THE
SOFTWARE.
EOF

git add README.md LICENSE
git commit -m "Add README and LICENSE"
```

4.2.4 Virtual Environment Setup

We'll create a virtual environment and install basic development packages:

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

```
# Activate the environment (Linux/macOS)
source .venv/bin/activate
# On Windows: .venv\Scripts\activate

# Initial package installation for development
pip install pytest ruff mypy build
```

4.3 2. Building the Core Functionality

Let's start with the core module implementation:

```
# Create the package structure
mkdir -p src/simplebot
```

```
# Create the package __init__.py
# Create a file named src/simplebot/__init__.py with the following
contents:
"""SimpleBot - LLMs made simple for students and educators."""

from .core import get_response
from .personalities import (
    pirate_bot,
    shakespeare_bot,
    emoji_bot,
    teacher_bot,
    coder_bot,
)

__version__ = "0.1.0"

__all__ = [
    "get_response",
    "pirate_bot",
    "shakespeare_bot",
    "teacher_bot",
    "emoji_bot",
    "coder_bot",
]

# Create the core module
# Create a file named src/simplebot/core.py with the following contents:
"""Core functionality for SimpleBot."""

import requests
import random
import time
from typing import Optional, Dict, Any

# Cache for the last used model to avoid redundant loading messages
_last_model: Optional[str] = None
```

```

def get_response(
    prompt: str,
    model: str = "llama3",
    system: str = "You are a helpful assistant.",
    stream: bool = False,
    api_url: Optional[str] = None,
) -> str:
    """
    Send a prompt to the LLM API and retrieve the model's response.

    Args:
        prompt: The text prompt to send to the language model
        model: The name of the model to use
        system: System instructions that control the model's behaviour
        stream: Whether to stream the response
        api_url: Custom API URL (defaults to local Ollama server)

    Returns:
        The model's response text, or an error message if the request
    fails
    """
    global _last_model

    # Default to local Ollama if no API URL is provided
    if api_url is None:
        api_url = "http://localhost:11434/api/generate"

    # Handle model switching with friendly messages
    if model != _last_model:
        warmup_messages = [
            f" Loading model '{model}' into RAM... give me a sec...",
            f" Spinning up the AI core for '{model}'...",
            f" Summoning the knowledge spirits... '{model}'
            booting...",
            f" Thinking really hard with '{model}'...",
            f" Switching to model: {model} ... (may take a few
            seconds)",
        ]
        print(random.choice(warmup_messages))

    # Short pause to simulate/allow for model loading
    time.sleep(1.5)
    _last_model = model

    # Validate input
    if not prompt.strip():
        return " Empty prompt."

    # Prepare the request payload
    payload: Dict[str, Any] = {
        "model": model,

```

```

        "prompt": prompt,
        "system": system,
        "stream": stream
    }

    try:
        # Send request to the LLM API
        response = requests.post(
            api_url,
            json=payload,
            timeout=10
        )
        response.raise_for_status()
        data = response.json()
        return data.get("response", " No response from model.")
    except requests.RequestException as e:
        return f" Connection Error: {str(e)}"
    except Exception as e:
        return f" Error: {str(e)}"
EOF

# Create the personalities module
# Create a file named src/simplebot/personalities.py with the following
contents:
"""Pre-defined personality bots for SimpleBot."""

from .core import get_response
from typing import Optional

def pirate_bot(prompt: str, model: Optional[str] = None) -> str:
    """
    Generate a response in the style of a 1700s pirate with nautical
    slang.

    Args:
        prompt: The user's input text/question
        model: Optional model override

    Returns:
        A response written in pirate vernacular
    """
    return get_response(
        prompt,
        system="You are a witty pirate from the 1700s. "
            "Use nautical slang, say 'arr' occasionally, "
            "and reference sailing, treasure, and the sea.",
        model=model or "llama3"
    )

def shakespeare_bot(prompt: str, model: Optional[str] = None) -> str:
    """

```

```

Generate a response in the style of William Shakespeare.

Args:
    prompt: The user's input text/question
    model: Optional model override

Returns:
    A response written in Shakespearean style
"""
return get_response(
    prompt,
    system="You respond in the style of William Shakespeare, "
          "using Early Modern English vocabulary and phrasing.",
    model=model or "llama3"
)

def emoji_bot(prompt: str, model: Optional[str] = None) -> str:
    """
    Generate a response primarily using emojis with minimal text.

    Args:
        prompt: The user's input text/question
        model: Optional model override

    Returns:
        A response composed primarily of emojis
    """
    return get_response(
        prompt,
        system="You respond using mostly emojis, mixing minimal words "
              "and symbols to convey meaning. You love using expressive "
              "
              "emoji strings.",
        model=model or "llama3"
    )

def teacher_bot(prompt: str, model: Optional[str] = None) -> str:
    """
    Generate a response in the style of a patient, helpful educator.

    Args:
        prompt: The user's input text/question
        model: Optional model override

    Returns:
        A response with an educational approach
    """
    return get_response(
        prompt,
        system="You are a patient, encouraging teacher who explains "
              "concepts clearly at an appropriate level. Break down "

```

```

        "complex ideas into simpler components and use analogies
        "
        "when helpful.",
        model=model or "llama3"
    )

def coder_bot(prompt: str, model: Optional[str] = None) -> str:
    """
    Generate a response from a coding assistant optimised for
    programming help.

    Args:
        prompt: The user's input programming question or request
        model: Optional model override (defaults to a coding-specific
model)

    Returns:
        A technical response focused on code-related assistance
    """
    return get_response(
        prompt,
        system="You are a skilled coding assistant who explains and
writes "
            "code clearly and concisely. Prioritize best practices, "
            "readability, and proper error handling.",
        model=model or "codellama"
    )
EOF

git add src/
git commit -m "Add core SimpleBot functionality"

```

4.4 3. Package Configuration

Let's set up the package configuration in `pyproject.toml`:

```
# Create pyproject.toml directory
```

i Modern Packaging

This case study uses the newer `pyproject.toml`-only approach for simplicity and to follow current best practices. Many existing Python projects still use `setup.py`, either alongside `pyproject.toml` or as their primary configuration. The `setup.py` approach remains valuable for packages with complex build requirements, custom build steps, or when supporting older tools and Python versions. For SimpleBot, our straightforward package requirements allow us to use the cleaner, declarative `pyproject.toml` approach.

4.5 Create a file named `pyproject.toml` with the following contents:

Let's set up the package configuration in `pyproject.toml`:

```
# Create a file named pyproject.toml with the following contents:
[build-system]
requires = ["setuptools>=61.0", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "simplebot"
version = "0.1.0"
description = "LLMs made simple for students and educators"
readme = "README.md"
requires-python = ">=3.7"
license = {text = "MIT"}
authors = [
    {name = "SimpleBot Team", email = "example@example.com"}
]
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
    "Intended Audience :: Education",
    "Topic :: Education :: Computer Aided Instruction (CAI)",
]
dependencies = [
    "requests>=2.25.0",
]

[project.optional-dependencies]
dev = [
    "pytest>=7.0.0",
    "pytest-cov",
    "ruff",
    "mypy",
]

[project.urls]
"Homepage" = "https://github.com/simplebot-team/simplebot"
"Bug Tracker" = "https://github.com/simplebot-team/simplebot/issues"

# Tool configurations
[tool.ruff]
select = ["E", "F", "I"]
line-length = 88

[tool.ruff.per-file-ignores]
"__init__.py" = ["F401"]

[tool.mypy]
```

```

python_version = "3.7"
warn_return_any = true
warn_unused_configs = true
disallow_untyped_defs = true
disallow_incomplete_defs = true

[[tool.mypy.overrides]]
module = "tests.*"
disallow_untyped_defs = false

[tool.pytest.ini_options]
testpaths = ["tests"]
EOF

# Create requirements.in file
# Create a file named requirements.in with the following contents:
# Direct dependencies
requests>=2.25.0
EOF

# Create requirements-dev.in
# Create a file named requirements-dev.in with the following contents:
# Development dependencies
pytest>=7.0.0
pytest-cov
ruff
mypy
build
twine
EOF

git add pyproject.toml requirements*.in
git commit -m "Add package configuration and dependency files"

```

4.6 4. Writing Tests

Let's create some tests for our SimpleBot functionality:

```

# Create test files
# Create a file named tests/__init__.py with the following contents:
"""SimpleBot test package."""
EOF

# Create a file named tests/test_core.py with the following contents:
"""Tests for the SimpleBot core module."""

import pytest
from unittest.mock import patch, MagicMock
from simplebot.core import get_response

@patch("simplebot.core.requests.post")

```

```

def test_successful_response(mock_post):
    """Test that a successful API response is handled correctly."""
    # Setup mock
    mock_response = MagicMock()
    mock_response.json.return_value = {"response": "Test response"}
    mock_post.return_value = mock_response

    # Call function
    result = get_response("Test prompt")

    # Assertions
    assert result == "Test response"
    mock_post.assert_called_once()

@patch("simplebot.core.requests.post")
def test_empty_prompt(mock_post):
    """Test that empty prompts are handled correctly."""
    result = get_response("")
    assert "Empty prompt" in result
    mock_post.assert_not_called()

@patch("simplebot.core.requests.post")
def test_api_error(mock_post):
    """Test that API errors are handled gracefully."""
    # Setup mock to raise an exception
    mock_post.side_effect = Exception("Test error")

    # Call function
    result = get_response("Test prompt")

    # Assertions
    assert "Error" in result
    assert "Test error" in result
EOF

# Create a file named tests/test_personalities.py with the following
contents:
"""Tests for the SimpleBot personalities module."""

import pytest
from unittest.mock import patch
from simplebot import (
    pirate_bot,
    shakespeare_bot,
    emoji_bot,
    teacher_bot,
    coder_bot,
)

@patch("simplebot.personalities.get_response")
def test_pirate_bot(mock_get_response):

```

```

    """Test that pirate_bot calls get_response with correct
    parameters."""
    # Setup
    mock_get_response.return_value = "Arr, test response!"

    # Call function
    result = pirate_bot("Test prompt")

    # Assertions
    assert result == "Arr, test response!"
    mock_get_response.assert_called_once()
    # Check that system prompt contains pirate references
    system_arg = mock_get_response.call_args[1]["system"]
    assert "pirate" in system_arg.lower()

@patch("simplebot.personalities.get_response")
def test_custom_model(mock_get_response):
    """Test that personality bots accept custom model parameter."""
    # Setup
    mock_get_response.return_value = "Custom model response"

    # Call functions with custom model
    shakespeare_bot("Test", model="custom-model")

    # Assertions
    assert mock_get_response.call_args[1]["model"] == "custom-model"
EOF

git add tests/
git commit -m "Add unit tests for SimpleBot"

```

4.7 5. Applying Code Quality Tools

Let's run our code quality tools and fix any issues:

```

# Install development dependencies
pip install -r requirements-dev.in

# Run Ruff for formatting and linting
ruff format .
ruff check .

# Run mypy for type checking
mypy src/

# Fix any issues identified by the tools
git add .
git commit -m "Apply code formatting and fix linting issues"

```

4.8 6. Documentation

Let's create basic documentation:

```
# Create docs directory
mkdir -p docs

# Create main documentation file
# Create a file named docs/index.md with the following contents:
# SimpleBot Documentation

> LLMs made simple for students and educators

SimpleBot is a lightweight Python wrapper that simplifies interactions
with Large Language Models (LLMs) for educational settings. It abstracts
away the complexity of API calls, model management, and error handling,
allowing students to focus on learning programming concepts through
engaging AI interactions.

## Installation

\\\`bash
pip install simplebot
\\\`

## Basic Usage

\\\`python
from simplebot import get_response

# Basic usage with default model
response = get_response("Tell me about planets")
print(response)
\\\`

## Personality Bots

SimpleBot comes with several pre-defined personality bots:

\\\`python
from simplebot import pirate_bot, shakespeare_bot, emoji_bot,
teacher_bot, coder_bot

# Get a response in pirate speak
pirate_response = pirate_bot("Tell me about sailing ships")
print(pirate_response)

# Get a response in Shakespearean style
shakespeare_response = shakespeare_bot("What is love?")
print(shakespeare_response)

# Get a response with emojis
```

```

emoji_response = emoji_bot("Explain happiness")
print(emoji_response)

# Get an educational response
teacher_response = teacher_bot("How do photosynthesis work?")
print(teacher_response)

# Get coding help
code_response = coder_bot("Write a Python function to check if a string
is a palindrome")
print(code_response)
```


```

```
API Reference
```

```
get_response()
```

```
```python
def get_response(
    prompt: str,
    model: str = "llama3",
    system: str = "You are a helpful assistant.",
    stream: bool = False,
    api_url: Optional[str] = None,
) -> str:
```

```

The core function for sending prompts to an LLM and getting responses.

```
Parameters:
```

- `prompt`: The text prompt to send to the language model
- `model`: The name of the model to use (default: "llama3")
- `system`: System instructions that control the model's behaviour
- `stream`: Whether to stream the response (default: False)
- `api\_url`: Custom API URL (defaults to local Ollama server)

```
Returns:
```

- A string containing the model's response or an error message
- EOF

```

Create examples file
Create a file named docs/examples.md with the following contents:
SimpleBot Examples

```

Here are some examples of using SimpleBot in educational settings.

```
Creating Custom Bot Personalities
```

You can create custom bot personalities:

```

\\\`python
from simplebot import get_response

def scientist_bot(prompt):
 """A bot that responds like a scientific researcher."""
 return get_response(
 prompt,
 system="You are a scientific researcher. Provide evidence-based
 "
 "responses with references to studies when possible. "
 "Be precise and methodical in your explanations."
)

result = scientist_bot("What happens during photosynthesis?")
print(result)
\\\`

Building a Simple Quiz System

\\\`python
from simplebot import teacher_bot

quiz_questions = [
 "What is the capital of France?",
 "Who wrote Romeo and Juliet?",
 "What is the chemical symbol for water?"
]

def generate_quiz():
 print("=== Quiz Time! ===")
 for i, question in enumerate(quiz_questions, 1):
 print(f"Question {i}: {question}")
 user_answer = input("Your answer: ")

 # Generate feedback on the answer
 feedback = teacher_bot(
 f"Question: {question}\nStudent answer: {user_answer}\n"
 "Provide brief, encouraging feedback on whether this answer "
 "is "
 "correct. If incorrect, provide the correct answer."
)
 print(f"Feedback: {feedback}\n")

Run the quiz
generate_quiz()
\\\`

Simulating a Conversation Between Bots

\\\`python

```

```

from simplebot import pirate_bot, shakespeare_bot

def bot_conversation(topic, turns=3):
 """Simulate a conversation between two bots on a given topic."""
 print(f"=== A conversation about {topic} ===")

 # Start with the pirate
 current_message = f"Tell me about {topic}"
 current_bot = "pirate"

 for i in range(turns):
 if current_bot == "pirate":
 response = pirate_bot(current_message)
 print(f" Pirate: {response}")
 current_message = f"Respond to this: {response}"
 current_bot = "shakespeare"
 else:
 response = shakespeare_bot(current_message)
 print(f" Shakespeare: {response}")
 current_message = f"Respond to this: {response}"
 current_bot = "pirate"
 print()

Run a conversation about the ocean
bot_conversation("the ocean", turns=4)
\\`\\`\\`
EOF

git add docs/
git commit -m "Add documentation"

```

## 4.9 7. Setup CI/CD with GitHub Actions

Now let's set up continuous integration:

```

Create GitHub Actions workflow directory
mkdir -p .github/workflows

Create CI workflow file
Create a file named .github/workflows/ci.yml with the following
contents:
name: Python CI

on:
 push:
 branches: [main]
 pull_request:
 branches: [main]

jobs:
 test:

```

```

runs-on: ubuntu-latest
strategy:
 matrix:
 python-version: ["3.7", "3.8", "3.9", "3.10"]

steps:
- uses: actions/checkout@v3

- name: Set up Python \${{ matrix.python-version }}
 uses: actions/setup-python@v4
 with:
 python-version: \${{ matrix.python-version }}
 cache: pip

- name: Install dependencies
 run: |
 python -m pip install --upgrade pip
 python -m pip install -e ".[dev]"

- name: Check formatting with Ruff
 run: ruff format --check .

- name: Lint with Ruff
 run: ruff check .

- name: Type check with mypy
 run: mypy src/

- name: Test with pytest
 run: pytest --cov=src/ tests/

- name: Build package
 run: python -m build
EOF

Create release workflow
Create a file named .github/workflows/release.yml with the following
contents:
name: Publish to PyPI

on:
 release:
 types: [created]

jobs:
 deploy:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3

 - name: Set up Python

```

```

uses: actions/setup-python@v4
with:
 python-version: "3.10"

- name: Install dependencies
 run: |
 python -m pip install --upgrade pip
 pip install build twine

- name: Build and publish
 env:
 TWINE_USERNAME: \${{ secrets.PYPI_USERNAME }}
 TWINE_PASSWORD: \${{ secrets.PYPI_PASSWORD }}
 run: |
 python -m build
 twine check dist/*
 twine upload dist/*
EOF

git add .github/
git commit -m "Add CI/CD workflows"

```

## 4.10 8. Finalizing for Distribution

Let's prepare for distribution:

```

Install the package in development mode
pip install -e .

Run the tests
pytest

Build the package
python -m build

Verify the package
twine check dist/*

```

## 4.11 9. Project Summary

By following the Python Development Workflow, we've transformed the SimpleBot concept into a well-structured, tested, and documented Python package that's ready for distribution. Let's review what we've accomplished:

1. **Project Foundation:**
  - Created a clear, organised directory structure
  - Set up version control with Git
  - Added essential files (README, LICENSE)
2. **Development Environment:**
  - Created a virtual environment

- Managed dependencies cleanly
- 3. **Code Quality:**
  - Applied type hints throughout the codebase
  - Used Ruff for formatting and linting
  - Used mypy for static type checking
- 4. **Testing:**
  - Created comprehensive unit tests with pytest
  - Used mocking to test external API interactions
- 5. **Documentation:**
  - Added clear docstrings
  - Created usage documentation with examples
- 6. **Packaging & Distribution:**
  - Configured the package with pyproject.toml
  - Set up CI/CD with GitHub Actions

## 4.12 10. Next Steps

If we were to continue developing SimpleBot, potential next steps might include:

1. **Enhanced Features:**
  - Add more personality bots
  - Support for conversation memory/context
  - Configuration file support
2. **Advanced Documentation:**
  - Set up MkDocs for a full documentation site
  - Add tutorials for classroom usage
3. **Performance Improvements:**
  - Add caching for responses
  - Implement Cython optimisation for performance-critical sections
4. **Security Enhancements:**
  - Add API key management
  - Implement content filtering for educational settings

### AI Tip: Using AI Throughout the Pipeline

Notice how AI could have accelerated every phase of this case study. You could ask AI to generate the initial project structure, draft the core module, write pytest tests, create the pyproject.toml configuration, generate GitHub Actions workflows, and draft the README. The pipeline you learned in earlier chapters gives you the knowledge to evaluate everything AI produces. Without that knowledge, you would accept AI output uncritically. With it, you orchestrate AI as a force multiplier.

This case study demonstrates how following a structured Python development workflow leads to a high-quality, maintainable, and distributable package — even for relatively small projects.

# Chapter 5

## Advanced Development Techniques

As your Python projects grow in complexity and requirements, you'll encounter challenges that require more sophisticated approaches than the foundational practices we've established. This chapter explores advanced techniques that build upon our core development pipeline, focusing on principles and patterns that scale with your project's needs.

Rather than diving into the specifics of every advanced tool, we'll focus on understanding **when and why** to adopt more complex solutions, maintaining our philosophy of "simple but not simplistic."

### 5.1 Performance optimisation: Measure First, optimise Second

Performance optimisation often feels compelling, but premature optimisation is a common trap. The key principle: **measure before you optimise**. Our development pipeline already includes the foundation for performance work through comprehensive testing and quality gates.

#### 5.1.1 Establishing Performance Baselines

Before optimising, establish measurable baselines using tools that integrate naturally with your existing workflow:

```
performance/benchmarks.py
import time
import pytest
from my_package.core import expensive_function

class TestPerformance:
 """Performance benchmarks for critical functions."""

 def test_expensive_function_performance(self, benchmark):
```

```

 """Benchmark the expensive function execution time."""
 # pytest-benchmark integrates with our existing test suite
 result = benchmark(expensive_function, large_dataset)
 assert result is not None # Basic correctness check

@pytest.mark.slow
def test_memory_usage_under_load(self):
 """Test memory behaviour with large datasets."""
 import psutil
 import os

 process = psutil.Process(os.getpid())
 initial_memory = process.memory_info().rss

 # Run memory-intensive operation
 result = process_large_dataset()

 final_memory = process.memory_info().rss
 memory_increase = final_memory - initial_memory

 # Assert reasonable memory usage (adjust threshold as needed)
 assert memory_increase < 100 * 1024 * 1024 # 100MB threshold

```

Add performance dependencies to your development requirements:

```

[tool.poe.tasks]
Add performance testing to your task automation
benchmark = "pytest --benchmark-only performance/"
profile = "python -m cProfile -o profile.stats src/my_package/main.py"
profile-view = "python -c 'import pstats; pstats.Stats(\"profile.stats\
\").sort_stats(\"cumulative\").print_stats(20)'"

```

This approach integrates performance measurement into your existing development workflow rather than introducing entirely new tools.

## 5.1.2 Performance optimisation Strategy

When benchmarks indicate performance issues, follow a systematic approach:

1. **Profile to identify bottlenecks** - Don't guess where the slowness is
2. **optimise the algorithms first** - Better algorithms beat micro-optimisations
3. **Consider caching strategically** - Cache expensive computations, not everything
4. **Measure the impact** - Ensure optimisations actually improve performance

```

Example: Adding strategic caching to expensive operations
from functools import lru_cache
from typing import Dict, Any

class DataProcessor:
 """Example of strategic performance optimisation."""

 @lru_cache(maxsize=128)

```

```

def expensive_calculation(self, key: str) -> Dict[str, Any]:
 """Cache expensive calculations with bounded memory usage."""
 # Expensive computation here
 return self._compute_complex_result(key)

def process_batch(self, items: list) -> list:
 """Process items in batches to reduce overhead."""
 # Batch processing reduces per-item overhead
 batch_size = 100
 results = []

 for i in range(0, len(items), batch_size):
 batch = items[i:i + batch_size]
 batch_results = self._process_batch_optimized(batch)
 results.extend(batch_results)

 return results

```

The key insight: **optimise within your existing architecture** before considering more complex solutions like Cython or asyncio.

## 5.2 Containerization: Development Environment Consistency

Containers address the challenge of environment reproducibility across different development machines and deployment environments. However, containerization should enhance, not replace, your existing development workflow.

### 5.2.1 Development Containers vs. Production Containers

**Development containers** prioritize developer experience: - Fast rebuild times - Volume mounts for live code editing - Development tools and debugging capabilities - Integration with your existing toolchain

**Production containers** prioritize runtime efficiency: - Minimal attack surface - optimised for size and startup time - No development dependencies - Security-focused configurations

### 5.2.2 Integrating Containers with Your Workflow

Create a Dockerfile that builds upon your existing dependency management:

```

Dockerfile - Multi-stage build supporting both development and
production
FROM python:3.11-slim as base

Install uv for fast dependency management
RUN pip install uv

WORKDIR /app

```

```
Copy dependency specifications
COPY pyproject.toml uv.lock ./

Development stage
FROM base as development
RUN uv sync --all-extras --dev
COPY . .
CMD ["uv", "run", "python", "-m", "my_package"]

Production stage
FROM base as production
RUN uv sync --frozen --no-dev
COPY src/ src/
RUN uv pip install -e .
CMD ["python", "-m", "my_package"]
```

Add container management to your task automation:

```
[tool.poe.tasks]
Development container tasks
docker-build = "docker build --target development -t my-project:dev ."
docker-run = "docker run -it --rm -v $(pwd):/app my-project:dev"
docker-test = "docker run --rm -v $(pwd):/app my-project:dev uv run
pytest"

Production container tasks
docker-build-prod = "docker build --target production -t my-project:prod
."
```

This approach uses containers to **enhance reproducibility** without disrupting your core development workflow.

### 5.2.3 When to Containerize

Consider containerization when you encounter: - **Environment inconsistencies** between team members - **Complex system dependencies** that are difficult to install - **Deployment environment differences** from development - **Service integration challenges** (databases, message queues, etc.)

Don't containerize simply because it's trendy — use it to solve specific reproducibility problems.

#### AI Tip: Dockerfile Generation

Dockerfiles are perfect candidates for AI generation. Try: *“Write a multi-stage Dockerfile for a Python FastAPI application using uv for dependency management. The final image should be minimal.”* AI handles the syntax and layer ordering well. But always review the base images it suggests (are they up to date? are they the slim variants?) and test the build — AI often gets volume mounts and port mappings wrong in docker-compose files.

## 5.3 Scaling Your Development Process

As projects grow, you'll need techniques for managing complexity while maintaining development velocity.

### 5.3.1 Modular Architecture Patterns

Design your codebase for growth by establishing clear module boundaries:

```
src/my_package/core/interfaces.py
from abc import ABC, abstractmethod
from typing import Any, Dict

class DataProcessor(ABC):
 """Interface for data processing implementations."""

 @abstractmethod
 def process(self, data: Dict[str, Any]) -> Dict[str, Any]:
 """Process data according to implementation-specific logic."""
 pass

class StorageBackend(ABC):
 """Interface for storage implementations."""

 @abstractmethod
 def save(self, key: str, data: Dict[str, Any]) -> bool:
 """Save data to storage backend."""
 pass

 @abstractmethod
 def load(self, key: str) -> Dict[str, Any]:
 """Load data from storage backend."""
 pass
```

This interface-based design allows you to: 1. **Test implementations independently** with mocks and stubs 2. **Swap implementations** without changing dependent code 3. **Add new implementations** without modifying existing code 4. **Maintain clear boundaries** between different parts of your system

### 5.3.2 Configuration Management

As projects grow, configuration becomes more complex. Establish patterns early:

```
src/my_package/config.py
from dataclasses import dataclass
from pathlib import Path
from typing import Optional
import os

@dataclass
class DatabaseConfig:
 """Database connection configuration."""
 host: str
```

```

port: int
username: str
password: str
database: str

@classmethod
def from_env(cls) -> 'DatabaseConfig':
 """Create config from environment variables."""
 return cls(
 host=os.getenv('DB_HOST', 'localhost'),
 port=int(os.getenv('DB_PORT', '5432')),
 username=os.getenv('DB_USERNAME', ''),
 password=os.getenv('DB_PASSWORD', ''),
 database=os.getenv('DB_NAME', ''),
)

@dataclass
class AppConfig:
 """Main application configuration."""
 debug: bool
 database: DatabaseConfig
 log_level: str

 @classmethod
 def load(cls, config_path: Optional[Path] = None) -> 'AppConfig':
 """Load configuration from environment and optional config
 file."""
 # Implementation handles environment variables,
 # config files, and sensible defaults
 pass

```

This approach provides: - **Type safety** through dataclasses and type hints - **Environment-based configuration** for different deployment contexts - **Testable configuration** through dependency injection - **Clear documentation** of required configuration values

### 5.3.3 Database Integration Patterns

When your application needs persistent storage, integrate database operations cleanly with your existing testing and development workflow:

```

src/my_package/database.py
from contextlib import contextmanager
from typing import Generator
import sqlalchemy as sa
from sqlalchemy.orm import sessionmaker

class DatabaseManager:
 """Manages database connections and sessions."""

 def __init__(self, connection_string: str):
 self.engine = sa.create_engine(connection_string)

```

```

 self.SessionLocal = sessionmaker(bind=self.engine)

 @contextmanager
 def get_session(self) -> Generator[sa.orm.Session, None, None]:
 """Get a database session with automatic cleanup."""
 session = self.SessionLocal()
 try:
 yield session
 session.commit()
 except Exception:
 session.rollback()
 raise
 finally:
 session.close()

Integration with your application
class UserService:
 """Service for user-related operations."""

 def __init__(self, db_manager: DatabaseManager):
 self.db_manager = db_manager

 def create_user(self, email: str, name: str) -> User:
 """Create a new user."""
 with self.db_manager.get_session() as session:
 user = User(email=email, name=name)
 session.add(user)
 session.flush() # Get the ID without committing
 return user

```

Test database operations with fixtures:

```

tests/conftest.py
import pytest
from my_package.database import DatabaseManager

@pytest.fixture
def db_manager():
 """Provide a test database manager."""
 # Use in-memory SQLite for tests
 manager = DatabaseManager("sqlite:///memory:")
 # Create tables
 Base.metadata.create_all(manager.engine)
 return manager

@pytest.fixture
def user_service(db_manager):
 """Provide a user service with test database."""
 return UserService(db_manager)

```

This pattern maintains clean separation between business logic and data persistence while integrating smoothly with your testing infrastructure.

## 5.4 API Development and Integration

When building applications that expose or consume APIs, maintain the same development quality principles.

### 5.4.1 API Design Principles

Design APIs that are: 1. **Consistent** - Similar operations work similarly 2. **Documented** - Clear, up-to-date documentation 3. **Versioned** - Handle changes without breaking existing clients 4. **Testable** - Easy to test both as provider and consumer

```
src/my_package/api/schemas.py
from pydantic import BaseModel, Field
from typing import List, Optional
from datetime import datetime

class UserCreate(BaseModel):
 """Schema for creating a new user."""
 email: str = Field(..., description="User's email address")
 name: str = Field(..., min_length=1, description="User's full name")

class User(BaseModel):
 """Schema for user data."""
 id: int
 email: str
 name: str
 created_at: datetime

 class Config:
 from_attributes = True # For SQLAlchemy integration

class UserList(BaseModel):
 """Schema for user list responses."""
 users: List[User]
 total: int
 page: int
 per_page: int
```

### 5.4.2 API Testing Strategy

Test APIs at multiple levels:

```
tests/test_api.py
import pytest
from fastapi.testclient import TestClient
from my_package.api.main import app

@pytest.fixture
def client():
 """API test client."""
 return TestClient(app)
```

```

def test_create_user_success(client, db_manager):
 """Test successful user creation."""
 user_data = {
 "email": "test@example.com",
 "name": "Test User"
 }

 response = client.post("/users/", json=user_data)

 assert response.status_code == 201
 assert response.json()["email"] == user_data["email"]
 assert "id" in response.json()

def test_create_user_validation_error(client):
 """Test user creation with invalid data."""
 invalid_data = {
 "email": "not-an-email",
 "name": "" # Empty name should fail validation
 }

 response = client.post("/users/", json=invalid_data)

 assert response.status_code == 422
 assert "detail" in response.json()

```

This approach integrates API testing with your existing pytest infrastructure and maintains the same quality standards.

## 5.5 Cross-Platform Development Considerations

When your Python application needs to run across different operating systems, handle platform differences gracefully within your existing development workflow.

### 5.5.1 Path and Environment Handling

Use `pathlib` and environment-aware patterns:

```

src/my_package/utils/paths.py
from pathlib import Path
import os
import sys
from typing import Optional

class PathManager:
 """Handle cross-platform path operations."""

 @staticmethod
 def get_config_dir() -> Path:
 """Get the platform-appropriate configuration directory."""
 if sys.platform == "win32":
 config_dir = Path(os.getenv('APPDATA', '')) / 'my_package'

```

```

 elif sys.platform == "darwin": # macOS
 config_dir = Path.home() / 'Library' / 'Application Support'
/ 'my_package'
 else: # Linux and other Unix-like systems
 config_dir = Path(os.getenv('XDG_CONFIG_HOME', Path.home() /
'.config')) / 'my_package'

 config_dir.mkdir(parents=True, exist_ok=True)
 return config_dir

 @staticmethod
 def get_data_dir() -> Path:
 """Get the platform-appropriate data directory."""
 if sys.platform == "win32":
 data_dir = Path(os.getenv('LOCALAPPDATA', '')) /
'my_package'
 elif sys.platform == "darwin":
 data_dir = Path.home() / 'Library' / 'Application Support' /
'my_package'
 else:
 data_dir = Path(os.getenv('XDG_DATA_HOME', Path.home() /
'.local' / 'share')) / 'my_package'

 data_dir.mkdir(parents=True, exist_ok=True)
 return data_dir

```

## 5.5.2 Testing Across Platforms

Use your existing CI/CD pipeline to test across platforms:

```

.github/workflows/test.yml - Platform matrix testing
name: Tests
on: [push, pull_request]

jobs:
 test:
 runs-on: ${ matrix.os }
 strategy:
 matrix:
 os: [ubuntu-latest, windows-latest, macos-latest]
 python-version: [3.9, 3.10, 3.11]

 steps:
 - uses: actions/checkout@v4
 - name: Set up Python
 uses: actions/setup-python@v4
 with:
 python-version: ${ matrix.python-version }

 - name: Install uv
 run: pip install uv

```

```
- name: Install dependencies
 run: uv sync

- name: Run tests
 run: uv run pytest
```

This extends your existing quality gates to ensure cross-platform compatibility.

## 5.6 When to Adopt Advanced Techniques

The key to advanced techniques is **selective adoption based on actual needs**:

### 5.6.1 Adopt Containerization When:

- Team members struggle with environment setup
- You need to integrate with external services during development
- Deployment environments differ significantly from development

### 5.6.2 Adopt Performance optimisation When:

- Benchmarks show actual performance problems
- Performance requirements are clearly defined
- You have established baseline measurements

### 5.6.3 Adopt Advanced Architecture When:

- Code complexity makes maintenance difficult
- You need to support multiple implementations of core functionality
- Team size makes modular development beneficial

### 5.6.4 Don't Adopt Advanced Techniques When:

- Your current approach works well
- The complexity cost exceeds the benefits
- You haven't mastered the foundational practices

## 5.7 Maintaining Development Velocity

The most important principle for advanced techniques: **they should enhance, not replace, your core development practices**. Your testing, code quality, documentation, and automation should continue to work as you adopt more sophisticated approaches.

Advanced techniques are tools for solving specific problems, not goals in themselves. Focus on delivering value through your software while maintaining the solid development foundation you've established.

 AI Tip: Architecture Decisions

When deciding whether to adopt an advanced technique, describe your situation to your AI assistant: *“My Flask app handles 50 requests per second and response times are around 200ms. Users are complaining about slow page loads. Should I add caching, switch to async, or look at the database queries first?”* AI can help you reason through trade-offs, but the decision is yours — AI does not know your team’s capacity or your deployment constraints.

# Chapter 6

## Project Management and Automation

Moving beyond individual development practices, this chapter focuses on project-level management, automation, and collaboration workflows. We'll explore tools and techniques that help you maintain consistency, automate repetitive tasks, and establish sustainable development practices.

### 6.1 Task Automation with Poe the Poet

One of the first challenges in any Python project is managing the growing number of commands you need to run: testing, linting, formatting, building documentation, and more. While traditional Unix environments might use Makefiles, Python projects benefit from a more integrated approach.

**Poe the Poet** provides a powerful task runner that integrates seamlessly with your `pyproject.toml` file, offering a cross-platform alternative to Makefiles that works naturally with your existing Python toolchain.

#### 6.1.1 Setting Up Poe the Poet

Add Poe the Poet as a development dependency to your project:

```
uv add --dev poethepoet
```

This aligns with our philosophy of keeping project tooling within the project itself, ensuring every developer has access to the same automation tools.

#### 6.1.2 Defining Project Tasks

Define your common development tasks in your `pyproject.toml` file:

```
[tool.poe.tasks]
Code quality tasks
lint = "ruff check ."
format = "ruff format ."
```

```

type-check = "mypy src/"

Testing tasks
test = "pytest tests/"
test-cov = "pytest --cov=src tests/"

Project management
clean = { shell = "rm -rf dist/ .coverage htmlcov/ .pytest_cache/" }
install-dev = { shell = "uv sync && pre-commit install" }

Documentation
docs-serve = "mkdocs serve"
docs-build = "mkdocs build"

Combined workflows
check = ["format", "lint", "type-check", "test"]
build = ["clean", "check", "uv build"]

```

This configuration demonstrates several key principles:

1. **Single source of truth:** All project automation is defined in one place
2. **Composable tasks:** Complex workflows are built from simpler tasks
3. **Cross-platform compatibility:** Tasks work on Windows, macOS, and Linux
4. **Integration with existing tools:** Works seamlessly with uv, ruff, pytest, and other tools in our stack

### 6.1.3 Advanced Task Configuration

For more complex scenarios, Poe supports parameterized tasks and conditional execution:

```

[tool.poe.tasks]
Task with parameters
test-file = { cmd = "pytest ${file}", args = [
 { name = "file", default = "tests/", help = "Test file or
 directory" }
]}

Multi-step setup task
setup = { shell = """
 uv sync
 pre-commit install
 echo "Development environment ready!"
 """ }

Environment-specific tasks
[tool.poe.tasks.deploy]
shell = """
if ["$ENVIRONMENT" = "production"]; then
 echo "Deploying to production..."
 # Add production deployment commands
else
 echo "Deploying to staging..."

```

```
Add staging deployment commands
fi
"""
```

### 6.1.4 Running Tasks

Execute your defined tasks using the `poe` command through `uv`:

```
Run individual tasks
uv run poe lint
uv run poe test

Run parameterized tasks
uv run poe test-file tests/test_specific.py

Chain multiple tasks
uv run poe format lint test

Run complex workflows
uv run poe check # Runs format, lint, type-check, test in sequence
uv run poe build # Full build pipeline
```

### 6.1.5 Integration with Development Workflow

The power of Poe the Poet becomes apparent when integrated into your daily development routine:

**Pre-commit hooks** can reference your Poe tasks:

```
.pre-commit-config.yaml
repos:
 - repo: local
 hooks:
 - id: poe-check
 name: Run project checks
 entry: uv run poe check
 language: system
 pass_filenames: false
```

**IDE integration** allows running tasks directly from your editor, while **CI/CD pipelines** can use the same task definitions:

```
GitHub Actions example
- name: Run checks
 run: uv run poe check
```

This approach eliminates the disconnect between local development and automated systems — everyone uses the same commands.

### 💡 AI Tip: Task Automation

AI can generate Poe the Poet task definitions, Makefiles, and shell scripts from natural language descriptions. Try: “*Write Poe the Poet tasks for: formatting with ruff, linting, running tests with coverage, and a ‘check’ task that runs all three in sequence.*” Since these are configuration files with well-defined syntax, AI output is usually accurate. But test each task before committing it — path assumptions and tool flags can vary between projects.

## 6.2 Project Setup and Structure

Consistent project structure is fundamental to maintainable Python development. While Python is famously flexible, establishing conventions early saves significant time and confusion as projects grow.

### 6.2.1 Modern Python Project Layout

Our recommended project structure balances simplicity with scalability:

```
my-project/
 pyproject.toml # Project configuration and dependencies
 README.md # Project overview and setup instructions
 .gitignore # Version control exclusions
 .pre-commit-config.yaml # Automated code quality checks
 src/ # Source code (src layout)
 my_project/
 __init__.py
 main.py # Entry point for applications
 core/ # Core modules
 tests/ # Test code
 __init__.py
 conftest.py # pytest configuration
 test_main.py
 docs/ # Documentation
 mkdocs.yml
 scripts/ # Utility scripts
 setup_dev.py
```

This structure follows several important principles:

**Src Layout:** Placing source code in a `src/` directory prevents accidental imports of uninstalled code during development and testing. This is particularly important for ensuring your tests run against the installed package, not just local files.

**Clear Separation:** Tests, documentation, and source code are clearly separated, making the project structure immediately understandable to new contributors.

**Configuration Co-location:** All project configuration lives in `pyproject.toml`, providing a single source of truth for project metadata, dependencies, and tool configuration.

## 6.2.2 Initializing New Projects

Create new projects following this structure using uv:

```
Create a new package project
uv init my-project --package
cd my-project

Set up the recommended structure
mkdir -p tests docs scripts
touch tests/__init__.py tests/conftest.py

Add essential development dependencies
uv add --dev pytest pytest-cov ruff mypy poethepoet pre-commit

Initialize git and pre-commit
git init
uv run pre-commit install
```

## 6.2.3 Application vs. Package Considerations

The structure varies slightly depending on whether you're building an **application** (end-user focused) or a **package** (library for other developers):

**Applications** typically include: - Configuration files and settings management - Entry point scripts or CLI interfaces  
- Deployment configurations - User documentation focused on usage

**Packages** emphasize: - Clean, documented APIs - Comprehensive test coverage - Developer documentation  
- Distribution metadata for PyPI

Most projects start as applications and may later extract reusable components into packages. Our recommended structure accommodates both paths naturally.

## 6.3 Team Collaboration Workflows

### 6.3.1 Code Review Standards

Establish clear expectations for code reviews that align with your automated tooling:

1. **Automated checks must pass:** All pre-commit hooks and CI checks should be green before review
2. **Test coverage requirements:** New code should include appropriate tests
3. **Documentation updates:** Public API changes require documentation updates
4. **Consistent style:** Rely on automated formatting (Ruff) rather than manual style discussions

### 6.3.2 Release Management

Define clear release processes that leverage your automation:

```
[tool.poe.tasks]
Release preparation
```

```

pre-release = ["check", "test-cov", "docs-build"]

Version management (using setuptools-scm for git-based versioning)
version = "python -m setuptools_scm"

Release workflow
release = { shell = """
 echo "Current version: $(python -m setuptools_scm)"
 git tag v$(python -m setuptools_scm --strip-dev)
 git push origin --tags
 uv build
 twine upload dist/*
 """ }

```

### 6.3.3 Managing Technical Debt

Use your automation to continuously monitor and address technical debt:

```

[tool.poe.tasks]
Code quality metrics
complexity = "radon cc src/ -a"
maintainability = "radon mi src/"
debt = ["complexity", "maintainability"]

Dependency analysis
deps-outdated = "pip list --outdated"
deps-security = "pip-audit"

```

Regular execution of these tasks helps maintain code quality and security over time.

## 6.4 Development Environment Standards

### 6.4.1 Editor-Agnostic Configuration

While developers may prefer different editors, project configuration should work consistently across environments. Our approach centers on `pyproject.toml` configuration that most modern Python editors understand:

```

[tool.ruff]
line-length = 88
target-version = "py39"

[tool.ruff.lint]
select = ["E", "F", "B", "I"]
ignore = ["E501"] # Line length handled by formatter

[tool.mypy]
python_version = "3.9"
strict = true
warn_return_any = true

```

```
[tool.pytest.ini_options]
testpaths = ["tests"]
addopts = "--cov=src --cov-report=term-missing"
```

This configuration works automatically with VS Code, PyCharm, Vim, Emacs, and other editors with Python support.

## 6.4.2 Development Environment Reproducibility

Ensure consistent development environments across team members:

```
[tool.poe.tasks]
doctor = { shell = """
 echo "Python version: $(python --version)"
 echo "uv version: $(uv --version)"
 echo "Project dependencies:"
 uv pip list
 echo "Development environment: Ready"
 """}
}
```

New team members can quickly verify their setup with `uv run poe doctor`.

### AI Tip: Onboarding Documentation

AI can help you write contributing guides, development setup instructions, and team conventions documents. Provide your `pyproject.toml` and project structure, then ask: “*Write a CONTRIBUTING.md that explains how to set up the development environment, run tests, and submit a pull request for this project.*” This is documentation that benefits everyone on the team and that AI can draft well from your project’s configuration.

This chapter has established the foundation for scalable project management through automation, consistent structure, and collaborative workflows. These practices become increasingly valuable as projects grow in size and complexity, ensuring that good habits established early continue to serve the project throughout its lifecycle.



# Chapter 7

## Multi-Platform Distribution

Moving beyond traditional Python package publishing, this chapter explores how to distribute complete applications to users across web, desktop, and mobile platforms—all from a single codebase. We'll focus on an architecture pattern that works exceptionally well with AI-assisted development: FastAPI backend + React frontend + multiple distribution targets.

### **i** A Note on Technologies Beyond Python

This chapter introduces React and Electron—technologies outside Python's ecosystem. Don't worry: the goal isn't to master these tools, but to understand the architecture that lets you ship to multiple platforms. Your Python skills remain central (the backend is still FastAPI), and the companion templates handle the frontend implementation details. With AI assistance, you can work effectively with React and Electron by understanding their role in the architecture rather than memorizing their APIs.

## 7.1 The Modern Distribution Challenge

Today's users expect applications everywhere:

- **Web:** Accessible from any browser, no installation required
- **Desktop:** Native experience on Windows, macOS, and Linux
- **Mobile:** Installable apps or Progressive Web Apps (PWAs)

For indie developers and small teams, maintaining separate codebases for each platform is impractical. The solution is an **API-first architecture** where a single backend serves multiple frontends, and those frontends can be packaged for different platforms.

### 7.1.1 What We're Building

By the end of this chapter, you'll understand how to structure a project that can ship to:

1. **Web:** Docker container deployable to any cloud platform
2. **PWA:** Installable web app with offline capabilities
3. **Desktop:** Native applications for Windows, macOS, and Linux via Electron

All from a single codebase, with automated builds through GitHub Actions.

## 7.2 Why FastAPI + React?

This isn't about which technologies are theoretically “best”—it's about which combination is most practical for AI-assisted development in 2025 and beyond.

### 7.2.1 Training Data Density

Python and JavaScript/React represent the largest pools of training data for AI models. This means:

- **Better code generation:** AI has seen more examples of these patterns
- **More accurate suggestions:** Edge cases are better handled
- **Richer ecosystem knowledge:** AI understands popular libraries deeply

When you ask AI to help with a FastAPI endpoint or a React component, you're working with the technologies AI knows best.

### 7.2.2 Ecosystem Maturity

Both ecosystems have:

- **Solved most common problems:** Authentication, forms, state management, API design
- **Extensive documentation:** AI can reference official docs and community resources
- **Active communities:** Stack Overflow answers, GitHub issues, blog posts

This maturity translates directly to better AI assistance.

### 7.2.3 Clean Separation

The API-first approach creates natural boundaries:

```

React Frontend ← → FastAPI Backend
(TypeScript) API (Python)

```

This separation means:

- AI can work on backend or frontend independently
- Changes to one side don't break the other (if the API contract holds)
- Different team members (or AI sessions) can work in parallel
- Testing is straightforward at each boundary

## 7.3 The Architecture Pattern

### 7.3.1 Project Structure for AI Collaboration

Here's the recommended structure for a multi-platform application:

```

my-app/
 backend/ # FastAPI backend (Python)
 src/
 my_app/
 __init__.py
 main.py # FastAPI application
 api/ # API routes
 __init__.py
 routes.py
 models/ # Pydantic models
 __init__.py
 schemas.py
 services/ # Business logic
 __init__.py
 core.py
 tests/
 pyproject.toml
 Dockerfile

 frontend/ # React frontend (TypeScript)
 src/
 components/ # React components
 hooks/ # Custom hooks
 services/ # API client
 App.tsx
 main.tsx
 public/
 package.json
 vite.config.ts
 Dockerfile

 electron/ # Desktop wrapper
 main.js # Electron main process
 preload.js # Preload script
 package.json

 docker/ # Container configurations
 docker-compose.yml # Development environment
 docker-compose.prod.yml # Production configuration
 Dockerfile.combined # Single container for deployment

 .github/
 workflows/
 ci.yml # Test on push
 build.yml # Build all platforms
 release.yml # Publish releases

 CLAUDE.md # AI context file
 README.md

```

This structure provides clear boundaries that both humans and AI can navigate effectively.

### 7.3.2 The CLAUDE.md Pattern

A key practice for AI-assisted development is maintaining a `CLAUDE.md` file (or similar context file) at the project root. This file provides AI assistants with project-specific context:

```
Project: My Application

Architecture
- Backend: FastAPI (Python 3.11+)
- Frontend: React 18 + TypeScript + Vite
- Desktop: Electron
- Database: SQLite (development), PostgreSQL (production)

Conventions
- Backend API routes follow RESTful conventions
- All API responses use Pydantic models
- Frontend components use functional style with hooks
- State management via React Query for server state

Directory Structure
- `/backend/src/my_app/api/` - API route definitions
- `/backend/src/my_app/models/` - Pydantic schemas
- `/frontend/src/components/` - React components
- `/frontend/src/services/` - API client functions

Development Commands
- Backend: `cd backend && uv run uvicorn my_app.main:app --reload`
- Frontend: `cd frontend && npm run dev`
- Both: `docker-compose up`

Key Decisions
- Using SQLite for simplicity; PostgreSQL in production
- Electron for desktop instead of Tauri (more mature, better AI support)
- Single Docker container for deployment (backend serves frontend static files)
```

This file serves as a “briefing document” for AI assistants, reducing the need to repeatedly explain project context.

## 7.4 Building the Backend: FastAPI

FastAPI is an excellent choice for this architecture because it’s:

- **Type-native:** Pydantic models provide automatic validation and documentation
- **Self-documenting:** OpenAPI/Swagger docs are generated automatically
- **Async-capable:** Handles concurrent requests efficiently
- **AI-friendly:** Extensive training data and clear patterns

### 7.4.1 A Minimal FastAPI Backend

```
backend/src/my_app/main.py
"""Main FastAPI application."""

from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from fastapi.staticfiles import StaticFiles
from pathlib import Path

from .api.routes import router

app = FastAPI(
 title="My Application",
 description="API for my multi-platform application",
 version="1.0.0",
)

CORS configuration for development
app.add_middleware(
 CORSMiddleware,
 allow_origins=["http://localhost:5173"], # Vite dev server
 allow_credentials=True,
 allow_methods=["*"],
 allow_headers=["*"],
)

API routes
app.include_router(router, prefix="/api")

Serve frontend static files in production
frontend_path = Path(__file__).parent.parent.parent.parent / "frontend" / "dist"
if frontend_path.exists():
 app.mount("/", StaticFiles(directory=frontend_path, html=True),
name="frontend")
```

## 7.4.2 API Routes with Pydantic Models

```
backend/src/my_app/api/routes.py
"""API route definitions."""

from fastapi import APIRouter, HTTPException
from typing import List

from ..models.schemas import Item, ItemCreate, ItemUpdate

router = APIRouter()

In-memory storage for demonstration
items_db: dict[int, Item] = {}
next_id = 1
```

```

@router.get("/items", response_model=List[Item])
async def list_items():
 """List all items."""
 return list(items_db.values())

@router.post("/items", response_model=Item, status_code=201)
async def create_item(item: ItemCreate):
 """Create a new item."""
 global next_id
 new_item = Item(id=next_id, **item.model_dump())
 items_db[next_id] = new_item
 next_id += 1
 return new_item

@router.get("/items/{item_id}", response_model=Item)
async def get_item(item_id: int):
 """Get a specific item by ID."""
 if item_id not in items_db:
 raise HTTPException(status_code=404, detail="Item not found")
 return items_db[item_id]

@router.put("/items/{item_id}", response_model=Item)
async def update_item(item_id: int, item: ItemUpdate):
 """Update an existing item."""
 if item_id not in items_db:
 raise HTTPException(status_code=404, detail="Item not found")

 stored_item = items_db[item_id]
 update_data = item.model_dump(exclude_unset=True)
 updated_item = stored_item.model_copy(update=update_data)
 items_db[item_id] = updated_item
 return updated_item

@router.delete("/items/{item_id}", status_code=204)
async def delete_item(item_id: int):
 """Delete an item."""
 if item_id not in items_db:
 raise HTTPException(status_code=404, detail="Item not found")
 del items_db[item_id]

```

### 7.4.3 Pydantic Models

```

backend/src/my_app/models/schemas.py
"""Pydantic models for API request/response validation."""

```

```
from pydantic import BaseModel, Field
from datetime import datetime
from typing import Optional

class ItemBase(BaseModel):
 """Base item schema with common fields."""
 title: str = Field(..., min_length=1, max_length=100)
 description: Optional[str] = Field(None, max_length=500)
 completed: bool = False

class ItemCreate(ItemBase):
 """Schema for creating a new item."""
 pass

class ItemUpdate(BaseModel):
 """Schema for updating an item (all fields optional)."""
 title: Optional[str] = Field(None, min_length=1, max_length=100)
 description: Optional[str] = Field(None, max_length=500)
 completed: Optional[bool] = None

class Item(ItemBase):
 """Schema for item responses."""
 id: int
 created_at: datetime = Field(default_factory=datetime.utcnow)

 class Config:
 from_attributes = True
```

The type hints and Pydantic models serve double duty: they validate data at runtime AND they help AI generate more accurate code when working with your API.

## 7.5 Building the Frontend: React

The frontend consumes the FastAPI backend through a typed API client.

### 7.5.1 Project Setup with Vite

```
Create React project with TypeScript
npm create vite@latest frontend -- --template react-ts
cd frontend
npm install
npm install @tanstack/react-query axios
```

## 7.5.2 API Client

```
// frontend/src/services/api.ts
import axios from 'axios';

const API_BASE_URL = import.meta.env.VITE_API_URL ||
'http://localhost:8000/api';

const api = axios.create({
 baseURL: API_BASE_URL,
 headers: {
 'Content-Type': 'application/json',
 },
});

// Types matching backend Pydantic models
export interface Item {
 id: number;
 title: string;
 description: string | null;
 completed: boolean;
 created_at: string;
}

export interface ItemCreate {
 title: string;
 description?: string;
 completed?: boolean;
}

export interface ItemUpdate {
 title?: string;
 description?: string;
 completed?: boolean;
}

// API functions
export const itemsApi = {
 list: async (): Promise<Item[]> => {
 const response = await api.get('/items');
 return response.data;
 },

 get: async (id: number): Promise<Item> => {
 const response = await api.get(`/items/${id}`);
 return response.data;
 },

 create: async (item: ItemCreate): Promise<Item> => {
 const response = await api.post('/items', item);
 return response.data;
 },
}
```

```

update: async (id: number, item: ItemUpdate): Promise<Item> => {
 const response = await api.put(`/items/${id}`, item);
 return response.data;
},

delete: async (id: number): Promise<void> => {
 await api.delete(`/items/${id}`);
},
};

```

### 7.5.3 React Component with React Query

```

// frontend/src/components/ItemList.tsx
import { useQuery, useMutation, useQueryClient } from
 '@tanstack/react-query';
import { itemsApi, Item, ItemCreate } from '../services/api';
import { useState } from 'react';

export function ItemList() {
 const queryClient = useQueryClient();
 const [newTitle, setNewTitle] = useState('');

 // Fetch items
 const { data: items, isLoading, error } = useQuery({
 queryKey: ['items'],
 queryFn: itemsApi.list,
 });

 // Create mutation
 const createMutation = useMutation({
 mutationFn: (item: ItemCreate) => itemsApi.create(item),
 onSuccess: () => {
 queryClient.invalidateQueries({ queryKey: ['items'] });
 setNewTitle('');
 },
 });

 // Toggle completion mutation
 const toggleMutation = useMutation({
 mutationFn: ({ id, completed }: { id: number; completed: boolean })
=>
 itemsApi.update(id, { completed }),
 onSuccess: () => {
 queryClient.invalidateQueries({ queryKey: ['items'] });
 },
 });

 // Delete mutation
 const deleteMutation = useMutation({

```

```

mutationFn: (id: number) => itemsApi.delete(id),
onSuccess: () => {
 queryClient.invalidateQueries({ queryKey: ['items'] });
},
});

if (isLoading) return <div>Loading...</div>;
if (error) return <div>Error loading items</div>;

return (
 <div className="item-list">
 <form
 onSubmit={(e) => {
 e.preventDefault();
 if (newTitle.trim()) {
 createMutation.mutate({ title: newTitle });
 }
 }}
 >
 <input
 type="text"
 value={newTitle}
 onChange={(e) => setNewTitle(e.target.value)}
 placeholder="Add new item..."
 />
 <button type="submit">Add</button>
 </form>

 {items?.map((item: Item) => (
 <li key={item.id}>
 <input
 type="checkbox"
 checked={item.completed}
 onChange={() =>
 toggleMutation.mutate({
 id: item.id,
 completed: !item.completed,
 })
 }
 />

 {item.title}

 <button onClick={() => deleteMutation.mutate(item.id)}>
 Delete
 </button>

))}

 </div>

```

```
);
}
```

## 7.6 Distribution Target: Web with Docker

The simplest distribution target is a containerized web application.

### 7.6.1 Single Container Deployment

For simpler deployments, combine backend and frontend in a single container:

```
docker/Dockerfile.combined
Multi-stage build for combined backend + frontend

Stage 1: Build frontend
FROM node:20-alpine AS frontend-build
WORKDIR /app/frontend
COPY frontend/package*.json ./
RUN npm ci
COPY frontend/ ./
RUN npm run build

Stage 2: Production image
FROM python:3.11-slim
WORKDIR /app

Install uv
RUN pip install uv

Copy backend
COPY backend/pyproject.toml backend/uv.lock ./
RUN uv sync --frozen --no-dev

COPY backend/src ./src

Copy built frontend
COPY --from=frontend-build /app/frontend/dist ./frontend/dist

Expose port
EXPOSE 8000

Run the application
CMD ["uv", "run", "uvicorn", "my_app.main:app", "--host", "0.0.0.0",
"--port", "8000"]
```

### 7.6.2 Docker Compose for Development

```
docker/docker-compose.yml
version: '3.8'
```

```

services:
 backend:
 build:
 context: ../backend
 dockerfile: Dockerfile
 ports:
 - "8000:8000"
 volumes:
 - ../backend/src:/app/src
 environment:
 - DEBUG=true
 command: uv run uvicorn my_app.main:app --reload --host 0.0.0.0

 frontend:
 build:
 context: ../frontend
 dockerfile: Dockerfile.dev
 ports:
 - "5173:5173"
 volumes:
 - ../frontend/src:/app/src
 environment:
 - VITE_API_URL=http://localhost:8000/api
 command: npm run dev -- --host

```

### 7.6.3 Cloud Deployment

The combined container can be deployed to any container platform:

```

Build the production image
docker build -f docker/Dockerfile.combined -t my-app:latest .

Push to registry (example: GitHub Container Registry)
docker tag my-app:latest ghcr.io/username/my-app:latest
docker push ghcr.io/username/my-app:latest

Deploy to fly.io (example)
flyctl deploy

```

## 7.7 Distribution Target: Progressive Web App (PWA)

PWAs provide an app-like experience without requiring app store distribution.

### 7.7.1 Vite PWA Configuration

```
npm install vite-plugin-pwa -D
```

```
// frontend/vite.config.ts
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
import { VitePWA } from 'vite-plugin-pwa';

export default defineConfig({
 plugins: [
 react(),
 VitePWA({
 registerType: 'autoUpdate',
 includeAssets: ['favicon.ico', 'robots.txt',
'apple-touch-icon.png'],
 manifest: {
 name: 'My Application',
 short_name: 'MyApp',
 description: 'A multi-platform application',
 theme_color: '#ffffff',
 icons: [
 {
 src: 'pwa-192x192.png',
 sizes: '192x192',
 type: 'image/png',
 },
 {
 src: 'pwa-512x512.png',
 sizes: '512x512',
 type: 'image/png',
 },
],
 },
 },
],
 workbox: {
 globPatterns: ['**/*.{js,css,html,ico,png,svg}'],
 runtimeCaching: [
 {
 urlPattern: /^https:\/\/api\.example\.com\/.*\/i,
 handler: 'NetworkFirst',
 options: {
 cacheName: 'api-cache',
 expiration: {
 maxEntries: 100,
 maxAgeSeconds: 60 * 60 * 24, // 24 hours
 },
 },
 },
],
 },
});
```

## 7.7.2 When PWA Is Enough

Consider PWA when:

- Your app is primarily web-based
- Users have reliable internet (or you implement offline support)
- You want to avoid app store review processes
- You need quick deployment and updates

## 7.8 Distribution Target: Desktop with Electron

For a full native desktop experience, Electron wraps your web application in a native shell.

### 7.8.1 Why Electron for Indie Developers

Despite larger bundle sizes compared to alternatives like Tauri, Electron offers advantages for indie developers:

1. **Maturity:** VS Code, Slack, Discord, and Figma prove it scales
2. **Auto-update:** Built-in update mechanisms via electron-updater
3. **Training data:** More examples mean better AI assistance
4. **Community:** Extensive documentation and solved problems
5. **Consistency:** Same behaviour across platforms

### 7.8.2 Electron Project Structure

```
// electron/main.js
const { app, BrowserWindow, Menu } = require('electron');
const path = require('path');
const { autoUpdater } = require('electron-updater');

// Handle creating/removing shortcuts on Windows when
installing/uninstalling
if (require('electron-squirrel-startup')) {
 app.quit();
}

let mainWindow;

function createWindow() {
 mainWindow = new BrowserWindow({
 width: 1200,
 height: 800,
 webPreferences: {
 preload: path.join(__dirname, 'preload.js'),
 contextIsolation: true,
 nodeIntegration: false,
 },
 });

 // In development, load from Vite dev server
```

```

 if (process.env.NODE_ENV === 'development') {
 mainWindow.loadURL('http://localhost:5173');
 mainWindow.webContents.openDevTools();
 } else {
 // In production, load the built frontend
 mainWindow.loadFile(path.join(__dirname,
'../frontend/dist/index.html'));
 }
 }
}

app.whenReady().then(() => {
 createWindow();

 // Check for updates in production
 if (process.env.NODE_ENV !== 'development') {
 autoUpdater.checkForUpdatesAndNotify();
 }
});

app.on('window-all-closed', () => {
 if (process.platform !== 'darwin') {
 app.quit();
 }
});

app.on('activate', () => {
 if (BrowserWindow.getAllWindows().length === 0) {
 createWindow();
 }
});

// Auto-updater events
autoUpdater.on('update-available', () => {
 console.log('Update available');
});

autoUpdater.on('update-downloaded', () => {
 console.log('Update downloaded');
 // Optionally prompt user to restart
});

```

### 7.8.3 Preload Script

```

// electron/preload.js
const { contextBridge, ipcRenderer } = require('electron');

// Expose protected methods that allow the renderer process to use
// the ipcRenderer without exposing the entire object
contextBridge.exposeInMainWorld('electronAPI', {
 getVersion: () => ipcRenderer.invoke('get-version'),

```

```
platform: process.platform,
});
```

## 7.8.4 Electron Package Configuration

```
{
 "name": "my-app-desktop",
 "version": "1.0.0",
 "description": "My Application - Desktop",
 "main": "main.js",
 "scripts": {
 "start": "electron .",
 "build": "electron-builder",
 "build:win": "electron-builder --win",
 "build:mac": "electron-builder --mac",
 "build:linux": "electron-builder --linux"
 },
 "build": {
 "appId": "com.example.myapp",
 "productName": "My Application",
 "directories": {
 "output": "dist"
 },
 "files": [
 "main.js",
 "preload.js",
 "../frontend/dist/**/*"
],
 "win": {
 "target": ["nsis", "portable"],
 "icon": "icons/icon.ico"
 },
 "mac": {
 "target": ["dmg", "zip"],
 "icon": "icons/icon.icns",
 "category": "public.app-category.productivity"
 },
 "linux": {
 "target": ["AppImage", "deb"],
 "icon": "icons",
 "category": "Utility"
 },
 "publish": {
 "provider": "github",
 "owner": "username",
 "repo": "my-app"
 }
 },
 "dependencies": {
 "electron-updater": "^6.1.0"
 }
}
```

```

 },
 "devDependencies": {
 "electron": "^28.0.0",
 "electron-builder": "^24.0.0"
 }
}

```

### 7.8.5 Handling the Backend in Desktop Apps

For desktop apps, you have two options for the backend:

**Option 1: Remote Backend** The desktop app connects to a hosted API (same as the web version).

**Option 2: Embedded Backend** Bundle the Python backend with the Electron app and run it locally.

```

// electron/main.js - Embedded backend example
const { spawn } = require('child_process');
const path = require('path');

let backendProcess;

function startBackend() {
 const backendPath = path.join(__dirname, '../backend');

 backendProcess = spawn('python', ['-m', 'uvicorn', 'my_app.main:app',
'--port', '8000'], {
 cwd: backendPath,
 env: { ...process.env, PYTHONUNBUFFERED: '1' },
 });

 backendProcess.stdout.on('data', (data) => {
 console.log(`Backend: ${data}`);
 });

 backendProcess.stderr.on('data', (data) => {
 console.error(`Backend error: ${data}`);
 });
}

app.whenReady().then(() => {
 startBackend();

 // Wait for backend to start, then create window
 setTimeout(createWindow, 2000);
});

app.on('before-quit', () => {
 if (backendProcess) {
 backendProcess.kill();
 }
}

```

```
});
```

For a more robust embedded backend, consider using PyInstaller to bundle Python into a standalone executable.

## 7.9 CI/CD for Multi-Platform Builds

GitHub Actions can automate building and releasing for all platforms.

### 7.9.1 Continuous Integration Workflow

```
.github/workflows/ci.yml
name: CI

on:
 push:
 branches: [main]
 pull_request:
 branches: [main]

jobs:
 backend-test:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v4

 - name: Set up Python
 uses: actions/setup-python@v5
 with:
 python-version: '3.11'

 - name: Install uv
 run: pip install uv

 - name: Install dependencies
 working-directory: ./backend
 run: uv sync --all-extras

 - name: Run tests
 working-directory: ./backend
 run: uv run pytest --cov

 frontend-test:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v4

 - name: Set up Node
 uses: actions/setup-node@v4
 with:
```

```

 node-version: '20'
 cache: 'npm'
 cache-dependency-path: frontend/package-lock.json

 - name: Install dependencies
 working-directory: ./frontend
 run: npm ci

 - name: Run tests
 working-directory: ./frontend
 run: npm test

 - name: Build
 working-directory: ./frontend
 run: npm run build

```

## 7.9.2 Multi-Platform Build Workflow

```

.github/workflows/build.yml
name: Build

on:
 push:
 tags:
 - 'v*'

jobs:
 build-web:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v4

 - name: Set up Docker Buildx
 uses: docker/setup-buildx-action@v3

 - name: Login to GitHub Container Registry
 uses: docker/login-action@v3
 with:
 registry: ghcr.io
 username: ${{ github.actor }}
 password: ${{ secrets.GITHUB_TOKEN }}

 - name: Build and push
 uses: docker/build-push-action@v5
 with:
 context: .
 file: ./docker/Dockerfile.combined
 push: true
 tags: ghcr.io/${{ github.repository }}:${{ github.ref_name }}

```

```

build-desktop:
 runs-on: ${{ matrix.os }}
 strategy:
 matrix:
 os: [ubuntu-latest, windows-latest, macos-latest]

 steps:
 - uses: actions/checkout@v4

 - name: Set up Node
 uses: actions/setup-node@v4
 with:
 node-version: '20'

 - name: Install frontend dependencies
 working-directory: ./frontend
 run: npm ci

 - name: Build frontend
 working-directory: ./frontend
 run: npm run build

 - name: Install Electron dependencies
 working-directory: ./electron
 run: npm ci

 - name: Build Electron app
 working-directory: ./electron
 run: npm run build
 env:
 GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}

 - name: Upload artifacts
 uses: actions/upload-artifact@v4
 with:
 name: desktop-${{ matrix.os }}
 path: electron/dist/*

release:
 needs: [build-web, build-desktop]
 runs-on: ubuntu-latest
 steps:
 - name: Download all artifacts
 uses: actions/download-artifact@v4

 - name: Create Release
 uses: softprops/action-gh-release@v1
 with:
 files: |
 desktop-ubuntu-latest/*
 desktop-windows-latest/*

```

```
desktop-macos-latest/*
generate_release_notes: true
```

## 7.10 End-to-End Testing with Playwright

Once you're distributing across web, PWA, and desktop, you need confidence that the full application works as users experience it. Unit tests verify your Python backend logic; Playwright verifies that buttons click, pages load, and workflows complete.

### 7.10.1 Why Playwright

Playwright is a browser automation framework with first-class Python support. Compared to alternatives like Selenium or Cypress:

- **Multi-browser:** Tests run against Chromium, Firefox, and WebKit from a single API
- **Python-native:** Integrates directly with pytest via `pytest-playwright`
- **Electron support:** Can test desktop apps, not just web pages
- **Auto-waiting:** No manual sleep statements—Playwright waits for elements to be ready
- **Trace viewer:** Built-in debugging tool that records screenshots, DOM snapshots, and network activity

### 7.10.2 Installation

```
pip install pytest-playwright
playwright install
```

### 7.10.3 Testing Your Web Application

A basic test that verifies your FastAPI + React app works end-to-end:

```
tests/e2e/test_web_app.py
import pytest
from playwright.sync_api import Page, expect

@pytest.fixture(scope="session")
def base_url():
 """URL where your app is running during tests."""
 return "http://localhost:8000"

def test_homepage_loads(page: Page, base_url):
 """Verify the app loads and shows expected content."""
 page.goto(base_url)
 expect(page).to_have_title("My Application")
 expect(page.locator("nav")).to_be_visible()

def test_create_item_workflow(page: Page, base_url):
 """Test a complete user workflow: create and verify an item."""
 page.goto(base_url)
```

```

Click the "New Item" button
page.click("button:has-text('New Item')")

Fill in the form
page.fill('input[name="title"]', "Test Item")
page.fill('textarea[name="description"]', "Created by Playwright")

Submit and verify
page.click("button:has-text('Save')")
expect(page.locator("text=Test Item")).to_be_visible()

def test_api_error_handling(page: Page, base_url):
 """Verify the UI handles API errors gracefully."""
 page.goto(f"{base_url}/items/nonexistent-id")
 expect(page.locator("text=Not Found")).to_be_visible()

```

#### 7.10.4 Testing Electron Desktop Apps

Playwright can connect to Electron applications directly, testing your desktop builds with the same API:

```

tests/e2e/test_desktop.py
import pytest
from playwright.sync_api import Playwright

@pytest.fixture
def electron_app(playwright: Playwright):
 """Launch the Electron app for testing."""
 app = playwright.electron.launch(args=["electron/main.js"])
 yield app
 app.close()

@pytest.fixture
def window(electron_app):
 """Get the main application window."""
 window = electron_app.first_window
 window.wait_for_load_state("domcontentloaded")
 return window

def test_app_launches(window):
 """Verify the desktop app launches and shows the main view."""
 expect(window).to_have_title("My Application")

def test_offline_functionality(window, electron_app):
 """Test that the app works without network access."""
 # Electron apps with embedded backends should work offline
 context = window.context
 context.set_offline(True)

 window.reload()

```

```
expect(window.locator("nav")).to_be_visible()
expect(window.locator("text=Connection Error")).not_to_be_visible()
```

### 7.10.5 Integrating E2E Tests into CI/CD

Add Playwright to your existing GitHub Actions workflow:

```
In .github/workflows/ci.yml
e2e-tests:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v4
 - name: Set up Python
 uses: actions/setup-python@v4
 with:
 python-version: "3.11"
 - name: Install dependencies
 run: |
 pip install uv
 uv sync
 uv playwright install --with-deps chromium
 - name: Start application
 run: uv run uvicorn my_app.main:app --host 0.0.0.0 --port 8000 &
 - name: Run E2E tests
 run: uv run pytest tests/e2e/ --browser chromium
 - name: Upload test traces on failure
 if: failure()
 uses: actions/upload-artifact@v4
 with:
 name: playwright-traces
 path: test-results/
```

### 7.10.6 E2E Testing Best Practices

1. **Test user workflows, not implementation details:** Click buttons by their text, not by CSS selectors that change with refactors
2. **Keep E2E tests focused:** A handful of critical path tests is worth more than exhaustive UI coverage
3. **Use the trace viewer for debugging:** Run `playwright show-trace trace.zip` to step through failures visually
4. **Run against a real backend:** E2E tests should exercise the full stack—mocking defeats the purpose
5. **Mark E2E tests separately:** Use `@pytest.mark.e2e` so you can run them independently of unit tests

```
pyproject.toml
[tool.pytest.ini_options]
markers = [
 "e2e: end-to-end browser tests (require running application)",
]
```

## 7.11 The Human’s Role in This Architecture

With this architecture in place, the human developer’s role shifts to:

### 7.11.1 Architecture Decisions

- Defining the API contract (endpoints, data models)
- Choosing the right distribution targets for the use case
- Making tradeoffs between complexity and capability

### 7.11.2 Verification

- Reviewing AI-generated code for correctness
- Ensuring tests cover critical paths
- Validating that the API contract is maintained

### 7.11.3 Orchestration

- Managing the build and release pipeline
- Coordinating between backend and frontend work
- Handling platform-specific edge cases

### 7.11.4 User Experience

- Making design decisions AI can’t make
- Understanding user needs and requirements
- Testing across platforms and environments

The AI handles implementation details—writing the FastAPI routes, React components, Docker configurations—while you focus on the architecture, verification, and delivery.

## 7.12 Summary

Multi-platform distribution is achievable for indie developers through:

1. **API-first architecture:** FastAPI backend + React frontend with clear boundaries
2. **Container deployment:** Docker for web distribution
3. **PWA capabilities:** Installable web apps with offline support
4. **Electron packaging:** Native desktop apps for Windows, macOS, and Linux
5. **Automated CI/CD:** GitHub Actions for multi-platform builds

This architecture is deliberately simple—“simple but not simplistic”—because simplicity enables effective AI collaboration. Clear structures, typed interfaces, and well-defined boundaries let you leverage AI for implementation while maintaining control over the architecture and delivery.

In the next chapter, we’ll explore how to structure these templates and configurations for maximum reusability across projects.

## Chapter 8

# Beyond Scripts: Notebooks, Dashboards, and Interactive Python

### **i** Chapter Overview

Scripts aren't the only way to ship Python. This chapter explores notebooks, dashboards, and interactive tools—legitimate ways to share, deploy, and deliver Python work.

## 8.1 Why This Chapter Appears Late (But Isn't an Afterthought)

This chapter comes near the end of the book, but that's intentional—not because notebooks are less important. We wanted you to first understand the complete script-based workflow: project structure, testing, documentation, packaging, and distribution. These fundamentals apply whether you're writing scripts OR notebooks.

Now that you understand the “full” workflow, you can appreciate both when notebooks simplify things and when they create limitations. You'll recognise that `requirements.txt` matters for Binder just as it does for pip installs, that documentation practices transfer to notebook markdown cells, and that `nbdev`'s testing approach builds on `pytest` concepts you already know.

Notebooks are a first-class citizen in the Python ecosystem—not an alternative for people who can't handle “real” development. Many professional data scientists, researchers, and educators ship exclusively through notebooks. This chapter gives you the complete picture.

### 💡 The Trade-Off Up Front

Notebooks are often **easier to share** than packaged scripts—a Colab link gives anyone instant access with zero installation. But they **expose your code** by default, which creates friction for some audiences. Not everyone wants to scroll past Python cells to see results. Tools like Voilà and Mercury address this, but it's worth knowing: the simplicity of notebooks comes with visibility trade-offs.

## 8.2 The Three Ways to Write Python

There are only three ways to write Python code:

1. **REPL** - Interactive exploration (not for shipping)
2. **Scripts** - Traditional apps, packages, CLI tools (this book's main focus)
3. **Notebooks** - Data analysis, reports, teaching, prototypes

This book has focused primarily on scripts. But for many Python practitioners—especially in data science, education, and research—notebooks ARE the deliverable. A well-structured notebook with a sharing link is shipping.

## 8.3 When Notebooks Make Sense

Notebooks excel when:

- **The narrative matters** - Analysis with explanation, teaching materials
- **Exploration is the product** - Data investigation, research findings
- **Visuals are central** - Charts, plots, interactive widgets
- **Reproducibility is key** - Share exact environment and execution order
- **Zero-install is required** - Viewers shouldn't need to set up Python

Notebooks are less suited for:

- Production APIs or services
- CLI tools
- Reusable libraries (though nbdev challenges this)
- Long-running applications

## 8.4 Sharing and Viewing

### 8.4.1 GitHub Native Rendering

GitHub renders `.ipynb` files automatically. Simply push your notebook:

```
git add analysis.ipynb
git commit -m "Add quarterly analysis"
git push
```

Viewers see rendered output without running code. Limitations: large notebooks may not render, and formatting can be inconsistent.

### 8.4.2 nbviewer

nbviewer.org provides cleaner rendering:

```
https://nbviewer.org/github/username/repo/blob/main/notebook.ipynb
```

- Better formatting than GitHub
- Supports Gists
- Cacheable links for sharing

### 8.4.3 Gists for Quick Sharing

For standalone notebooks:

1. Create a Gist at [gist.github.com](https://gist.github.com)
2. Upload your `.ipynb` file
3. Share via nbviewer: [https://nbviewer.org/gist/username/gist\\_id](https://nbviewer.org/gist/username/gist_id)

## 8.5 Zero-Install Execution

The power of notebooks: viewers can RUN your code without installing anything.

### 8.5.1 Google Colab

The most accessible option. Add a badge to your README:

```
[![Open In Colab](BADGE_URL)](COLAB_URL)
```

Where:

```
BADGE_URL = https://colab.research.google.com
 /assets/colab-badge.svg
COLAB_URL = https://colab.research.google.com
 /github/USER/REPO/blob/main/FILE
```

**Colab advantages:**

- Zero setup for viewers
- Free GPU/TPU access
- Google Drive integration
- GitHub integration (open directly from repos)

**Colab workflow with GitHub:**

1. Develop locally or in Colab
2. Save to GitHub (File → Save a copy to GitHub)
3. Share Colab link that opens from GitHub
4. Viewers get latest version automatically

This gives you version control (GitHub) with zero-install execution (Colab)—a “clunky Dropbox” that’s actually better because it’s versioned.

### 8.5.2 Binder

mybinder.org turns any GitHub repo into interactive notebooks:

```
[![Binder](https://mybinder.org/badge_logo.svg)](https://mybinder.org/v2/gh/username/repo/main)
```

### Binder advantages:

- Works with `requirements.txt` or `environment.yml`
- Full JupyterLab environment
- No Google account required

### Binder limitations:

- Slower startup (builds environment)
- Sessions timeout
- Limited resources

## 8.5.3 Kaggle Kernels

For data science work, Kaggle provides:

- Free GPU access
- Built-in datasets
- Community sharing
- Competition integration

## 8.6 Notebooks as Applications

Transform notebooks into interactive applications that hide the code.

### 8.6.1 Voilà

Voilà converts notebooks into standalone dashboards:

```
pip install voila
voila notebook.ipynb
```

- Renders only output cells (code hidden)
- Supports ipywidgets for interactivity
- Deploy on Heroku, Binder, or your own server

### 8.6.2 Mercury

Mercury turns notebooks into web apps:

```
Add YAML header to notebook

title: My Analysis
description: Interactive data explorer
params:
 date_range:
 input: slider
 min: 2020
 max: 2024

```

- Automatic widget generation from parameters
- PDF/HTML export
- Authentication support
- Self-hostable

### 8.6.3 Streamlit (Notebook-Adjacent)

Not notebooks, but similar rapid-development workflow:

```
app.py
import streamlit as st
import pandas as pd

st.title("Data Explorer")
data = pd.read_csv("data.csv")
st.dataframe(data)
```

```
streamlit run app.py
```

- Python scripts, not notebooks
- But similar “write and see” iteration
- Easy deployment via Streamlit Cloud

### 8.6.4 Panel and HoloViz

For more complex dashboards:

- **Panel** - Flexible dashboarding from notebooks
- **HoloViews** - High-level plotting
- **hvPlot** - Interactive pandas plots

## 8.7 Notebooks as Libraries: nbdev

nbdev flips the script: develop libraries FROM notebooks.

```
notebook.ipynb → Python package + docs + tests
```

#### See It In Action

We dedicate Chapter 9 to building a complete package with nbdev—TextKit, a text analysis library. That case study parallels the SimpleBot chapter (Chapter 4) but follows the notebook-first workflow. Think of nbdev as an **alternative path to the same destination**: instead of writing `.py` files and separate docs, you write notebooks that generate both. The end result—a published package—is the same.

#### The nbdev philosophy:

- Write code, tests, and documentation together
- Export specific cells to modules
- Generate API docs automatically
- Literate programming for Python

**Basic workflow:**

```
#| export
def process_data(df):
 """Clean and transform dataframe.

 Parameters

 df : DataFrame
 Input data

 Returns

 DataFrame
 Cleaned data
 """
 return df.dropna()
```

```
#| test
def test_process_data():
 df = pd.DataFrame({'a': [1, None, 3]})
 result = process_data(df)
 assert len(result) == 2
```

**When to use nbdev:**

- You naturally develop in notebooks
- Documentation and code should live together
- Teaching libraries where explanation matters

## 8.8 Low-Code Alternatives

### 8.8.1 Anvil

Anvil provides a different model:

- Drag-and-drop UI builder
- Write Python for event handlers
- Hosted deployment included
- Database and user management built-in

```
Behind a button click
def button_click(self, **event_args):
 self.label.text = "Hello, " + self.text_box.text
```

**Anvil is good for:**

- Internal business tools
- Forms and data entry
- Quick prototypes with real UIs
- Teaching event-driven programming

**Trade-offs:**

- Vendor lock-in

- Less “Pythonic” project structure
- Limited customisation

## 8.9 Notebook Best Practices for Shipping

### 8.9.1 Structure Your Notebooks

1. Title and Overview (Markdown)
2. Setup and Imports
3. Data Loading
4. Analysis Sections (numbered)
5. Conclusions
6. Appendix (helper functions, details)

### 8.9.2 Environment Management

Include a requirements cell:

```
Requirements: pandas>=1.5, matplotlib>=3.6, seaborn>=0.12
```

Or ship with `requirements.txt` / `environment.yml` for Binder.

### 8.9.3 Clear Outputs vs. Keep Outputs

| Approach      | When                                      |
|---------------|-------------------------------------------|
| Clear outputs | Version control (smaller diffs)           |
| Keep outputs  | Sharing (viewers see results immediately) |

Consider: clear for development, render for sharing.

### 8.9.4 Use nbstripout

Automatically strip outputs on commit:

```
pip install nbstripout
nbstripout --install
```

### 8.9.5 Cell Tags and Metadata

Use tags for tools like Voilà and nbdev:

- `#| hide` - Hide cell in output
- `#| export` - Export to module (nbdev)
- `#| test` - Mark as test (nbdev)

## 8.10 Comparison Table

| Tool              | Type        | Hosting          | Best For                  |
|-------------------|-------------|------------------|---------------------------|
| GitHub + nbviewer | View only   | Free             | Simple sharing            |
| Colab             | Interactive | Free (Google)    | Zero-install, GPU         |
| Binder            | Interactive | Free             | Reproducible environments |
| Voilà             | Dashboard   | Self-host/Binder | Hide code, show results   |
| Mercury           | Web app     | Self-host        | Parameterized reports     |
| Streamlit         | Web app     | Streamlit Cloud  | Rapid app development     |
| nbdev             | Library dev | PyPI             | Literate programming      |
| Anvil             | Full app    | Anvil servers    | Low-code business apps    |

## 8.11 When to Graduate from Notebooks

Notebooks are great, but sometimes you need to move to scripts:

- **Tests are growing** - pytest is better than notebook tests
- **Reuse across projects** - Package your code
- **Production deployment** - APIs, services, CLI tools
- **Team collaboration** - Notebooks have merge conflicts

The path: Notebook → extract functions to `.py` → package → tests → CI/CD.

Or use nbdev to keep working in notebooks while generating proper packages.

## 8.12 Summary

- Notebooks are a legitimate shipping format
- “Shipping” can mean a Colab link, not just a packaged app
- Multiple tools exist to share, execute, and transform notebooks
- Choose based on your audience: viewers, runners, or users
- Know when to graduate to scripts (or use nbdev to avoid the choice)

### AI Tip: Notebooks and AI Assistants

Many notebook environments now include built-in AI assistants. Google Colab has Gemini, and JupyterLab supports extensions for Copilot and other AI tools. These are particularly useful for exploratory data analysis — ask AI to generate plotting code, explain error messages, or suggest the next analysis step. The interactive, cell-by-cell nature of notebooks makes AI collaboration especially natural: generate code in one cell, review and modify it, then run it immediately.

**Next:** In Chapter 9, we’ll build TextKit — a complete Python package developed entirely in notebooks using nbdev.

## 8.13 Exercises

1. **Share a notebook:** Push a notebook to GitHub and create both an nbviewer link and a Colab badge.
2. **Try Binder:** Add a `requirements.txt` to a repo and create a Binder link.

3. **Build a dashboard:** Take an analysis notebook and convert it to a Voilà dashboard.
4. **Explore nbdev:** Create a simple function in a notebook and export it to a Python module using nbdev.



## Chapter 9

# Case Study: From Notebook to Package with nbdev

### **i** Chapter Overview

This case study parallels Chapter 4 (SimpleBot), but follows a notebook-first workflow. We'll build TextKit—a text analysis library—entirely in Jupyter notebooks, then ship it as a published Python package using nbdev.

## 9.1 Project Overview

TextKit is a lightweight text analysis library that provides simple utilities for analyzing text. Key features include:

- Word and character statistics
- Readability scoring (Flesch-Kincaid, etc.)
- Basic sentiment indicators
- Text cleaning utilities

This project is ideal for our notebook case study because:

- **Natural notebook fit:** Text analysis involves exploration and visualization
- **Keeps the theme:** Complements SimpleBot's chatbot focus (analyzing what bots produce)
- **Real utility:** Functions you'd actually use in data analysis
- **Right size:** Small enough to complete, complex enough to demonstrate the workflow

By the end of this chapter, you'll have a package published to PyPI—built entirely from notebooks.

## 9.2 Why nbdev for This Project?

In Chapter 8, we introduced nbdev as a way to develop libraries from notebooks. Here's why it fits TextKit:

| Traditional Workflow                 | nbdev Workflow                |
|--------------------------------------|-------------------------------|
| Write code in <code>.py</code> files | Write code in notebooks       |
| Write separate test files            | Tests live next to code       |
| Write docs separately                | Docs generated from notebooks |
| Context switching                    | Single environment            |

For exploratory, iterative work like text analysis, nbdev keeps everything together.

## 9.3 1. Setting Up the nbdev Project

### 9.3.1 Installing nbdev

```
pip install nbdev
```

### 9.3.2 Creating the Project

```
nbdev_new --lib_name textkit --user yourusername --author "Your Name"
cd textkit
```

This creates:

```
textkit/
 nbs/
 00_core.ipynb # Your notebooks live here
 index.ipynb # Main module
 _quarto.yml # Becomes README and docs homepage
 textkit/ # Documentation config
 settings.ini # Generated Python package (don't edit directly)
 setup.py # Project configuration
 pyproject.toml # Generated for pip install
```

### 9.3.3 Key Insight: You Edit Notebooks, Not `.py` Files

The `textkit/` directory contains generated code. Your source of truth is `nbs/*.ipynb`.

## 9.4 2. Building the Core Module

### 9.4.1 The First Notebook: `00_core.ipynb`

Open `nbs/00_core.ipynb` in Jupyter. The structure:

```
Cell 1: Module header
#| default_exp core
```

This directive tells nbdev: “export cells from this notebook to `textkit/core.py`”.

## 9.4.2 Exporting Functions

```
#| export
def word_count(text: str) -> int:
 """Count words in text.

 Parameters

 text : str
 Input text to analyze

 Returns

 int
 Number of words

 Examples

 >>> word_count("Hello world")
 2
 >>> word_count("")
 0
 """
 if not text or not text.strip():
 return 0
 return len(text.split())
```

The `#| export` directive marks this cell for inclusion in the generated module.

## 9.4.3 Exploring as You Build

This is where notebooks shine. Between exported cells, add exploration:

```
Not exported - just exploration
sample_text = """
The quick brown fox jumps over the lazy dog.
This is a sample paragraph for testing our text analysis functions.
"""

print(f"Word count: {word_count(sample_text)}")
```

Your notebook becomes both implementation AND documentation of your thinking.

## 9.5 3. Adding Tests with nbdev

### 9.5.1 Inline Doctests

The docstring examples above ARE tests. nbdev runs them automatically:

```
nbdev_test
```

## 9.5.2 Dedicated Test Cells

For more complex tests:

```
#| test
def test_word_count_edge_cases():
 assert word_count("") == 0
 assert word_count(" ") == 0
 assert word_count("one") == 1
 assert word_count("one two three") == 3
 # Unicode handling
 assert word_count("café résumé") == 2
```

## 9.5.3 Running Tests

```
Run all tests
nbdev_test

Run tests for specific notebook
nbdev_test --path nbs/00_core.ipynb
```

# 9.6 4. Building More Functionality

## 9.6.1 Readability Scores

```
#| export
def flesch_reading_ease(text: str) -> float:
 """Calculate Flesch Reading Ease score.

 Scores typically range from 0-100:
 - 90-100: Very easy (5th grade)
 - 60-70: Standard (8th-9th grade)
 - 0-30: Very difficult (college graduate)

 Examples

 >>> score = flesch_reading_ease("The cat sat on the mat.")
 >>> 90 <= score <= 120 # Simple sentence = high score
 True
 """
 words = word_count(text)
 sentences = sentence_count(text)
 syllables = syllable_count(text)

 if words == 0 or sentences == 0:
 return 0.0

 return (
 206.835
 - 1.015 * (words / sentences)
```

```

 - 84.6 * (syllables / words)
)

```

## 9.6.2 Helper Functions

```

#! export
def sentence_count(text: str) -> int:
 """Count sentences in text.

 Examples

 >>> sentence_count("Hello. World!")
 2
 >>> sentence_count("No punctuation here")
 1
 """
 import re
 if not text.strip():
 return 0
 # Split on sentence-ending punctuation
 sentences = re.split(r'[.!?]+', text)
 # Filter empty strings
 return len([s for s in sentences if s.strip()])

```

```

#! export
def syllable_count(text: str) -> int:
 """Estimate syllable count (English approximation).

 Examples

 >>> syllable_count("hello")
 2
 >>> syllable_count("beautiful")
 4
 """
 import re
 text = text.lower()
 words = text.split()

 count = 0
 for word in words:
 word = re.sub(r'[^a-z]', '', word)
 if not word:
 continue
 # Simple heuristic: count vowel groups
 syllables = len(re.findall(r'[aeiouy]+', word))
 # Adjust for silent e
 if word.endswith('e') and syllables > 1:
 syllables -= 1
 count += max(1, syllables)

```

```
return count
```

## 9.7 5. Visualizations in Your Notebook

Notebooks excel at visual exploration. Add analysis cells (not exported):

```
Visualization - not exported, but shows in docs
import matplotlib.pyplot as plt

def visualize_readability(texts: dict[str, str]):
 """Compare readability across multiple texts."""
 names = list(texts.keys())
 scores = [flesch_reading_ease(t) for t in texts.values()]

 plt.figure(figsize=(10, 5))
 plt.barh(names, scores, color='steelblue')
 plt.xlabel('Flesch Reading Ease Score')
 plt.title('Readability Comparison')
 plt.axvline(x=60, color='red', linestyle='--', label='Standard
difficulty')
 plt.legend()
 plt.tight_layout()
 plt.show()

Demo with sample texts
samples = {
 "Children's book": "The cat sat. The dog ran. They played.",
 "News article": "The committee announced sweeping regulatory changes
affecting multiple industries.",
 "Academic paper": "The epistemological ramifications of quantum
indeterminacy necessitate reconceptualization.",
}

visualize_readability(samples)
```

This visualization appears in your generated documentation—showing users what the library can do.

## 9.8 6. Building the Text Analyzer Class

For a more complete API, add a class that combines functionality:

```
#! export
class TextAnalyzer:
 """Analyze text with multiple metrics.

 Examples

 >>> analyzer = TextAnalyzer("Hello world. How are you?")
```

```

>>> analyzer.word_count
5
>>> analyzer.sentence_count
2
"""

def __init__(self, text: str):
 self.text = text
 self._word_count = None
 self._sentence_count = None

@property
def word_count(self) -> int:
 if self._word_count is None:
 self._word_count = word_count(self.text)
 return self._word_count

@property
def sentence_count(self) -> int:
 if self._sentence_count is None:
 self._sentence_count = sentence_count(self.text)
 return self._sentence_count

@property
def avg_words_per_sentence(self) -> float:
 if self.sentence_count == 0:
 return 0.0
 return self.word_count / self.sentence_count

@property
def readability(self) -> float:
 return flesch_reading_ease(self.text)

def summary(self) -> dict:
 """Return all metrics as a dictionary."""
 return {
 "words": self.word_count,
 "sentences": self.sentence_count,
 "avg_words_per_sentence": round(self.avg_words_per_sentence,
1),
 "flesch_reading_ease": round(self.readability, 1),
 }

```

## 9.9 7. Adding an Interactive Widget

End with something users can interact with—demonstrating the notebook as an application:

```

Interactive demo (not exported - for notebook/docs only)
import ipywidgets as widgets
from IPython.display import display

```

```

def create_analyzer_widget():
 """Create an interactive text analyzer."""

 text_input = widgets.Textarea(
 value='Enter your text here...',
 placeholder='Paste text to analyze',
 description='Text:',
 layout=widgets.Layout(width='100%', height='150px')
)

 output = widgets.Output()

 def analyze(change):
 output.clear_output()
 with output:
 if text_input.value.strip():
 analyzer = TextAnalyzer(text_input.value)
 results = analyzer.summary()
 print(" Analysis Results")
 print("-" * 30)
 for key, value in results.items():
 print(f"{key.replace('_', ' ').title()}: {value}")

 text_input.observe(analyze, names='value')

 display(widgets.VBox([
 widgets.HTML("<h3> Text Analyzer</h3>"),
 text_input,
 output
]))

Show the widget
create_analyzer_widget()

```

When viewed in Colab or Binder, users can interact with your library without installing anything.

## 9.10 8. Generating the Package

### 9.10.1 Export to Python Modules

```
nbdev_export
```

This generates `textkit/core.py` from your notebook's `#| export` cells.

### 9.10.2 Verify Everything Works

```
Run tests
nbdev_test
```

```
Check for issues
nbdev_clean
nbdev_prepare
```

### 9.10.3 The Generated Code

Look at `textkit/core.py`—it contains clean Python code generated from your notebooks, with proper imports and structure.

## 9.11 9. Documentation

### 9.11.1 The Index Notebook

`nbs/index.ipynb` becomes both your `README.md` and documentation homepage. Include:

1. Installation instructions
2. Quick start example
3. Feature overview

```
In nbs/index.ipynb

TextKit

> Simple text analysis for Python

Installation

```bash
pip install textkit
```

9.12 Quick Start

```
from textkit.core import TextAnalyzer

text = "Your text here. Analyze it easily."
analyzer = TextAnalyzer(text)
print(analyzer.summary())
```

```
### Build Documentation

```bash
nbdev_docs
```

This generates a Quarto-based documentation site in `_docs/`.

## 9.13 10. Publishing to PyPI

### 9.13.1 Prepare for Release

```
Clean and prepare
nbdev_prepare

Build distribution
python -m build
```

### 9.13.2 Publish

```
Test PyPI first
twine upload --repository testpypi dist/*

Then real PyPI
twine upload dist/*
```

### 9.13.3 The Result

```
pip install textkit
```

You’ve shipped a Python package—developed entirely in notebooks.

## 9.14 11. Sharing the Notebook Itself

Beyond the package, share the development notebook:

### 9.14.1 Colab Badge

```
[![Open In Colab](BADGE)](COLAB_LINK)

Replace BADGE and COLAB_LINK with your repo URLs.
```

### 9.14.2 Binder Badge

```
[![Binder](https://mybinder.org/badge_logo.svg)](https://mybinder.org/v2/gh/username/textkit/main)
```

Users can:

1. **Install the package** via pip (traditional)
2. **Explore the notebook** to understand the code (educational)
3. **Run interactively** in Colab/Binder (zero-install)

## 9.15 Comparing Workflows

Here’s how this case study compares to the SimpleBot approach (Chapter 4):

Aspect	SimpleBot (Scripts)	TextKit (nbdev)
Source files	.py in <code>src/</code>	.ipynb in <code>nbs/</code>
Tests	Separate <code>tests/</code> directory	Inline with code
Documentation	Separate <code>docs/</code>	Generated from notebooks
Exploration	Separate REPL/scratch files	Integrated in notebooks
Output	Package on PyPI	Package on PyPI
Best for	Traditional dev, teams	Exploratory, teaching

Both workflows produce the same result: a published package. Choose based on how you like to work.

## 9.16 When to Use This Workflow

The nbdev approach works best when:

- **Exploration is central:** You're figuring things out as you build
- **Teaching matters:** Others will learn from your notebooks
- **Docs should show execution:** You want live examples in documentation
- **Solo or small team:** Git conflicts in notebooks are real

Consider traditional scripts when:

- **Large teams:** Notebook diffs are harder to review
- **Complex architecture:** Many interconnected modules
- **Heavy IDE reliance:** Refactoring tools work better with .py files
- **Existing codebase:** Converting to nbdev is non-trivial

## 9.17 Summary

- **nbdev inverts the workflow:** Notebooks are source, .py files are generated
- **Tests live with code:** Doctests and `#! test` cells eliminate context switching
- **Exploration becomes documentation:** Your investigative work helps users
- **Same destination:** Published package, installable via pip
- **Different journey:** Iterative, visual, integrated

## 9.18 Exercises

1. **Extend TextKit:** Add a `sentiment_words()` function that counts positive/negative words from a simple word list. Include doctests.
2. **Add a notebook:** Create `01_advanced.ipynb` with functions for text comparison (e.g., similarity between two texts).
3. **Publish to TestPyPI:** Go through the full publication workflow to TestPyPI.
4. **Create a Voilà dashboard:** Convert the interactive widget section into a standalone Voilà dashboard.
5. **Compare workflows:** Take one function from TextKit and rewrite it in the traditional script workflow. Reflect on the differences.



# Chapter 10

## Conclusion: Embracing Efficient Python Development

Throughout this guide, we've built a comprehensive Python development pipeline that balances simplicity with professional practices. From project structure to deployment, we've covered tools and techniques that help create maintainable, reliable, and efficient Python code.

### 10.1 The Power of a Complete Pipeline

Each component of our development workflow serves a specific purpose:

- **Project structure** provides organisation and clarity
- **Version control** enables collaboration and change tracking
- **Virtual environments** isolate dependencies
- **Dependency management** ensures reproducible environments
- **Code formatting and linting** maintain consistent, error-free code
- **Testing** verifies functionality
- **Type checking** catches type errors early
- **Security scanning** prevents vulnerabilities
- **Dead code detection** keeps projects lean
- **Documentation** makes code accessible to others
- **CI/CD** automates quality checks and deployment
- **Package publishing** shares your work with the world

Together, these practices create a development experience that is both efficient and enjoyable. You spend less time on repetitive tasks and more time solving the real problems your code addresses.

### 10.2 Your Path Forward: A Practical Adoption Strategy

The concepts in this book are most valuable when applied systematically. Here's a concrete roadmap for implementing these practices, tailored to different project stages and team sizes:

### 10.2.1 For Your Next New Project (Week 1)

**Immediate implementation** - Use these from day one: 1. **Project structure**: Start with the `src` layout and proper directory organisation 2. **Version control**: Initialize Git immediately with a proper `.gitignore` 3. **Virtual environment**: Use `uv` or `pip-tools` for dependency management 4. **Basic automation**: Set up Poe the Poet with essential tasks (`lint`, `test`, `format`)

```
Your starting checklist - 15 minutes to professional setup
uv init my-project --package
cd my-project
Copy your preferred pyproject.toml template
uv add --dev pytest ruff mypy poethepoet pre-commit
uv run pre-commit install
git init && git add . && git commit -m "Initial project setup"
```

### 10.2.2 For Existing Projects (Month 1-2)

**Gradual integration** - Add one practice per week: - **Week 1**: Add code formatting with Ruff (`uv run ruff format .`) - **Week 2**: Introduce basic testing with `pytest` - **Week 3**: Add pre-commit hooks for automated quality checks - **Week 4**: Set up task automation with Poe the Poet - **Week 5**: Add type checking with `mypy` - **Week 6**: Implement basic CI/CD with GitHub Actions

This pace prevents workflow disruption while building better practices.

### 10.2.3 For Team Environments (Month 2-3)

**Collaborative workflows** - Focus on consistency and shared practices: - **Documentation standards**: Establish README templates and docstring conventions - **Code review processes**: Define what automated checks must pass before review - **Shared configurations**: centralise tool configuration in `pyproject.toml` - **Development environment parity**: Use containers or detailed setup documentation

### 10.2.4 Advanced Techniques (Month 3+)

Only after mastering the fundamentals: - **Performance optimisation**: When benchmarks indicate actual problems - **Advanced architecture**: When code complexity impedes development - **Containerization**: When environment consistency becomes problematic

## 10.3 AI as Your Development Partner

Throughout this book, we have built a professional Python development pipeline. AI amplifies every part of it:

- **Project scaffolding**: AI generates directory structures, configuration files, and boilerplate
- **Code quality**: AI explains linting errors, suggests type annotations, and writes docstrings
- **Testing**: AI generates test cases, including edge cases you might miss
- **Documentation**: AI drafts READMEs, API docs, and contributing guides

- **CI/CD:** AI generates GitHub Actions workflows and Dockerfile configurations
- **Debugging:** AI analyses error messages and suggests fixes

The critical insight is that AI works best when you understand what it produces. The pipeline practices in this book give you that understanding. Without knowing what a good `pyproject.toml` looks like, you cannot evaluate AI-generated configuration. Without understanding `pytest` fixtures, you cannot judge AI-generated tests. The fundamentals make AI useful rather than dangerous.

### ! Orchestrate, Don't Delegate

The title of this book is deliberate. *Orchestrate* AI means you direct the work, review the output, and make the decisions. AI generates code faster than you can type it — but you still need to know whether that code is correct, secure, and maintainable. The pipeline you have built is your quality gate. Everything AI produces passes through it: linted by Ruff, checked by mypy, tested by pytest, reviewed by you.

## 10.4 Beyond Tools: Engineering Culture

The most important outcome isn't just using specific tools—it's developing habits and values that lead to better software:

- **Think defensively:** Use tools that catch mistakes early
- **Value maintainability:** Write code for humans, not just computers
- **Embrace automation:** Let computers handle repetitive tasks
- **Practice continuous improvement:** Regularly refine your workflow
- **Share knowledge:** Document not just what code does, but why

## 10.5 When to Consider More Advanced Tools

As your projects grow more complex, you might explore more sophisticated tools:

- **Containerization** with Docker for consistent environments
- **Orchestration** with Kubernetes for complex deployments
- **Monorepo tools** like Pants or Bazel for large codebases
- **Feature flagging** for controlled feature rollouts
- **Advanced monitoring** for production insights

However, the core practices we've covered will remain valuable regardless of the scale you reach.

## 10.6 Common Implementation Challenges and Solutions

As you implement these practices, you'll likely encounter some common obstacles. Here's how to address them:

### 10.6.1 “This Seems Like Too Much Overhead”

**Symptom:** Tools feel burdensome and slow down development **Solution:** Start smaller and focus on automation - Begin with just `ruff` `format` and `pytest` - Use pre-commit hooks to make quality checks automatic - Remember: 5 minutes of setup saves hours of debugging later

### 10.6.2 “My Team Resists New Processes”

**Symptom:** Team members bypass or ignore new practices **Solution:** Lead by example and demonstrate value - Start with your own projects and show improved outcomes - Introduce practices that solve existing pain points - Make adherence easy with good tooling and clear documentation

### 10.6.3 “Tool Configuration is Confusing”

**Symptom:** Conflicting configurations or unclear settings **Solution:** Use our recommended starting templates - Copy configuration from successful projects - Use the companion templates to bootstrap correctly - Focus on standard configurations before customising

### 10.6.4 “I Don’t Know When to Add Advanced Practices”

**Symptom:** Uncertainty about when complexity is justified **Solution:** Let pain points guide your decisions - Add testing when manual verification becomes tedious - Add CI/CD when manual releases cause errors - Add advanced architecture when code becomes hard to maintain - Never add complexity that doesn’t solve an actual problem

## 10.7 Staying Updated and Growing

Python’s ecosystem continues to evolve. Maintain relevance by:

### 10.7.1 Following Core Development Principles

- **Python Enhancement Proposals (PEPs):** Understand the direction of the language
- **Community discussions:** Participate in forums like Python Discourse or [Reddit r/Python](https://www.reddit.com/r/Python)
- **Release notes:** Read updates for your core dependencies (`pytest`, `ruff`, `uv`, etc.)

### 10.7.2 Practical Learning Approach

- **Test new tools in small projects** before adopting them in production
- **Attend conferences or meetups** (virtual or in-person) for broader perspective
- **Read other people’s code** to see different implementation approaches
- **Contribute to open source** to deepen understanding of development practices

### 10.7.3 Continuous Improvement Mindset

- **Regular retrospectives:** What’s working well? What’s causing friction?
- **Experiment with alternatives:** Try new tools when they solve specific problems
- **Share knowledge:** Write about your experiences and learn from feedback

## 10.8 Final Thoughts

This book represents more than a collection of Python tools—it’s a philosophy of development that prioritizes sustainability, maintainability, and developer happiness. The practices we’ve explored create a foundation that serves projects from first prototype to production scale.

### 10.8.1 The Universal Principles Behind the Tools

While we’ve used Python tooling as our examples, the core concepts transfer across languages and domains:

- **Clear project structure** reduces cognitive load in any language
- **Automated quality checks** catch errors early regardless of the technology stack
- **Comprehensive testing** provides confidence when making changes
- **Thoughtful automation** eliminates repetitive work and reduces human error
- **Progressive complexity** allows practices to evolve with project needs

These principles remain constant even as specific tools evolve.

### 10.8.2 Your Development Journey Continues

The practices in this book form a foundation, not a destination. As you apply these concepts:

- **Trust the process:** Initially, some practices may feel like overhead, but their value becomes clear as projects grow
- **Adapt to your context:** Not every practice fits every project, but understanding the principles helps you make informed decisions
- **Share your knowledge:** Teaching others these practices deepens your own understanding and improves the broader development community

### 10.8.3 Your Pipeline Is Agent-Ready Infrastructure

AI coding agents, tools like Cursor, Claude Code, and OpenHands, are getting remarkably capable. They can generate code, run tests, fix errors, and iterate. But they work best when they have structure to work within.

Everything in this book builds that structure. A test suite gives an agent a way to verify its own output. A CI pipeline catches mistakes before they reach users. A clear project layout gives an agent context about where code belongs. Pre-commit hooks enforce quality standards automatically.

Without this infrastructure, an agent is generating code into a void. With it, an agent has guardrails, feedback loops, and a deployment path. The pipeline you have built is what turns AI from a code generator into a reliable development partner. That is what orchestration means.

### 10.8.4 Starting Your Next Project

You now have everything needed to begin any Python project with professional practices from day one. Whether you use our bash script for transparency, GitHub templates

for convenience, or cookiecutter templates for customisation, you can establish solid foundations in minutes rather than hours.

More importantly, you understand **why** these practices matter and **when** to apply them. This knowledge will serve you well as you encounter new challenges and evaluate new tools.

### 10.8.5 A Personal Note

Remember that perfect is the enemy of good. Start with the basics, improve incrementally, and focus on delivering value through your code. The best development pipeline is one that you'll actually use consistently.

The Python ecosystem will continue evolving—new tools will emerge, and current tools will improve—but the underlying principles of clear structure, automated quality, comprehensive testing, and thoughtful automation will remain valuable throughout your development career.

We hope this guide helps you build software that not only works but is also maintainable, reliable, and enjoyable to develop. The investment you make in better development practices pays dividends for years to come.

Happy coding, and may your development pipeline serve you well!

# Acknowledgments

This book started as a personal quest: find a sane Python development workflow that could target multiple platforms without drowning in complexity. The practices in these pages were tested on real projects before they were written up, and the ones that survived are the ones that actually worked under pressure.

Students who used earlier versions of these workflows — in project templates, lab exercises, and assignment scaffolding — provided the practical feedback that separated good ideas from impractical ones. When a workflow looked elegant in documentation but caused confusion in practice, they said so.

The Python community deserves particular credit. The developers behind `uv`, `ruff`, `mypy`, `pytest`, and the broader ecosystem of tools covered in this book have made professional Python development accessible in ways that were not possible even a few years ago. Their work is the foundation this book builds on.

The open source community behind Quarto, GitHub, and GitHub Pages made it possible to write, build, and publish across multiple formats. The cookiecutter and template ecosystems made it possible to turn the book's recommendations into something readers can use immediately.

AI tools were used throughout the writing process. Claude (Anthropic) served as a conversation partner for drafting, iterating, and refining. The process was the same one the book advocates: orchestrate the AI, but own the decisions. Every recommendation, every tool choice, every architectural opinion reflects the author's judgement. The AI made the work faster. It did not make the decisions.



# About the Author



**Michael Borck** is a software developer and educator passionate about the intersection of human expertise and artificial intelligence. He developed the Intentional Prompting methodology to help programmers maintain agency and deepen their understanding while leveraging AI tools effectively.

Michael believes that the future of programming lies not in delegating to AI, but in conversing with it—treating AI as a collaborative partner that enhances human capability rather than replacing human understanding.

When not writing about AI collaboration, Michael works on practical applications of these principles across software development, education, and creative projects. He creates educational software and resources, and explores the 80/20 principle in learning and productivity.

---

## Connect

- [michaelborck.dev](https://michaelborck.dev) — Professional work and projects
- [michaelborck.education](https://michaelborck.education) — Educational software and resources
- [8020workshop.com](https://8020workshop.com) — Passion projects and workshops
- [LinkedIn](#)

---

## Other Books in This Series

*Foundational Methodology:*

- *Conversation, Not Delegation: Your Expertise + AI's Breadth = Amplified Thinking*
- *Converse Python, Partner AI: The Python Edition*

*Python Track:*

- *Think Python, Direct AI: Computational Thinking for Beginners*
- *Code Python, Consult AI: Python Fundamentals for the AI Era*

*Web Track:*

- *Build Web, Guide AI: Business Web Development with AI*

*For Educators:*

- *Partner, Don't Police: AI in the Business Classroom*

# Appendix A

## Glossary of Python Development Terms

### A.1 A

- **API (Application Programming Interface):** A set of definitions and protocols for building and integrating application software.
- **Artifact:** Any file or package produced during the software development process, such as documentation or distribution packages.

### A.2 C

- **CI/CD (Continuous Integration/Continuous Deployment):** Practices where code changes are automatically tested (CI) and deployed to production (CD) when they pass quality checks.
- **CLI (Command Line Interface):** A text-based interface for interacting with software using commands.
- **Code Coverage:** A measure of how much of your code is executed during testing.
- **Code Linting:** The process of analysing code for potential errors, style issues, and suspicious constructs.

### A.3 D

- **Dependency:** An external package or module that your project requires to function properly.
- **Docstring:** A string literal specified in source code that is used to document a specific segment of code.
- **Dynamic Typing:** A programming language feature where variable types are checked during runtime rather than compile time.
- – **Cookiecutter:** A project templating tool that helps developers create new projects with a predefined structure, configuration files, and boilerplate code. Cookiecutter uses Jinja2 templating to customise files based on user inputs during project generation.

## A.4 E

- **Entry Point:** A function or method that serves as an access point to an application, module, or library.

## A.5 F

- **Fixture:** In testing, a piece of code that sets up a system for testing and provides test data.

## A.6 G

- **Git:** A distributed version control system for tracking changes in source code.
- **GitHub Repository Template:** A repository that can be used as a starting point for new projects on GitHub.
- **GitHub/GitLab:** Web-based platforms for hosting Git repositories with collaboration features.

## A.7 I

- **Integration Testing:** Testing how different parts of the system work together.

## A.8 L

- **Lock File:** A file that records the exact versions of dependencies needed by a project to ensure reproducible installations.

## A.9 M

- **Mocking:** Simulating the behaviour of real objects in controlled ways during testing.
- **Module:** A file containing Python code that can be imported and used by other Python files.
- **Monorepo:** A software development strategy where many projects are stored in the same repository.

## A.10 N

- **Namespace Package:** A package split across multiple directories or distribution packages.

## A.11 P

- **Package:** A directory of Python modules containing an additional `__init__.py` file.
- **PEP (Python Enhancement Proposal):** A design document providing information to the Python community, often proposing new features.

- **PEP 8:** The style guide for Python code.
- **PyPI (Python Package Index):** The official repository for third-party Python software.

## A.12 R

- **Refactoring:** Restructuring existing code without changing its external behaviour.
- **Repository:** A storage location for software packages and version control.
- **Requirements File:** A file listing the dependencies required for a Python project.
- **Reproducible Build:** A build that can be recreated exactly regardless of when or where it's built.

## A.13 S

- **Semantic Versioning:** A versioning scheme in the format MAJOR.MINOR.PATCH, where each number increment indicates the type of change.
- **Static Analysis:** analysing code without executing it to find potential issues.
- **Static Typing:** Specifying variable types at compile time instead of runtime.
- **Stub Files:** Files that contain type annotations for modules that don't have native typing support.

## A.14 T

- **Test-Driven Development (TDD):** A development process where tests are written before the code.
- **Type Annotation:** Syntax for indicating the expected type of variables, function parameters, and return values.
- **Type Hinting:** Adding type annotations to Python code to help with static analysis and IDE assistance.

## A.15 U

- **Unit Testing:** Testing individual components in isolation from the rest of the system.

## A.16 V

- **Virtual Environment:** An isolated Python environment that allows packages to be installed for use by a particular project, without affecting other projects.

## A.17 W

- **Wheel:** A built-package format for Python that can be installed more quickly than source distributions.



# Appendix B

## Distribution Templates

This appendix provides ready-to-use templates and configurations for multi-platform distribution. These templates complement the architecture patterns discussed in Chapter 8: Multi-Platform Distribution.

### B.1 Companion Repositories

To quickly start a new project with this architecture, use one of our companion repositories:

#### B.1.1 Cookiecutter Template

For full customisation with interactive prompts:

```
Install cookiecutter
pip install cookiecutter

Create a new project
cookiecutter gh:michael-borck/ship-python-cookiecutter
```

The cookiecutter template prompts for:

- Project name and description
- Author information
- Backend options (database choice, authentication)
- Frontend options (styling framework, state management)
- Distribution targets (web, PWA, desktop)
- CI/CD configuration

#### B.1.2 GitHub Template

For quick starts without installing tools:

1. Visit <https://github.com/michael-borck/ship-python-template>
2. Click “Use this template”
3. Clone your new repository and customise

### B.1.3 Example Repository

A complete working example with all features:

- <https://github.com/michael-borck/ship-python-example>

## B.2 FastAPI Backend Template

### B.2.1 pyproject.toml

```
[project]
name = "my-app-backend"
version = "0.1.0"
description = "Backend API for My Application"
readme = "README.md"
requires-python = ">=3.11"
dependencies = [
 "fastapi>=0.109.0",
 "uvicorn[standard]>=0.27.0",
 "pydantic>=2.5.0",
 "pydantic-settings>=2.1.0",
 "sqlalchemy>=2.0.0",
 "alembic>=1.13.0",
 "python-multipart>=0.0.6",
]

[project.optional-dependencies]
dev = [
 "pytest>=7.4.0",
 "pytest-cov>=4.1.0",
 "pytest-asyncio>=0.23.0",
 "httpx>=0.26.0",
 "ruff>=0.1.0",
 "mypy>=1.8.0",
]

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[tool.hatch.build.targets.wheel]
packages = ["src/my_app"]

[tool.ruff]
target-version = "py311"
line-length = 88

[tool.ruff.lint]
select = ["E", "F", "I", "B", "UP"]

[tool.mypy]
python_version = "3.11"
```

```

strict = true

[tool.pytest.ini_options]
testpaths = ["tests"]
asyncio_mode = "auto"

[tool.poe.tasks]
dev = "uvicorn my_app.main:app --reload"
test = "pytest --cov=my_app"
lint = "ruff check ."
format = "ruff format ."
typecheck = "mypy src/"
check = ["format", "lint", "typecheck", "test"]

```

## B.2.2 Application Configuration

```

src/my_app/config.py
"""Application configuration using pydantic-settings."""

from pydantic_settings import BaseSettings
from functools import lru_cache

class Settings(BaseSettings):
 """Application settings loaded from environment variables."""

 # Application
 app_name: str = "My Application"
 debug: bool = False
 version: str = "0.1.0"

 # Server
 host: str = "0.0.0.0"
 port: int = 8000

 # Database
 database_url: str = "sqlite:///./app.db"

 # CORS
 cors_origins: list[str] = ["http://localhost:5173"]

 # Security
 secret_key: str = "change-me-in-production"

 class Config:
 env_file = ".env"
 env_file_encoding = "utf-8"

@lru_cache

```

```
def get_settings() -> Settings:
 """Get cached settings instance."""
 return Settings()
```

### B.2.3 Database Setup with SQLAlchemy

```
src/my_app/database.py
"""Database configuration and session management."""

from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, DeclarativeBase
from contextlib import contextmanager
from typing import Generator

from .config import get_settings

class Base(DeclarativeBase):
 """Base class for SQLAlchemy models."""
 pass

engine = create_engine(
 get_settings().database_url,
 connect_args={"check_same_thread": False} # For SQLite
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)

def get_db() -> Generator:
 """Dependency for getting database sessions."""
 db = SessionLocal()
 try:
 yield db
 finally:
 db.close()

@contextmanager
def get_db_context():
 """Context manager for database sessions outside of FastAPI."""
 db = SessionLocal()
 try:
 yield db
 db.commit()
 except Exception:
 db.rollback()
 raise
```

```

 finally:
 db.close()

def init_db():
 """Initialize database tables."""
 Base.metadata.create_all(bind=engine)

```

## B.2.4 Backend Dockerfile

```

backend/Dockerfile
FROM python:3.11-slim

WORKDIR /app

Install uv
RUN pip install uv

Copy dependency files
COPY pyproject.toml uv.lock ./

Install dependencies
RUN uv sync --frozen --no-dev

Copy source code
COPY src ./src

Create non-root user
RUN useradd -m appuser && chown -R appuser:appuser /app
USER appuser

Expose port
EXPOSE 8000

Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
 CMD python -c "import urllib.request;
 urllib.request.urlopen('http://localhost:8000/health')"

Run application
CMD ["uv", "run", "uvicorn", "my_app.main:app", "--host", "0.0.0.0",
 "--port", "8000"]

```

## B.3 React Frontend Template

### B.3.1 package.json

```

{
 "name": "my-app-frontend",

```

```

"private": true,
"version": "0.1.0",
"type": "module",
"scripts": {
 "dev": "vite",
 "build": "tsc && vite build",
 "preview": "vite preview",
 "test": "vitest",
 "test:coverage": "vitest run --coverage",
 "lint": "eslint . --ext ts,tsx --report-unused-disable-directives
--max-warnings 0",
 "typecheck": "tsc --noEmit"
},
"dependencies": {
 "@tanstack/react-query": "^5.17.0",
 "axios": "^1.6.0",
 "react": "^18.2.0",
 "react-dom": "^18.2.0",
 "react-router-dom": "^6.21.0"
},
"devDependencies": {
 "@types/react": "^18.2.0",
 "@types/react-dom": "^18.2.0",
 "@typescript-eslint/eslint-plugin": "^6.0.0",
 "@typescript-eslint/parser": "^6.0.0",
 "@vitejs/plugin-react": "^4.2.0",
 "eslint": "^8.56.0",
 "eslint-plugin-react-hooks": "^4.6.0",
 "eslint-plugin-react-refresh": "^0.4.0",
 "typescript": "^5.3.0",
 "vite": "^5.0.0",
 "vite-plugin-pwa": "^0.17.0",
 "vitest": "^1.2.0"
}
}

```

### B.3.2 Vite Configuration with PWA

```

// frontend/vite.config.ts
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
import { VitePWA } from 'vite-plugin-pwa';

export default defineConfig({
 plugins: [
 react(),
 VitePWA({
 registerType: 'autoUpdate',
 includeAssets: ['favicon.ico', 'apple-touch-icon.png',
'mask-icon.svg'],

```

```
manifest: {
 name: 'My Application',
 short_name: 'MyApp',
 description: 'A multi-platform application',
 theme_color: '#ffffff',
 background_color: '#ffffff',
 display: 'standalone',
 scope: '/',
 start_url: '/',
 icons: [
 {
 src: 'pwa-192x192.png',
 sizes: '192x192',
 type: 'image/png',
 },
 {
 src: 'pwa-512x512.png',
 sizes: '512x512',
 type: 'image/png',
 },
 {
 src: 'pwa-512x512.png',
 sizes: '512x512',
 type: 'image/png',
 purpose: 'any maskable',
 },
],
},
devOptions: {
 enabled: true,
},
workbox: {
 globPatterns: ['**/*.{js,css,html,ico,png,svg,woff2}'],
 runtimeCaching: [
 {
 urlPattern: /^https:\/\/\api\..*\/i,
 handler: 'NetworkFirst',
 options: {
 cacheName: 'api-cache',
 expiration: {
 maxEntries: 100,
 maxAgeSeconds: 60 * 60 * 24, // 24 hours
 },
 cacheableResponse: {
 statuses: [0, 200],
 },
 },
 },
],
},
},
}),
```

```

],
server: {
 port: 5173,
 proxy: {
 '/api': {
 target: 'http://localhost:8000',
 changeOrigin: true,
 },
 },
},
build: {
 outDir: 'dist',
 sourcemap: true,
},
});

```

```

// frontend/tsconfig.json
{
 "compilerOptions": {
 "target": "ES2020",
 "useDefineForClassFields": true,
 "lib": ["ES2020", "DOM", "DOM.Iterable"],
 "module": "ESNext",
 "skipLibCheck": true,

 "moduleResolution": "bundler",
 "allowImportingTsExtensions": true,
 "resolveJsonModule": true,
 "isolatedModules": true,
 "noEmit": true,
 "jsx": "react-jsx",

 "strict": true,
 "noUnusedLocals": true,
 "noUnusedParameters": true,
 "noFallthroughCasesInSwitch": true,

 "baseUrl": ".",
 "paths": {
 "@/*": ["src/*"]
 }
 },
 "include": ["src"],
 "references": [{ "path": "./tsconfig.node.json" }]
}

```

### B.3.4 API Client Template

```
// frontend/src/services/api.ts
import axios, { AxiosInstance, AxiosError } from 'axios';

const API_BASE_URL = import.meta.env.VITE_API_URL || '/api';

class ApiClient {
 private client: AxiosInstance;

 constructor() {
 this.client = axios.create({
 baseURL: API_BASE_URL,
 headers: {
 'Content-Type': 'application/json',
 },
 });
 }

 // Response interceptor for error handling
 this.client.interceptors.response.use(
 (response) => response,
 (error: AxiosError) => {
 if (error.response?.status === 401) {
 // Handle unauthorized
 window.location.href = '/login';
 }
 return Promise.reject(error);
 }
);
}

async get<T>(url: string): Promise<T> {
 const response = await this.client.get<T>(url);
 return response.data;
}

async post<T>(url: string, data: unknown): Promise<T> {
 const response = await this.client.post<T>(url, data);
 return response.data;
}

async put<T>(url: string, data: unknown): Promise<T> {
 const response = await this.client.put<T>(url, data);
 return response.data;
}

async delete(url: string): Promise<void> {
 await this.client.delete(url);
}

setAuthToken(token: string): void {
 this.client.defaults.headers.common['Authorization'] = `Bearer ${token}`;
}
```

```

}

clearAuthToken(): void {
 delete this.client.defaults.headers.common['Authorization'];
}
}

export const apiClient = new ApiClient();

```

## B.4 Electron Wrapper Template

### B.4.1 Main Process

```

// electron/main.js
const { app, BrowserWindow, Menu, shell, ipcMain } =
require('electron');
const path = require('path');
const { autoUpdater } = require('electron-updater');
const log = require('electron-log');

// Configure logging
log.transports.file.level = 'info';
autoUpdater.logger = log;

// Handle Squirrel events for Windows
if (require('electron-squirrel-startup')) {
 app.quit();
}

const isDev = process.env.NODE_ENV === 'development';
let mainWindow;

function createWindow() {
 mainWindow = new BrowserWindow({
 width: 1200,
 height: 800,
 minWidth: 800,
 minHeight: 600,
 webPreferences: {
 preload: path.join(__dirname, 'preload.js'),
 contextIsolation: true,
 nodeIntegration: false,
 sandbox: true,
 },
 show: false, // Show when ready
 titleBarStyle: 'hiddenInset', // macOS
 });

 // Show window when ready
 mainWindow.once('ready-to-show', () => {

```

```
 mainWindow.show();
 });

 // Load content
 if (isDev) {
 mainWindow.loadURL('http://localhost:5173');
 mainWindow.webContents.openDevTools();
 } else {
 mainWindow.loadFile(path.join(__dirname,
 './frontend/dist/index.html'));
 }

 // Open external links in browser
 mainWindow.webContents.setWindowOpenHandler(({ url }) => {
 shell.openExternal(url);
 return { action: 'deny' };
 });

 // Build menu
 const template = [
 {
 label: 'File',
 submenu: [
 { role: 'quit' }
]
 },
 {
 label: 'Edit',
 submenu: [
 { role: 'undo' },
 { role: 'redo' },
 { type: 'separator' },
 { role: 'cut' },
 { role: 'copy' },
 { role: 'paste' },
 { role: 'selectAll' }
]
 },
 {
 label: 'View',
 submenu: [
 { role: 'reload' },
 { role: 'forceReload' },
 { role: 'toggleDevTools' },
 { type: 'separator' },
 { role: 'resetZoom' },
 { role: 'zoomIn' },
 { role: 'zoomOut' },
 { type: 'separator' },
 { role: 'togglefullscreen' }
]
 }
]
```

```

 },
 {
 label: 'Help',
 submenu: [
 {
 label: 'Documentation',
 click: async () => {
 await shell.openExternal('https://example.com/docs');
 }
 },
 {
 label: 'Check for Updates',
 click: () => {
 autoUpdater.checkForUpdatesAndNotify();
 }
 }
]
 }
]
};

const menu = Menu.buildFromTemplate(template);
Menu.setApplicationMenu(menu);
}

// App lifecycle
app.whenReady().then(() => {
 createWindow();

 // Check for updates (production only)
 if (!isDev) {
 autoUpdater.checkForUpdatesAndNotify();
 }

 app.on('activate', () => {
 if (BrowserWindow.getAllWindows().length === 0) {
 createWindow();
 }
 });
});

app.on('window-all-closed', () => {
 if (process.platform !== 'darwin') {
 app.quit();
 }
});

// IPC handlers
ipcMain.handle('get-version', () => {
 return app.getVersion();
});

```

```

ipcMain.handle('check-for-updates', () => {
 return autoUpdater.checkForUpdatesAndNotify();
});

// Auto-updater events
autoUpdater.on('checking-for-update', () => {
 log.info('Checking for update...');
});

autoUpdater.on('update-available', (info) => {
 log.info('Update available:', info);
});

autoUpdater.on('update-not-available', (info) => {
 log.info('Update not available:', info);
});

autoUpdater.on('error', (err) => {
 log.error('Error in auto-updater:', err);
});

autoUpdater.on('download-progress', (progressObj) => {
 log.info(`Download speed: ${progressObj.bytesPerSecond} - Downloaded
 ${progressObj.percent}%`);
});

autoUpdater.on('update-downloaded', (info) => {
 log.info('Update downloaded:', info);
 // Optionally prompt user and restart
});

```

## B.4.2 Preload Script

```

// electron/preload.js
const { contextBridge, ipcRenderer } = require('electron');

contextBridge.exposeInMainWorld('electronAPI', {
 // App info
 getVersion: () => ipcRenderer.invoke('get-version'),
 platform: process.platform,

 // Updates
 checkForUpdates: () => ipcRenderer.invoke('check-for-updates'),

 // Window controls (if using frameless window)
 minimise: () => ipcRenderer.send('window-minimise'),
 maximise: () => ipcRenderer.send('window-maximise'),
 close: () => ipcRenderer.send('window-close'),
});

```

### B.4.3 Electron Builder Configuration

```

{
 "name": "my-app",
 "version": "1.0.0",
 "description": "My Application",
 "main": "main.js",
 "author": "Your Name <you@example.com>",
 "license": "MIT",
 "scripts": {
 "start": "electron .",
 "build": "electron-builder",
 "build:all": "electron-builder -mwl",
 "build:mac": "electron-builder --mac",
 "build:win": "electron-builder --win",
 "build:linux": "electron-builder --linux",
 "release": "electron-builder --publish always"
 },
 "build": {
 "appId": "com.yourname.myapp",
 "productName": "My Application",
 "copyright": "Copyright © 2025 Your Name",
 "directories": {
 "output": "dist",
 "buildResources": "build"
 },
 "files": [
 "main.js",
 "preload.js",
 "../frontend/dist/**/*"
],
 "extraMetadata": {
 "main": "main.js"
 },
 "mac": {
 "category": "public.app-category.productivity",
 "icon": "build/icon.icns",
 "hardenedRuntime": true,
 "gatekeeperAssess": false,
 "entitlements": "build/entitlements.mac.plist",
 "entitlementsInherit": "build/entitlements.mac.plist",
 "target": [
 {
 "target": "dmg",
 "arch": ["x64", "arm64"]
 },
 {
 "target": "zip",
 "arch": ["x64", "arm64"]
 }
]
 }
 },
},

```

```
"win": {
 "icon": "build/icon.ico",
 "target": [
 {
 "target": "nsis",
 "arch": ["x64"]
 },
 {
 "target": "portable",
 "arch": ["x64"]
 }
]
},
"nsis": {
 "oneClick": false,
 "perMachine": false,
 "allowToChangeInstallationDirectory": true,
 "deleteAppDataOnUninstall": true
},
"linux": {
 "icon": "build/icons",
 "category": "Utility",
 "target": [
 {
 "target": "AppImage",
 "arch": ["x64"]
 },
 {
 "target": "deb",
 "arch": ["x64"]
 }
]
},
"publish": {
 "provider": "github",
 "owner": "your-username",
 "repo": "my-app",
 "releaseType": "release"
},
"dependencies": {
 "electron-log": "^5.0.0",
 "electron-updater": "^6.1.0"
},
"devDependencies": {
 "electron": "^28.0.0",
 "electron-builder": "^24.9.0"
}
}
```

## B.5 Docker Configuration

### B.5.1 Combined Dockerfile (Backend + Frontend)

```

docker/Dockerfile.combined
Build stage for frontend
FROM node:20-alpine AS frontend-builder

WORKDIR /app/frontend
COPY frontend/package*.json ./
RUN npm ci --production=false

COPY frontend/ ./
RUN npm run build

Build stage for backend dependencies
FROM python:3.11-slim AS backend-builder

WORKDIR /app
RUN pip install uv

COPY backend/pyproject.toml backend/uv.lock ./
RUN uv sync --frozen --no-dev

Production stage
FROM python:3.11-slim AS production

WORKDIR /app

Install uv for running
RUN pip install uv

Copy backend dependencies
COPY --from=backend-builder /app/.venv /app/.venv

Copy backend source
COPY backend/src ./src

Copy frontend build
COPY --from=frontend-builder /app/frontend/dist ./static

Create non-root user
RUN useradd -m -u 1000 appuser && \
 chown -R appuser:appuser /app
USER appuser

Environment
ENV PATH="/app/.venv/bin:$PATH"
ENV PYTHONPATH="/app/src"
ENV STATIC_FILES_DIR="/app/static"

EXPOSE 8000

```

```
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
 CMD python -c "import urllib.request;
 urllib.request.urlopen('http://localhost:8000/health')"
```

```
CMD ["uvicorn", "my_app.main:app", "--host", "0.0.0.0", "--port",
"8000"]
```

## B.5.2 Docker Compose for Development

```
docker/docker-compose.yml
version: "3.9"

services:
 backend:
 build:
 context: ../backend
 dockerfile: Dockerfile.dev
 ports:
 - "8000:8000"
 volumes:
 - ../backend/src:/app/src:ro
 environment:
 - DEBUG=true
 - DATABASE_URL=sqlite:///./dev.db
 command: uv run uvicorn my_app.main:app --reload --host 0.0.0.0
 --port 8000
 healthcheck:
 test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
 interval: 30s
 timeout: 10s
 retries: 3

 frontend:
 build:
 context: ../frontend
 dockerfile: Dockerfile.dev
 ports:
 - "5173:5173"
 volumes:
 - ../frontend/src:/app/src:ro
 - ../frontend/public:/app/public:ro
 environment:
 - VITE_API_URL=http://localhost:8000/api
 command: npm run dev -- --host 0.0.0.0
 depends_on:
 - backend

Optional: PostgreSQL for production-like development
db:
```

```

image: postgres:16-alpine
ports:
 - "5432:5432"
environment:
 - POSTGRES_USER=myapp
 - POSTGRES_PASSWORD=devpassword
 - POSTGRES_DB=myapp
volumes:
 - postgres_data:/var/lib/postgresql/data

volumes:
 postgres_data:

```

## B.6 GitHub Actions Workflows

### B.6.1 Complete CI/CD Workflow

```

.github/workflows/ci-cd.yml
name: CI/CD

on:
 push:
 branches: [main]
 tags: ['v*']
 pull_request:
 branches: [main]

env:
 REGISTRY: ghcr.io
 IMAGE_NAME: ${github.repository}

jobs:
 # =====
 # Testing
 # =====
 test-backend:
 name: Test Backend
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v4

 - name: Set up Python
 uses: actions/setup-python@v5
 with:
 python-version: '3.11'

 - name: Install uv
 run: pip install uv

 - name: Install dependencies

```

```
 working-directory: ./backend
 run: uv sync --all-extras

- name: Run linting
 working-directory: ./backend
 run: |
 uv run ruff check .
 uv run ruff format --check .

- name: Run type checking
 working-directory: ./backend
 run: uv run mypy src/

- name: Run tests
 working-directory: ./backend
 run: uv run pytest --cov --cov-report=xml

- name: Upload coverage
 uses: codecov/codecov-action@v3
 with:
 files: ./backend/coverage.xml
 flags: backend

test-frontend:
 name: Test Frontend
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v4

 - name: Set up Node.js
 uses: actions/setup-node@v4
 with:
 node-version: '20'
 cache: 'npm'
 cache-dependency-path: frontend/package-lock.json

 - name: Install dependencies
 working-directory: ./frontend
 run: npm ci

 - name: Run linting
 working-directory: ./frontend
 run: npm run lint

 - name: Run type checking
 working-directory: ./frontend
 run: npm run typecheck

 - name: Run tests
 working-directory: ./frontend
 run: npm run test:coverage
```

```

- name: Build
 working-directory: ./frontend
 run: npm run build

=====
Docker Build
=====
build-docker:
 name: Build Docker Image
 needs: [test-backend, test-frontend]
 runs-on: ubuntu-latest
 if: github.event_name == 'push'
 permissions:
 contents: read
 packages: write

steps:
- uses: actions/checkout@v4

- name: Set up Docker Buildx
 uses: docker/setup-buildx-action@v3

- name: Log in to Container Registry
 uses: docker/login-action@v3
 with:
 registry: ${ env.REGISTRY }
 username: ${ github.actor }
 password: ${ secrets.GITHUB_TOKEN }

- name: Extract metadata
 id: meta
 uses: docker/metadata-action@v5
 with:
 images: ${ env.REGISTRY }/${ env.IMAGE_NAME }
 tags: |
 type=ref,event=branch
 type=semver,pattern={{version}}
 type=semver,pattern={{major}}.{{minor}}
 type=sha

- name: Build and push
 uses: docker/build-push-action@v5
 with:
 context: .
 file: ./docker/Dockerfile.combined
 push: true
 tags: ${ steps.meta.outputs.tags }
 labels: ${ steps.meta.outputs.labels }
 cache-from: type=gha
 cache-to: type=gha,mode=max

```

```
=====
Desktop Builds
=====
build-desktop:
 name: Build Desktop (${ matrix.os })
 needs: [test-backend, test-frontend]
 if: startsWith(github.ref, 'refs/tags/v')
 runs-on: ${ matrix.os }
 strategy:
 fail-fast: false
 matrix:
 include:
 - os: ubuntu-latest
 platform: linux
 - os: windows-latest
 platform: win
 - os: macos-latest
 platform: mac

 steps:
 - uses: actions/checkout@v4

 - name: Set up Node.js
 uses: actions/setup-node@v4
 with:
 node-version: '20'

 - name: Install frontend dependencies
 working-directory: ./frontend
 run: npm ci

 - name: Build frontend
 working-directory: ./frontend
 run: npm run build

 - name: Install Electron dependencies
 working-directory: ./electron
 run: npm ci

 - name: Build Electron app
 working-directory: ./electron
 run: npm run build:${ matrix.platform }
 env:
 GH_TOKEN: ${ secrets.GITHUB_TOKEN }

 - name: Upload artifacts
 uses: actions/upload-artifact@v4
 with:
 name: desktop-${ matrix.platform }
 path: |
```

```

 electron/dist/*.dmg
 electron/dist/*.zip
 electron/dist/*.exe
 electron/dist/*.AppImage
 electron/dist/*.deb
 if-no-files-found: ignore

=====
Release
=====
release:
 name: Create Release
 needs: [build-docker, build-desktop]
 if: startsWith(github.ref, 'refs/tags/v')
 runs-on: ubuntu-latest
 permissions:
 contents: write

 steps:
 - uses: actions/checkout@v4

 - name: Download all artifacts
 uses: actions/download-artifact@v4
 with:
 path: artifacts

 - name: Create Release
 uses: softprops/action-gh-release@v1
 with:
 files: artifacts/**/*
 generate_release_notes: true
 draft: false
 prerelease: ${contains(github.ref, '-') }

```

## B.7 CLAUDE.md Template

```

Project: [Your Project Name]

Overview
[Brief description of what this project does]

Architecture

Stack
- Backend: FastAPI (Python 3.11+)
- Frontend: React 18 + TypeScript + Vite
- Desktop: Electron 28
- Database: SQLite (dev) / PostgreSQL (prod)
- Deployment: Docker, GitHub Actions

```

```
Directory Structure
project/
 backend/ # FastAPI application
 src/my_app/ # Source code
 tests/ # Backend tests
 frontend/ # React application
 src/ # Source code
 electron/ # Desktop wrapper
 docker/ # Container configs
 .github/ # CI/CD workflows

Conventions

Backend
- All API routes under /api/ prefix
- Pydantic models for all request/response schemas
- SQLAlchemy for database operations
- Async handlers where beneficial

Frontend
- Functional components with hooks
- React Query for server state
- Axios for API calls
- TypeScript strict mode

General
- Semantic versioning (v1.2.3)
- Conventional commits
- All code formatted and linted before commit

Development Commands

Backend
cd backend
uv run uvicorn my_app.main:app --reload
uv run pytest
uv run ruff check .

Frontend
cd frontend
npm run dev
npm test
npm run lint

Docker
docker-compose -f docker/docker-compose.yml up

API Endpoints
- GET /api/health - Health check
- GET /api/items - List items
- POST /api/items - Create item
```

```

- GET /api/items/{id} - Get item
- PUT /api/items/{id} - Update item
- DELETE /api/items/{id} - Delete item

Environment Variables
- DEBUG - Enable debug mode (default: false)
- DATABASE_URL - Database connection string
- SECRET_KEY - Application secret key
- CORS_ORIGINS - Allowed CORS origins

Key Decisions
1. Using uv for Python dependency management (fast, reliable)
2. Electron over Tauri (more mature, better AI support)
3. SQLite for development (simple, no setup required)
4. Single Docker container for deployment (simpler ops)
5. GitHub Actions for CI/CD (integrated with repo)

Common Tasks

Adding a new API endpoint
1. Define Pydantic models in backend/src/my_app/models/
2. Create route handler in backend/src/my_app/api/
3. Add tests in backend/tests/
4. Update TypeScript types in frontend/src/services/api.ts

Adding a new React component
1. Create component in frontend/src/components/
2. Use React Query hooks for data fetching
3. Add to relevant pages/routes

Creating a release
1. Update version in relevant package files
2. Create git tag: git tag v1.2.3
3. Push tag: git push origin v1.2.3
4. GitHub Actions builds and publishes automatically

```

This appendix provides the essential templates for implementing the multi-platform distribution architecture. For complete, up-to-date templates with all configurations, see the companion repositories listed at the beginning of this appendix.

# Appendix C

## AI Tools for Python Development

The integration of AI into software development represents one of the most significant shifts in how developers work. This appendix provides an overview of AI tools available for Python development, guidance on how to use them effectively, and important considerations for their ethical use.

### C.1 Overview of Current AI Tools and Their Strengths

#### C.1.1 Code Assistants and Completion Tools

- **GitHub Copilot:**
  - **Strengths:** Real-time code suggestions directly in your IDE; trained on public GitHub repositories; understands context from open files
  - **Best for:** Rapid code generation, boilerplate reduction, exploring implementation alternatives
  - **Integration:** Available for VS Code, Visual Studio, JetBrains IDEs, and Neovim
- **JetBrains AI Assistant:**
  - **Strengths:** Deeply integrated with JetBrains IDEs; code explanation and generation; documentation creation
  - **Best for:** PyCharm users; explaining complex code; generating docstrings
  - **Integration:** Built into PyCharm and other JetBrains products
- **Tabnine:**
  - **Strengths:** Code completion with local models option; privacy-focused; adapts to your coding style
  - **Best for:** Teams with strict data privacy requirements; personalized code suggestions
  - **Integration:** Works with most popular IDEs including VS Code and PyCharm

## C.1.2 Conversational AI Assistants

- **Claude (Anthropic):**
  - **Strengths:** Excellent reasoning capabilities; strong Python knowledge; handles lengthy context
  - **Best for:** Complex problem-solving; explaining algorithms; reviewing code; documentation creation
  - **Access:** Web interface, API, Claude Code (terminal)
- **ChatGPT/GPT-4 (OpenAI):**
  - **Strengths:** Wide knowledge base; good at generating code and explaining concepts
  - **Best for:** Troubleshooting; learning concepts; brainstorming ideas; code generation
  - **Access:** Web interface, API, plugins for various platforms
- **Gemini (Google):**
  - **Strengths:** Strong code analysis and generation; multimodal capabilities useful for analysing diagrams
  - **Best for:** Code support; learning resources; teaching concepts
  - **Access:** Web interface, API, Duet AI integrations

## C.1.3 AI-Enhanced Code Review Tools

- **DeepSource:**
  - **Strengths:** Continuous analysis; focuses on security issues, anti-patterns, and performance
  - **Best for:** Automated code reviews; maintaining code quality standards
  - **Integration:** GitHub, GitLab, Bitbucket
- **Codiga:**
  - **Strengths:** Real-time code analysis; custom rule creation; automated PR comments
  - **Best for:** Enforcing team-specific best practices; providing quick feedback
  - **Integration:** GitHub, GitLab, Bitbucket, and various IDEs
- **Sourcery:**
  - **Strengths:** Python-specific refactoring suggestions; explains why changes are recommended
  - **Best for:** Learning better Python patterns; gradual code quality improvement
  - **Integration:** VS Code, JetBrains IDEs, GitHub

## C.1.4 AI Documentation Tools

- **Mintlify Writer:**
  - **Strengths:** Auto-generates documentation from code; supports various docstring formats
  - **Best for:** Quickly documenting existing codebases; maintaining consistent documentation
  - **Integration:** VS Code, JetBrains IDEs
- **Docstring Generator AI:**
  - **Strengths:** Creates detailed docstrings following specified formats (Google, NumPy, etc.)
  - **Best for:** Consistently formatting documentation across a project
  - **Integration:** VS Code extension

## C.2 Guidelines for Effective Prompting

The quality of AI output depends significantly on how you formulate your requests. Here are strategies to get the most from AI tools when working with Python:

### C.2.1 General Prompting Principles

1. **Be specific and detailed:** Include relevant context about your project, such as Python version, frameworks used, and existing patterns to follow.

```
Less effective
"Write a function to process user data."

More effective
"Write a Python 3.10 function to process user data that:
- Takes a dictionary of user attributes
- Validates email and age fields
- Returns a normalized user object
- Follows our project's error handling pattern of raising
ValueError with descriptive messages
- Uses type hints"
```

2. **Provide examples:** When you need code that follows certain patterns or styles, provide examples.

```
"Here's how we write API handler functions in our project:

async def get_user(user_id: int) -> Dict[str, Any]:
 try:
 response = await http_client.get(f"/users/{user_id}")
 return response.json()
 except HTTPError as e:
 log.error(f"Failed to fetch user {user_id}: {e}")
 raise UserFetchError(f"Could not retrieve user: {e}")

Please write a similar function for fetching user orders."
```

3. **Use iterative refinement:** Start with a basic request, then refine the results.

```
Initial prompt
"Write a function to parse CSV files with pandas."

Follow-up refinements
"Now add error handling for missing files."
"Update it to support both comma and semicolon delimiters."
"Add type hints to the function."
```

4. **Specify output format:** Clarify how you want information presented.

```
"Explain the difference between @classmethod and @staticmethod in
Python.
Format your response with:
1. A brief definition of each
2. Code examples showing typical use cases"
```

```
3. A table comparing their key attributes"
```

## C.2.2 Python-Specific Prompting Strategies

1. **Request specific Python versions or features:** Clarify which Python version you're targeting.

```
"Write this function using Python 3.9+ features like the new dictionary merge operator."
```

2. **Specify testing frameworks:** When requesting tests, mention your preferred framework.

```
"Generate pytest test cases for this function, using fixtures and parametrize for the test scenarios."
```

3. **Ask for alternative approaches:** Python often offers multiple solutions to problems.

```
"Show three different ways to implement this list filtering function, explaining the tradeoffs between readability, performance, and memory usage."
```

4. **Request educational explanations:** For learning purposes, ask the AI to explain its reasoning.

```
"Write a function to efficiently find duplicate elements in a list, then explain why the algorithm you chose is efficient and what its time complexity is."
```

## C.2.3 Using AI for Code Review

When using AI to review your Python code, structured prompts yield better results:

```
"Review this Python code for:
1. Potential bugs or edge cases
2. Performance issues
3. Pythonic improvements
4. PEP 8 compliance
5. Possible security concerns

def process_user_input(data):
 # [your code here]

For each issue found, please:
- Describe the problem
- Explain why it's problematic
- Suggest a specific improvement with code"
```

## C.2.4 Troubleshooting with AI

When debugging problems, provide context systematically:

```
"I'm getting this error when running my Python script:

[Error message]

Here's the relevant code:

[your code here]

I've already tried:
1. [attempted solution 1]
2. [attempted solution 2]

I'm using Python 3.9 with packages: pandas 1.5.3, numpy 1.23.0

What might be causing this error and how can I fix it?"
```

## C.3 Ethical Considerations and Limitations

As you integrate AI tools into your Python development workflow, consider these important ethical considerations and limitations:

### C.3.1 Ethical Considerations

#### 1. Intellectual Property and Licensing

- Code generated by AI may be influenced by training data with various licenses
- For commercial projects, consult your legal team about AI code usage policies
- Consider adding comments attributing AI-generated sections when substantial

#### 2. Security Risks

- Never blindly implement AI-suggested security-critical code without review
- AI may recommend outdated or vulnerable patterns it learned from older code
- Verify cryptographic implementations, authentication mechanisms, and input validation independently

#### 3. Overreliance and Skill Development

- Balance AI usage with developing personal understanding
- For educational settings, consider policies on appropriate AI assistance
- Use AI to enhance learning rather than bypass it

#### 4. Bias and Fairness

- AI may perpetuate biases present in training data
- Review generated code for potential unfair treatment or assumptions
- Be especially careful with user-facing features and data processing pipelines

#### 5. Environmental Impact

- Large AI models have significant computational and energy costs
- Consider using more efficient, specialized code tools for routine tasks
- Batch similar requests when possible instead of making many small queries

### C.3.2 Technical Limitations

#### 1. Knowledge Cutoffs

- AI assistants have knowledge cutoffs and may not be aware of recent Python developments
  - Verify suggestions for newer Python versions or recently updated libraries
  - Example: An AI might not know about features introduced in Python 3.11 or 3.12 if its training cutoff predates them
2. **Context Length Restrictions**
    - Most AI tools have limits on how much code they can process at once
    - For large files or complex projects, focus queries on specific components
    - Provide essential context rather than entire codebases
  3. **Hallucinations and Inaccuracies**
    - AI can confidently suggest incorrect implementations or non-existent functions
    - Always verify generated code works as expected
    - Be especially wary of package import suggestions, API usage patterns, and framework-specific code
  4. **Understanding Project-Specific Context**
    - AI lacks full understanding of your project architecture and requirements
    - Generated code may not align with your established patterns or constraints
    - Always review for compatibility with your broader codebase
  5. **Time-Sensitive Information**
    - Best practices, dependencies, and security recommendations change over time
    - Verify suggestions against current Python community standards
    - Double-check deprecation warnings and avoid outdated patterns

### C.3.3 Practical Mitigation Strategies

1. **Code Review Process**
  - Establish clear guidelines for reviewing AI-generated code
  - Use the same quality standards for AI-generated and human-written code
  - Consider automated testing requirements for AI contributions
2. **Attribution and Documentation**
  - Document where and how AI tools were used in your development process
  - Consider noting substantial AI contributions in code comments
  - Example: `# Initial implementation generated by GitHub Copilot, modified to handle edge cases`
3. **Verification Practices**
  - Test AI-generated code thoroughly, especially edge cases
  - Verify performance characteristics claimed by AI suggestions
  - Cross-check security recommendations with trusted sources
4. **Balanced Use Policy**
  - Develop team guidelines for appropriate AI tool usage
  - Encourage use for boilerplate, documentation, and creative starting points
  - Emphasize human oversight for architecture, security, and critical algorithms
5. **Continuous Learning**
  - Use AI explanations as learning opportunities
  - Ask AI to explain its suggestions and verify understanding
  - Build knowledge to reduce dependency on AI for core concepts

## C.4 The Future of AI in Python Development

AI tools for Python development are evolving rapidly. Current trends suggest these future directions:

- **More specialized Python-specific models:** Trained specifically on Python codebases with deeper framework understanding
- **Enhanced IDE integration:** More seamless AI assistance throughout the development workflow
- **Improved testing capabilities:** AI generating more comprehensive test suites with higher coverage
- **Custom models for organisations:** Trained on internal codebases to better match company standards
- **Agent-based development:** AI systems that can execute multi-step development tasks with minimal guidance

As these tools evolve, maintaining a balanced approach that leverages AI strengths while preserving human oversight will remain essential for quality Python development.



# Appendix D

## Python Development Workflow Checklist

This checklist provides a practical reference for setting up and maintaining Python projects of different scales. Choose the practices that match your project's complexity and team size.

Development Stage	Simple/Beginner Project	Intermediate/Large Project
<b>Project Setup</b>		
Create project structure	Basic directory with code and tests	Full <code>src</code> layout with package under <code>src/</code>
Initialize version control	<code>git init</code> and basic <code>.gitignore</code>	Advanced <code>.gitignore</code> with branch strategies
Add essential files	README.md	README.md, LICENSE, CONTRIBUTING.md
<b>Environment Setup</b>		
Create virtual environment	<code>python -m venv .venv</code>	<code>uv venv</code> or containerized environment
Track dependencies	<code>pip freeze &gt; requirements.txt</code>	<code>requirements.in</code> with <code>pip-compile</code> or <code>uv pip compile</code>
Install dependencies	<code>pip install -r requirements.txt</code>	<code>pip-sync</code> or <code>uv pip sync</code>
<b>Code Quality</b>		
Formatting	Basic PEP 8 adherence	Automated with Ruff ( <code>ruff format</code> )
Linting	or basic Flake8	Ruff with multiple rule sets enabled
Type checking	or basic annotations	<code>mypy</code> with increasing strictness
Security scanning		Bandit
Dead code detection		Vulture

Development Stage	Simple/Beginner Project	Intermediate/Large Project
<b>Testing</b>		
Unit tests	Basic pytest	Comprehensive pytest with fixtures
Test coverage	or basic	pytest-cov with coverage targets
Mocking		pytest-mock for external dependencies
Integration tests		For component interactions
Functional tests		For key user workflows
<b>Documentation</b>		
Code documentation	Basic docstrings	Comprehensive docstrings (Google/NumPy style)
API documentation	Generated with pydoc	MkDocs + mkdocstrings
User guides	Basic README usage examples	Comprehensive MkDocs site with tutorials
<b>Version Control Practices</b>		
Commit frequency	Regular commits	Atomic, focused commits
Commit messages	Basic descriptive messages	Structured commit messages with context
Branching	or basic feature branches	Git-flow or trunk-based with feature branches
Code reviews		Pull/Merge requests with review guidelines
<b>Automation</b>		
Local automation		pre-commit hooks
CI pipeline	or basic	GitHub Actions with matrix testing
CD pipeline		Automated deployments/releases
<b>Packaging &amp; Distribution</b>		
Package configuration	Basic <code>pyproject.toml</code>	Comprehensive configuration with extras
Build system	Basic <code>setuptools</code>	Modern build with PEP 517 support
Release process	Manual versioning	Semantic versioning with automation
Publication	or manual PyPI upload	Automated PyPI deployment via CI
<b>Maintenance</b>		
Dependency updates	Manual updates	Scheduled updates with <code>dependabot</code>

---

Development Stage	Simple/Beginner Project	Intermediate/Large Project
Security monitoring		Vulnerability scanning
Performance profiling		Regular profiling and benchmarking
User feedback channels		Issue templates and contribution guidelines

---

## D.1 Project Progression Path

For projects that start simple but grow in complexity, follow this progression:

- 1. Start with the essentials:**
  - Project structure and version control
  - Virtual environment
  - Basic testing
  - Clear README
- 2. Add code quality tools incrementally:**
  - First add Ruff for formatting and basic linting
  - Then add mypy for critical modules
  - Finally add security scanning
- 3. Enhance testing as complexity increases:**
  - Add coverage reporting
  - Implement mocking for external dependencies
  - Add integration tests for component interactions
- 4. Improve documentation with growth:**
  - Start with good docstrings from day one
  - Transition to MkDocs when README becomes insufficient
  - Generate API documentation from docstrings
- 5. Automate processes as repetition increases:**
  - Add pre-commit hooks for local checks
  - Implement CI for testing across environments
  - Add CD when deployment becomes routine

Remember: Don't overengineer! Choose the practices that add value to your specific project and team. It's better to implement a few practices well than to poorly implement many.



# Appendix E

## Introduction to Python IDEs and Editors

While this book focuses on Python development practices rather than specific tools, your choice of development environment can significantly impact your productivity and workflow. This appendix provides a brief overview of popular editors and IDEs for Python development, with particular attention to how they integrate with the tools and practices discussed throughout this book.

### E.1 Visual Studio Code

Visual Studio Code (VS Code) has become one of the most popular editors for Python development due to its balance of lightweight design and powerful features.

#### E.1.1 Key Features for Python Development

- **Python Extension:** Microsoft's official Python extension provides IntelliSense, linting, debugging, code navigation, and Jupyter notebook support
- **Virtual Environment Detection:** Automatically detects and allows switching between virtual environments
- **Integrated Terminal:** Run Python scripts and commands without leaving the editor
- **Debugging:** Full-featured debugging with variable inspection and breakpoints
- **Extensions Ecosystem:** Rich marketplace with extensions for most Python tools

#### E.1.2 Integration with Development Tools

- **Virtual Environments:** Detects venv, conda, and other environment types; shows active environment in status bar
- **Linting/Formatting:** Native integration with Ruff, Black, mypy, and other quality tools
- **Testing:** Test Explorer UI for pytest, unittest
- **Package Management:** Terminal integration for pip, Poetry, PDM, and other package managers
- **Git:** Built-in Git support for commits, branches, and pull requests

### E.1.3 Configuration Example

.vscode/settings.json:

```
{
 "python.defaultInterpreterPath":
 "${workspaceFolder}/.venv/bin/python",
 "python.formatting.provider": "none",
 "editor.formatOnSave": true,
 "editor.codeActionsOnSave": {
 "source.fixAll.ruff": true,
 "source.organizeImports.ruff": true
 },
 "python.testing.pytestEnabled": true,
 "python.linting.mypyEnabled": true
}
```

### E.1.4 AI-Assistant Integration

- **GitHub Copilot:** Code suggestions directly in the editor
- **IntelliCode:** AI-enhanced code completions
- **Live Share:** Collaborative coding sessions

## E.2 Neovim

Neovim is a highly extensible text editor popular among developers who prefer keyboard-centric workflows and extensive customisation.

### E.2.1 Key Features for Python Development

- **Extensible Architecture:** Lua-based configuration and plugin system
- **Terminal Integration:** Built-in terminal emulator
- **Modal Editing:** Efficient text editing with different modes
- **Performance:** Fast startup and response, even for large files

### E.2.2 Integration with Development Tools

- **Language Server Protocol (LSP):** Native support for Python language servers like Pyright and Jedi
- **Virtual Environments:** Support through plugins and configuration
- **Code Completion:** Various completion engines (nvim-cmp, COC)
- **Linting/Formatting:** Integration with tools like Ruff, Black, and mypy
- **Testing:** Run tests through plugins or terminal integration

### E.2.3 Configuration Example

Simplified `init.lua` excerpt for Python development:

```
-- Python LSP setup
require('lspconfig').pyright.setup{
 settings = {
 python = {
```

```

 analysis = {
 typeCheckingMode = "basic",
 autoSearchPaths = true,
 useLibraryCodeForTypes = true
 }
 }
}
}

-- Formatting on save with Black
vim.api.nvim_create_autocmd("BufWritePre", {
 pattern = "*.py",
 callback = function()
 vim.lsp.buf.format()
 end,
})

```

### E.2.4 AI-Assistant Integration

- **GitHub Copilot.vim:** Code suggestions
- **Neural:** Code completions powered by local models

## E.3 Emacs

Emacs is a highly customisable text editor with a rich ecosystem of packages and a long history in the development community.

### E.3.1 Key Features for Python Development

- **Extensibility:** customisable with Emacs Lisp
- **Org Mode:** Literate programming and documentation
- **Multiple Modes:** Specialized modes for different file types
- **Integrated Environment:** Email, shell, and other tools integrated

### E.3.2 Integration with Development Tools

- **Python Mode:** Syntax highlighting, indentation, and navigation for Python
- **Virtual Environments:** Support through pyvenv, conda.el
- **Linting/Formatting:** Integration with Flycheck, Black, Ruff
- **Testing:** Run tests with pytest-emacs
- **Package Management:** Manage dependencies through shell integration

### E.3.3 Configuration Example

Excerpt from `.emacs` or `init.el`:

```

;; Python development setup
(use-package python-mode
 :ensure t
 :config
 (setq python-shell-interpreter "python3"))

```

```
(use-package blacken
 :ensure t
 :hook (python-mode . blacken-mode))

(use-package pyvenv
 :ensure t
 :config
 (pyvenv-mode 1))
```

### E.3.4 AI-Assistant Integration

- **Copilot.el:** GitHub Copilot integration
- **ChatGPT-shell:** Interact with LLMs from within Emacs

## E.4 AI-Enhanced Editors

### E.4.1 Cursor

Cursor (formerly Warp AI) is built on top of VS Code but focused on AI-assisted development.

#### E.4.1.1 Key Features

- **AI Chat:** Integrated chat interface for coding assistance
- **Code Explanation:** Ask about selected code
- **Code Generation:** Generate code from natural language descriptions
- **VS Code Base:** All VS Code features and extensions available
- **customised for AI Interaction:** UI designed around AI-assisted workflows

#### E.4.1.2 Integration with Python Tools

- Inherits VS Code's excellent Python ecosystem support
- AI features that understand Python code context
- Assistance with complex Python patterns and libraries

### E.4.2 Whisper (Anthropic)

Claude Code (Whisper) from Anthropic is an AI-enhanced development environment:

#### E.4.2.1 Key Features

- **Terminal-Based Assistant:** AI-powered code generation from the command line
- **Task Automation:** Natural language for development tasks
- **Context-Aware Assistance:** Understands project structure and code
- **Code Explanation:** In-depth explanations of complex code

#### E.4.2.2 Integration with Python Tools

- Works alongside existing development environments
- Can assist with tool configuration and integration

- Helps debug issues with Python tooling

## E.5 Choosing the Right Environment

The best development environment depends on your specific needs:

- **VS Code:** Excellent for most Python developers; balances ease of use with powerful features
- **Neovim:** Ideal for keyboard-focused developers who value speed and customisation
- **Emacs:** Great for developers who want an all-in-one environment with deep customisation
- **AI-Enhanced Editors:** Valuable for those looking to leverage AI in their workflow

Consider these factors when choosing:

1. **Learning curve:** VS Code has a gentle learning curve, while Neovim and Emacs require more investment
2. **Performance needs:** Neovim offers the best performance for large files
3. **Extensibility importance:** Emacs and Neovim offer the deepest customisation
4. **Team standards:** Consider what your team uses for easier collaboration
5. **AI assistance:** If AI-assisted development is important, specialized editors may offer better integration

## E.6 Editor-Agnostic Best Practices

Regardless of your chosen editor, follow these best practices:

1. **Learn keyboard shortcuts:** They dramatically increase productivity
2. **Use extensions for Python tools:** Integrate the tools from this book
3. **Set up consistent formatting:** Configure your editor to use the same tools as your CI pipeline
4. **customise for your workflow:** Adapt your environment to your specific needs
5. **Version control your configuration:** Track editor settings in Git for consistency

Remember that the editor is just a tool—the development practices in this book can be applied regardless of your chosen environment. The best editor is the one that helps you implement good development practices while staying out of your way during the creative process.



# Appendix F

## Python Development Tools Reference

This reference provides brief descriptions of the development tools mentioned throughout the guide, organised by their primary function.

### F.1 Environment & Dependency Management

- **venv**: Python's built-in tool for creating isolated virtual environments.
- **pip**: The standard package installer for Python.
- **pip-tools**: A set of tools for managing Python package dependencies with pinned versions via requirements.txt files.
- **uv**: A Rust-based, high-performance Python package manager and environment manager compatible with pip.
- **pipx**: A tool for installing and running Python applications in isolated environments.

### F.2 Code Quality & Formatting

- **Ruff**: A fast, Rust-based Python linter and formatter that consolidates multiple tools.
- **Black**: An opinionated Python code formatter that enforces a consistent style.
- **isort**: A utility to sort Python imports alphabetically and automatically separate them into sections.
- **Flake8**: A code linting tool that checks Python code for style and logical errors.
- **Pylint**: A comprehensive Python static code analyzer that looks for errors and enforces coding standards.

### F.3 Testing

- **pytest**: A powerful, flexible testing framework for Python that simplifies test writing and execution.
- **pytest-cov**: A pytest plugin for measuring code coverage during test execution.

- **pytest-mock**: A pytest plugin for creating and managing mock objects in tests.

## F.4 Type Checking

- **mypy**: A static type checker for Python that helps catch type-related errors before runtime.
- **pydoc**: Python's built-in documentation generator and help system.

## F.5 Security & Code Analysis

- **Bandit**: A tool designed to find common security issues in Python code.
- **Vulture**: A tool that detects unused code in Python programs.

## F.6 Documentation

- **MkDocs**: A fast and simple static site generator for building project documentation from Markdown files.
- **mkdocs-material**: A Material Design theme for MkDocs.
- **mkdocstrings**: A MkDocs plugin that automatically generates documentation from docstrings.
- **Sphinx**: A comprehensive documentation tool that supports multiple output formats.

## F.7 Package Building & Distribution

- **build**: A simple, correct PEP 517 package builder for Python projects.
- **twine**: A utility for publishing Python packages to PyPI securely.
- **setuptools**: The standard library for packaging Python projects.
- **setuptools-scm**: A tool that manages your Python package versions using git metadata.
- **wheel**: A built-package format for Python that provides faster installation.

## F.8 Continuous Integration & Deployment

- **GitHub Actions**: GitHub's built-in CI/CD platform for automating workflows.
- **pre-commit**: A framework for managing and maintaining pre-commit hooks.
- **Codecov**: A tool for measuring and reporting code coverage in CI pipelines.

## F.9 Version Control

- **Git**: A distributed version control system for tracking changes in source code.
- **GitHub/GitLab**: Web-based platforms for hosting Git repositories with collaboration features.

## F.10 Project Setup & Management

- **Cookiecutter:** A command-line utility that creates projects from templates, enabling consistent project setup with predefined structure and configurations. It uses a templating system to generate files and directories based on user inputs.
- **GitHub Repository Templates:** A GitHub feature that allows repositories to serve as templates for new projects. Users can generate new repositories with the same directory structure and files without needing to install additional tools. Unlike cookiecutter, GitHub templates don't support parameterization but offer a zero-installation approach to project scaffolding.

## F.11 Advanced Tools

- **Cython:** A language that makes writing C extensions for Python as easy as writing Python.
- **Docker:** A platform for developing, shipping, and running applications in containers.
- **Kubernetes:** An open-source system for automating deployment, scaling, and management of containerized applications.
- **Pants/Bazel:** Build systems designed for monorepos and large codebases.



# Appendix G

## Comparison of Python Environment and Package Management Tools

This appendix provides a side-by-side comparison of the major Python environment and package management tools covered throughout this book.

### G.1 Comparison Table

Feature	venv	conda	uv	Hatch	Poetry	PDM
<b>Core Focus</b>	Virtual environments	Environments & packages across languages	Fast package installation	Project management	Dependency management & packaging	Standards-compliant packaging
<b>Implementation Language</b>	Python	Python	Rust	Python	Python	Python
<b>Performance</b>	Standard	Moderate	Very Fast	Standard	Moderate	Fast
<b>Virtual Environment Support</b>	Built-in	Built-in	Built-in	Built-in	Built-in	Optional (PEP 582)
<b>Lock File</b>	No (requires pip-tools)	No (uses explicit envs)	Yes	Yes	Yes	Yes
<b>Dependency Resolution</b>	Basic (via pip)	Sophisticated	Efficient	Basic	Sophisticated	Sophisticated
<b>Non-Python Dependencies</b>	No	Yes	No	No	No	No

Feature	venv	conda	uv	Hatch	Poetry	PDM
<b>Project Config File</b>	None	environ- ment.yml	re- quire- ments.txt	pypro- ject.toml	pypro- ject.toml	pypro- ject.toml
<b>PEP 621 Compliance</b>	N/A	No	N/A	Yes	Partial	Yes
<b>Multiple Environment Management</b>	No (one env per directory)	Yes	No	Yes	No	Via configu- ration
<b>Dependency Groups</b>	No	Via separate files	Via sepa- rate files	Yes	Yes	Yes
<b>Package Building</b>	No	Limited	No	Yes	Yes	Yes
<b>Publishing to PyPI</b>	No	Limited	No	Yes	Yes	Yes
<b>Cross-Platform Support</b>	Yes	Yes	Yes	Yes	Yes	Yes
<b>Best For</b>	Simple projects, teaching	Scien- tific/ML projects	Fast instal- la- tions, CI en- viron- ments	Dev work- flow au- toma- tion	Library development	Standards- focused projects
<b>Learning Curve</b>	Low	Moderate	Low	Mod- erate	Moderate- High	Moder- ate
<b>Script/Task Running</b>	No	Limited	No	Ad- vanced	Basic	Ad- vanced
<b>Community Size/Adoption</b>	Very High	Very High	Grow- ing	Mod- erate	High	Grow- ing
<b>Plugin System</b>	No	No	No	Yes	Limited	Yes
<b>Development Status</b>	Sta- ble/Ma- ture	Stable/Ma- ture	Active Devel- op- ment	Active Devel- op- ment	Stable/Ma- ture	Active Devel- opment

## G.2 Installation Methods

Tool	pip/pipx	Home- brew	Official Installer	Platform Package Managers
<b>venv</b>	Built-in with Python	N/A	N/A	N/A

Tool	pip/pipx	Home- brew	Official Installer	Platform Package Managers
<b>conda</b>	No	Yes	Yes (Mini-conda/Anaconda)	Some
<b>uv</b>	Yes	Yes	Yes (curl installer)	Growing
<b>Hatch</b>	Yes	Yes	No	Some
<b>Poetry</b>	Yes	Yes	Yes (custom installer)	Some
<b>PDM</b>	Yes	Yes	No	Some

## G.3 Typical Usage Patterns

Tool	Typical Command Sequence
<b>venv</b>	<code>python -m venv .venv &amp;&amp; source .venv/bin/activate &amp;&amp; pip install -r requirements.txt</code>
<b>conda</b>	<code>conda create -n myenv python=3.10 &amp;&amp; conda activate myenv &amp;&amp; conda install pandas numpy</code>
<b>uv</b>	<code>uv venv &amp;&amp; source .venv/bin/activate &amp;&amp; uv pip sync requirements.txt</code>
<b>Hatch</b>	<code>hatch init &amp;&amp; hatch shell &amp;&amp; hatch run test</code>
<b>Poetry</b>	<code>poetry init &amp;&amp; poetry add requests &amp;&amp; poetry install &amp;&amp; poetry run python script.py</code>
<b>PDM</b>	<code>pdm init &amp;&amp; pdm add requests pytest --dev &amp;&amp; pdm install &amp;&amp; pdm run pytest</code>

## G.4 Use Case Recommendations

### G.4.1 For Beginners

1. **venv + pip**: Simplest to understand, built-in to Python
2. **uv**: Fast, familiar pip-like interface with modern features

### G.4.2 For Data Science/Scientific Computing

1. **conda**: Best support for scientific packages and non-Python dependencies
2. **Poetry** or **PDM**: When standard Python packages are sufficient

### G.4.3 For Library Development

1. **Poetry**: Great packaging and publishing workflows
2. **Hatch**: Excellent for multi-environment testing
3. **PDM**: Standards-compliant approach

### G.4.4 For Application Development

1. **PDM**: PEP 582 mode simplifies deployment
2. **Poetry**: Lock file ensures reproducible environments
3. **Hatch**: Task management features help automate workflows

### G.4.5 For CI/CD Environments

1. **uv**: Fastest installation speeds
2. **Poetry/PDM**: Reliable lock files ensure consistency

### G.4.6 For Teams with Mixed Experience Levels

1. **Poetry**: Opinionated approach enforces consistency
2. **uv**: Familiar interface with performance benefits
3. **Hatch**: Flexibility for different team workflows

## G.5 Migration Paths

From	To	Migration Approach
<b>pip + requirements.txt</b>	<b>uv</b>	Use directly with existing requirements.txt
<b>pip + requirements.txt</b>	<b>Poetry</b>	<code>poetry init</code> then <code>poetry add packages</code>
<b>pip + requirements.txt</b>	<b>PDM</b>	<code>pdm import -f requirements requirements.txt</code>
<b>conda</b>	<b>Poetry/PDM</b>	Export conda env to requirements, then import
<b>Pipenv</b>	<b>Poetry</b>	<code>poetry init</code> + manual migration or conversion tools
<b>Pipenv</b>	<b>PDM</b>	<code>pdm import -f pipenv Pipfile</code>
<b>Poetry</b>	<b>PDM</b>	<code>pdm import -f poetry pyproject.toml</code>

## G.6 When to Consider Multiple Tools

Some projects benefit from using multiple tools for different purposes:

- **conda + pip**: Use conda for complex dependencies, pip for Python-only packages
- **venv + uv**: Use venv for environment isolation, uv for fast package installation
- **Hatch + uv**: Use Hatch for project workflows, uv for faster installations

## G.7 Future Trends

The Python packaging ecosystem continues to evolve toward:

1. **Standards Compliance**: Increasing adoption of PEPs 518, 517, 621
2. **Performance optimisation**: More Rust-based tools like uv
3. **Simplified Workflows**: Better integration between tools
4. **Improved Lock Files**: More secure and deterministic builds
5. **Better Environment Management**: Alternatives to traditional virtual environments

By understanding the strengths and trade-offs of each tool, you can select the approach that best fits your specific project requirements and team preferences.

## Appendix H

# Python Development Pipeline Scaffold Python Script

```
#!/bin/bash
scaffold_python_project.sh - A simple script to create a Python
project with best practices
Usage: ./scaffold_python_project.sh my_project

if [-z "$1"]; then
 echo "Please provide a project name."
 echo "Usage: ./scaffold_python_project.sh my_project"
 exit 1
fi

PROJECT_NAME=$1
Convert hyphens to underscores for Python package naming conventions
PACKAGE_NAME=$(echo $PROJECT_NAME | tr '-' '_')

echo "Creating project: $PROJECT_NAME"
echo "Package name will be: $PACKAGE_NAME"

Create project directory
mkdir -p $PROJECT_NAME
cd $PROJECT_NAME

Create basic structure following the recommended src layout
The src layout enforces proper package installation and creates clear
boundaries
mkdir -p src/$PACKAGE_NAME tests docs

Create package files
__init__.py makes the directory a Python package
touch src/$PACKAGE_NAME/__init__.py
touch src/$PACKAGE_NAME/main.py
```

```

Create test files - keeping tests separate but adjacent to the
implementation
This follows the principle of separating implementation from tests
touch tests/__init__.py
touch tests/test_main.py

Create documentation placeholder - establishing documentation from the
start
Even minimal docs are better than no docs
echo "# $PROJECT_NAME Documentation" > docs/index.md

Create README.md with basic information
README is the first document anyone sees and should provide clear
instructions
echo "# $PROJECT_NAME

A Python project created with best practices.

Installation

\\\`bash
pip install $PROJECT_NAME
\\\`

Usage

\\\`python
from $PACKAGE_NAME import main
\\\`

Development

\\\`bash
Create virtual environment
python -m venv .venv
source .venv/bin/activate # On Windows: .venv\\Scripts\\activate

Install in development mode
pip install -e .

Run tests
pytest
\\\`
" > README.md

Create .gitignore file to exclude unnecessary files from version
control
This prevents committing files that should not be in the repository
echo "# Python
__pycache__/
*.py[cod]"

```

```

*$py.class
*.so
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
*.egg-info/
.installed.cfg
*.egg

Virtual environments
Never commit virtual environments to version control
.venv/
venv/
ENV/

Testing
.pytest_cache/
.coverage
htmlcov/

Documentation
docs/_build/

IDE
.idea/
.vscode/
*.swp
*.swo
" > .gitignore

Create pyproject.toml for modern Python packaging
This follows PEP 517/518 standards and centralizes project
configuration
echo "[build-system]
requires = [\"setuptools>=61.0\", \"wheel\"]
build-backend = \"setuptools.build_meta\"

[project]
name = \"$PROJECT_NAME\"
version = \"0.1.0\"
description = \"A Python project created with best practices\"

```

```

readme = \"README.md\"
requires-python = \">=3.8\"
authors = [
 {name = \"Your Name\", email = \"your.email@example.com\"}
]

[project.urls]
\"Homepage\" = \"https://github.com/yourusername/$PROJECT_NAME\"

Specify the src layout for better package isolation
[tool.setuptools]
package-dir = {\"\" = \"src\"}
packages = [\"$PACKAGE_NAME\"]

Configure pytest to look in the tests directory
[tool.pytest.ini_options]
testpaths = [\"tests\"]
" > pyproject.toml

Create requirements.in for direct dependencies
This approach is cleaner than freezing everything with pip freeze
echo \"# Project dependencies
Add your dependencies here, e.g.:
requests>=2.25.0
" > requirements.in

Create example main.py with docstrings and type hints
Starting with good documentation and typing practices from the
beginning
echo \"\"\"Main module for $PROJECT_NAME.\"\"\"

def example_function(text: str) -> str:
 \"\"\"Return a greeting message.

 Args:
 text: The text to include in the greeting.

 Returns:
 A greeting message.
 \"\"\"
 return f\"Hello, {text}!\"
" > src/$PACKAGE_NAME/main.py

Create example test file
Tests verify that code works as expected and prevent regressions
echo \"\"\"Tests for the main module.\"\"\"

from $PACKAGE_NAME.main import example_function

def test_example_function():
 \"\"\"Test the example function returns the expected greeting.\"\"\"

```

```
 result = example_function(\"World\")
 assert result == \"Hello, World!\"
" > tests/test_main.py

Initialize git repository
Version control should be established from the very beginning
git init
git add .
git commit -m "Initial project setup"

echo ""
echo "Project $PROJECT_NAME created successfully!"
echo ""
echo "Next steps:"
echo "1. cd $PROJECT_NAME"
echo "2. python -m venv .venv"
echo "3. source .venv/bin/activate # On Windows:
.venv\\Scripts\\activate"
echo "4. pip install -e ."
echo "5. pytest"
echo ""
echo "Happy coding!"
```



# Appendix I

## Cookiecutter Template

This appendix introduces and explains the companion cookiecutter template for the Python Development Pipeline described in this book. The template allows you to quickly scaffold new Python projects that follow all the recommended practices, saving you time and ensuring consistency across your projects.

### I.1 What is Cookiecutter?

Cookiecutter is a command-line utility that creates projects from templates. It takes a template directory containing a `cookiecutter.json` file with template variables and replaces them with user-provided values, generating a project directory structure with all necessary files.

### I.2 Getting Started with the Template

#### I.2.1 Prerequisites

- Python 3.7 or later
- pip package manager

#### I.2.2 Installation

First, install cookiecutter:

```
pip install cookiecutter
```

#### I.2.3 Creating a New Project

To create a new project using our Python Development Pipeline template:

```
cookiecutter gh:michael-borck/ship-python-cookiecutter
```

You'll be prompted to provide information about your project, such as:

- Project name
- Author information

- Python version requirements
- License type
- Development level (basic or advanced)
- Documentation preferences
- CI/CD preferences
- Package manager choice (pip-tools or uv)

After answering these questions, cookiecutter will generate a complete project structure with all the configuration files and setup based on your choices.

## I.3 Template Features

The template implements all the best practices discussed throughout this book:

### I.3.1 Project Structure

- Uses the recommended `src` layout for better package isolation
- Properly organised test directory
- Documentation setup with MkDocs (if selected)
- Clear separation of concerns across files and directories

### I.3.2 Development Environment

- Configured virtual environment instructions
- Dependency management using either pip-tools or uv
- `requirements.in` and `requirements-dev.in` files for clean dependency specification

### I.3.3 Code Quality Tools

- Ruff for formatting and linting
- mypy for type checking
- Bandit for security analysis (with advanced setup)
- Pre-configured with sensible defaults in `pyproject.toml`

### I.3.4 Testing

- pytest setup with example tests
- Coverage configuration
- Test helper fixtures

### I.3.5 Documentation

- MkDocs with Material theme (if selected)
- API documentation generation with mkdocstrings
- Template pages for quickstart, examples, and API reference

### I.3.6 CI/CD

- GitHub Actions workflows for testing, linting, and type checking
- Publish workflow for PyPI deployment
- Matrix testing across Python versions

## I.4 customisation Options

The template offers several customisation options during generation:

### I.4.1 Basic vs. Advanced Setup

- **Basic:** Lighter configuration focused on essential tools
- **Advanced:** Full suite of tools including security scanning, stricter type checking, and comprehensive CI/CD

### I.4.2 Documentation Options

- Choose whether to include MkDocs documentation setup
- If included, get a complete documentation structure ready for content

### I.4.3 CI/CD Options

- Include GitHub Actions workflows for automated testing and deployment
- Configure publishing workflows for PyPI integration

## I.5 Template Structure

The generated project follows this structure:

```

your_project/
 .github/
 workflows/
 ci.yml
 publish.yml
 src/
 your_package/
 __init__.py
 main.py
 tests/
 __init__.py
 test_main.py
 docs/
 index.md
 examples.md
 .gitignore
 LICENSE
 README.md
 requirements.in
 (human-maintained)
 requirements-dev.in
 pyproject.toml
GitHub specific configuration
GitHub Actions workflows
Continuous Integration workflow
Package publishing workflow
Main source code directory
Actual Python package
Makes the directory a package
Example module
Test suite
Makes tests importable
Tests for main.py
Documentation
Main documentation page
Example usage
Files to exclude from Git
License file
Project overview
Direct dependencies
Development dependencies
Project & tool configuration

```

## I.6 Post-Generation Steps

After creating your project, the template provides instructions for:

1. Creating and activating a virtual environment
2. Installing dependencies
3. Setting up version control
4. Running initial tests

The generated `README.md` includes detailed development setup instructions specific to your configuration choices.

## I.7 Extending the Template

You can extend or customise the template for your specific needs:

### I.7.1 Adding Custom Components

Fork the template repository and add additional files or configurations specific to your organisation or preferences.

### I.7.2 Modifying Tool Configurations

The `pyproject.toml` file contains all tool configurations and can be adjusted to match your coding standards and preferences.

### I.7.3 Creating Specialized Variants

Create specialized variants of the template for different types of projects (e.g., web applications, data science, CLI tools) while maintaining the core best practices.

## I.8 Best Practices for Using the Template

1. **Use for new projects:** The template is designed for new projects rather than retrofitting existing ones.
2. **Commit immediately after generation:** Make an initial commit right after generating the project to establish a clean baseline.
3. **Review and adjust configurations:** While the defaults are sensible, review and adjust configurations to match your specific project needs.
4. **Keep dependencies updated:** Regularly update the `requirements.in` files as your project evolves.
5. **Follow the workflow:** The template sets up the infrastructure, but you still need to follow the development workflow described in this book.

## I.9 Conclusion

The Python Development Pipeline cookiecutter template encapsulates the practices and principles discussed throughout this book, allowing you to rapidly bootstrap projects with best practices already in place. By using this template, you ensure consistency across projects and can focus more on solving problems rather than setting up infrastructure.

Whether you're starting a small personal project or a larger team effort, this template provides a solid foundation that can scale with your needs while maintaining professional development standards.



# Appendix J

## Hatch - Modern Python Project Management

### J.1 Introduction to Hatch

Hatch is a modern, extensible Python project management tool designed to simplify the development workflow through standardization and automation. Created by Ofek Lev and first released in 2017, Hatch has undergone significant evolution to become a comprehensive solution that handles environment management, dependency resolution, building, and publishing.

Unlike traditional tools that focus primarily on packaging or dependency management, Hatch takes a holistic approach to project management, addressing the entire development lifecycle. What sets Hatch apart is its flexibility, extensibility, and focus on developer experience through an intuitive CLI and plugin system.

### J.2 Key Features of Hatch

#### J.2.1 Project Management

Hatch provides comprehensive project management capabilities:

- **Project initialization:** Quickly set up standardized project structures
- **Flexible configuration:** Standardized configuration in `pyproject.toml`
- **Version management:** Easily bumper version numbers
- **Script running:** Execute defined project scripts

#### J.2.2 Environment Management

One of Hatch's standout features is its sophisticated environment handling:

- **Multiple environments per project:** Define development, testing, documentation environments
- **Matrix environments:** Test across Python versions and dependency sets
- **Isolated environments:** Clean, reproducible development spaces
- **Environment synchronization:** Keep environments updated

### J.2.3 Build and Packaging

Hatch streamlines the packaging workflow:

- **Standards-compliant:** Implements PEP 517/518 build system
- **Multiple build targets:** Source distributions and wheels
- **Build hooks:** customise the build process
- **Metadata standardization:** PEP 621 compliant metadata

### J.2.4 Extensibility

Hatch is designed for extensibility:

- **Plugin system:** Extend functionality through plugins
- **Custom commands:** Add project-specific commands
- **Environment customisation:** Define environment-specific tools
- **Build customisation:** Extend the build process

## J.3 Getting Started with Hatch

### J.3.1 Installation

Hatch can be installed through several methods:

```
Using pipx (recommended)
pipx install hatch

Using pip
pip install hatch

Using conda
conda install -c conda-forge hatch

Using Homebrew on macOS
brew install hatch
```

Verify your installation:

```
hatch --version
```

### J.3.2 Creating a New Project

Create a new project with Hatch:

```
Interactive project creation
hatch new

Non-interactive with defaults
hatch new my-project

With specific options
hatch new my-project --init
```

The project structure might look like:

```
my-project/
 src/
 my_project/
 __init__.py
 tests/
 __init__.py
 pyproject.toml
 README.md
```

### J.3.3 Basic Configuration

Hatch uses `pyproject.toml` for configuration:

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "my-project"
version = "0.1.0"
description = "A sample Python project"
readme = "README.md"
requires-python = ">=3.8"
license = {text = "MIT"}
authors = [
 {name = "Your Name", email = "your.email@example.com"},
]
dependencies = [
 "requests>=2.28.0",
 "pydantic>=2.0.0",
]

[project.optional-dependencies]
test = [
 "pytest>=7.0.0",
 "pytest-cov>=4.0.0",
]
dev = [
 "black>=23.0.0",
 "ruff>=0.0.220",
]

[tool.hatch.envs.default]
dependencies = [
 "pytest>=7.0.0",
 "black>=23.0.0",
 "ruff>=0.0.220",
]

[tool.hatch.envs.test]
dependencies = [
 "pytest>=7.0.0",
 "black>=23.0.0",
 "ruff>=0.0.220",
]
```

```
"pytest>=7.0.0",
"pytest-cov>=4.0.0",
]
```

## J.4 Essential Hatch Commands

### J.4.1 Environment Management

```
Create and activate the default environment
hatch shell

Create and activate a specific environment
hatch shell test

Run a command in the default environment
hatch run pytest

Run a command in a specific environment
hatch run test:pytest

List available environments
hatch env show

Clean all environments
hatch env prune
```

### J.4.2 Dependency Management

```
Install project dependencies
hatch env create

Update all dependencies
hatch env update

Update dependencies in a specific environment
hatch env update test

Show installed packages
hatch env show
```

### J.4.3 Building and Publishing

```
Build the package
hatch build

Build specific formats
hatch build -t wheel
```

```
Publish to PyPI
hatch publish

Publish to TestPyPI
hatch publish -r test
```

#### J.4.4 Version Management

```
Show current version
hatch version

Bump the version (patch, minor, major)
hatch version patch
hatch version minor
hatch version major

Set a specific version
hatch version 1.2.3
```

## J.5 Advanced Hatch Features

### J.5.1 Environment Matrix

Hatch can manage testing across multiple Python versions:

```
[tool.hatch.envs.test]
dependencies = [
 "pytest",
]

[[tool.hatch.envs.test.matrix]]
python = ["3.8", "3.9", "3.10", "3.11"]
```

Run commands across all environments:

```
Run tests across all Python versions
hatch run test:all:pytest
```

### J.5.2 Custom Scripts

Define project-specific scripts:

```
[tool.hatch.envs.default.scripts]
test = "pytest"
lint = "ruff check ."
format = "black ."

Complex scripts
dev = [
 "format",
```

```

 "lint",
 "test",
]

```

Run these scripts:

```

Run the test script
hatch run test

Run the complete dev script
hatch run dev

```

### J.5.3 Environment Features

Enable specific tools in environments:

```

[tool.hatch.envs.default]
features = ["dev", "test"]
dependencies = [
 "black",
 "pytest",
]

[tool.hatch.envs.default.scripts]
test = "pytest {args}"
format = "black {args:src tests}"

```

### J.5.4 Build Hooks

customise the build process:

```

[tool.hatch.build.hooks.vcs]
version-file = "src/my_project/_version.py"

[tool.hatch.build.hooks.custom]
path = "my_custom_build_hook.py"

```

## J.6 Best Practices with Hatch

### J.6.1 Project Structure

A recommended structure for Hatch projects:

```

my_project/
 src/
 my_package/ # Main package code
 __init__.py
 module.py
 tests/ # Test files
 __init__.py
 test_module.py

```

```
docs/ # Documentation
pyproject.toml # Project configuration
README.md # Project documentation
```

To use this source layout:

```
[tool.hatch.build]
packages = ["src/my_package"]
```

## J.6.2 Environment Management Strategies

1. **Specialized Environments:** Create purpose-specific environments

```
[tool.hatch.envs.default]
dependencies = ["pytest", "black", "ruff"]

[tool.hatch.envs.docs]
dependencies = ["sphinx", "sphinx-rtd-theme"]

[tool.hatch.envs.security]
dependencies = ["bandit", "safety"]
```

2. **Matrix Testing:** Test across Python versions

```
[[tool.hatch.envs.test.matrix]]
python = ["3.8", "3.9", "3.10", "3.11"]
```

3. **Feature Toggles:** organise functionality by feature

```
[tool.hatch.envs.default]
features = ["test", "lint"]
```

## J.6.3 Version Control Practices

1. **Configure version source:** Use git tags or a version file

```
[tool.hatch.version]
source = "vcs" # or "file"
```

2. **Automate version bumping:** Use Hatch's version commands in your workflow

```
Before release
hatch version minor
git commit -am "Bump version to $(hatch version)"
git tag v$(hatch version)
```

## J.6.4 Integration with Development Tools

Configure tools like Black and Ruff directly in `pyproject.toml`:

```
[tool.black]
line-length = 88
target-version = ["py39"]
```

```
[tool.ruff]
select = ["E", "F", "I"]
line-length = 88
```

## J.7 Integration with Development Workflows

### J.7.1 IDE Integration

Hatch environments work with most Python IDEs:

#### J.7.1.1 VS Code

1. Create environments: `hatch env create`
2. Find the environment path: `hatch env find default`
3. Select the interpreter from this path in VS Code

#### J.7.1.2 PyCharm

1. Create environments: `hatch env create`
2. Find the environment path: `hatch env find default`
3. Add the interpreter in PyCharm settings

### J.7.2 CI/CD Integration

#### J.7.2.1 GitHub Actions Example

```
name: Python CI

on:
 push:
 branches: [main]
 pull_request:
 branches: [main]

jobs:
 test:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3

 - name: Set up Python
 uses: actions/setup-python@v4
 with:
 python-version: "3.10"

 - name: Install Hatch
 run: pip install hatch

 - name: Run tests
 run: hatch run test:pytest
```

```
- name: Run linters
 run: hatch run lint:all
```

## J.8 Troubleshooting Common Issues

### J.8.1 Environment Creation Failures

If environments fail to create:

```
Show detailed errors
hatch env create -v

Try creating with verbose output
hatch -v env create

Check for conflicting dependencies
hatch dep show
```

### J.8.2 Build Issues

For build-related problems:

```
Verbose build output
hatch build -v

Clean build artifacts
hatch clean

Check configuration
hatch project metadata
```

### J.8.3 Plugin Problems

If plugins aren't working:

```
List installed plugins
hatch plugin list

Update plugins
pip install -U hatch-plugin-name
```

## J.9 Comparison with Other Tools

### J.9.1 Hatch vs. Poetry

- **Hatch:** More flexible, multiple environments, standards-focused
- **Poetry:** More opinionated, stronger dependency resolution
- **Key difference:** Hatch's multiple environments per project vs. Poetry's single environment approach

## J.9.2 Hatch vs. PDM

- **Hatch:** Focus on the entire development workflow
- **PDM:** Stronger focus on dependency management with PEP 582 support
- **Key difference:** Hatch's broader scope vs. PDM's emphasis on dependencies

## J.9.3 Hatch vs. pip + venv

- **Hatch:** Integrated environment and project management
- **pip + venv:** Separate tools requiring manual coordination
- **Key difference:** Hatch's automation vs. traditional manual approach

## J.10 When to Use Hatch

Hatch is particularly well-suited for:

1. **Complex Development Workflows:** Multiple environments, testing matrices
2. **Teams with Diverse Projects:** Standardization across different project types
3. **Open Source Maintainers:** Multiple environment testing and streamlined releases
4. **Projects Requiring customisation:** Plugin system for specialized needs

Hatch might not be ideal for:

1. **Very Simple Scripts:** Might be overkill for trivial projects
2. **Teams Heavily Invested in Poetry:** Migration costs might outweigh benefits
3. **Projects with Unusual Build Systems:** Some specialized build needs might require additional customisation

## J.11 Conclusion

Hatch represents a modern approach to Python project management that emphasizes flexibility, standards compliance, and developer experience. Its unique multi-environment capabilities, combined with comprehensive project lifecycle management, make it a powerful choice for both application and library development.

While newer than some alternatives like Poetry, Hatch's strict adherence to Python packaging standards ensures compatibility with the broader ecosystem. Its plugin system and flexible configuration options allow it to adapt to a wide range of project needs, from simple libraries to complex applications.

For developers looking for a tool that can grow with their projects and adapt to various workflows, Hatch provides a compelling combination of power and flexibility. Its focus on standardization and automation helps reduce the cognitive overhead of project management, allowing developers to focus more on writing code and less on managing tooling.

## Appendix K

# Using Conda for Environment Management

### K.1 Introduction to Conda

Conda is a powerful open-source package and environment management system that runs on Windows, macOS, and Linux. While similar to the virtual environment tools covered in the main text, conda offers distinct advantages for certain Python workflows, particularly in data science, scientific computing, and research domains.

Unlike tools that focus solely on Python packages, conda can package and distribute software for any language, making it especially valuable for projects with complex dependencies that extend beyond the Python ecosystem.

### K.2 When to Consider Conda

Conda is particularly well-suited for:

- **Data science projects** requiring scientific packages (NumPy, pandas, scikit-learn, etc.)
- **Research environments** with mixed-language requirements (Python, R, C/C++ libraries)
- **Projects with complex binary dependencies** that are difficult to compile
- **Cross-platform development** where consistent environments across operating systems are crucial
- **GPU-accelerated computing** requiring specific CUDA versions
- **Bioinformatics, computational physics, and other specialized scientific domains**

### K.3 Conda vs. Other Environment Tools

Feature	Conda	venv + pip	uv
<b>Focus</b>	Any language packages	Python packages	Python packages
<b>Binary package distribution</b>	Yes (pre-compiled)	Limited	Limited
<b>Dependency resolution</b>	Environment-level solver	Package-level solver	Fast, improved solver
<b>Platform support</b>	Windows, macOS, Linux	Windows, macOS, Linux	Windows, macOS, Linux
<b>Non-Python dependencies</b>	Excellent	Limited	Limited
<b>Speed</b>	Moderate	Moderate	Very fast
<b>Scientific package support</b>	Excellent	Good	Good

## K.4 Getting Started with Conda

### K.4.1 Installation

Conda is available through several distributions:

1. **Miniconda**: Minimal installer containing just conda and its dependencies
2. **Anaconda**: Full distribution including conda and 250+ popular data science packages

For most development purposes, Miniconda is recommended as it provides a minimal base that you can build upon as needed.

To install Miniconda:

```
Linux
wget
https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh

macOS
wget
https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.sh
bash Miniconda3-latest-MacOSX-x86_64.sh

Windows
Download the installer from
https://docs.conda.io/en/latest/miniconda.html
and run it
```

## K.4.2 Basic Conda Commands

### K.4.2.1 Creating Environments

```
Create a new environment with Python 3.10
conda create --name myenv python=3.10

Create environment with specific packages
conda create --name datasci python=3.10 numpy pandas matplotlib

Create environment from file
conda env create --file environment.yml
```

### K.4.2.2 Activating and Deactivating Environments

```
Activate an environment
conda activate myenv

Deactivate current environment
conda deactivate
```

### K.4.2.3 Managing Packages

```
Install packages
conda install numpy pandas

Install from specific channel
conda install -c conda-forge scikit-learn

Update packages
conda update numpy

Remove packages
conda remove pandas

List installed packages
conda list
```

### K.4.2.4 Environment Management

```
List all environments
conda env list

Remove an environment
conda env remove --name myenv

Export environment to file
conda env export > environment.yml

Clone an environment
```

```
conda create --name newenv --clone oldenv
```

## K.5 Environment Files with Conda

Conda uses YAML files to define environments, making them easily shareable and reproducible:

```
environment.yml
name: datasci
channels:
 - conda-forge
 - defaults
dependencies:
 - python=3.10
 - numpy=1.23
 - pandas>=1.4
 - matplotlib
 - scikit-learn
 - pip
 - pip:
 - some-package-only-on-pypi
```

This file defines: - The environment name (`datasci`) - Channels to search for packages (with preference order) - Conda packages with optional version constraints - Additional pip packages to install

Create this environment with:

```
conda env create -f environment.yml
```

## K.6 Best Practices for Conda

### K.6.1 Channel Management

Conda packages come from “channels.” The main ones are:

- **defaults:** Official Anaconda channel
- **conda-forge:** Community-led channel with more up-to-date packages

For consistent environments, specify channels explicitly in your environment files and consider adding channel priority:

```
channels:
 - conda-forge
 - defaults
```

This prioritizes conda-forge packages over defaults when both are available.

### K.6.2 minimising Environment Size

Conda environments can become large. Keep them streamlined by:

1. Only installing what you need

2. Using the `--no-deps` flag when appropriate
3. Considering a minimal base environment with `conda create --name myenv python`

### K.6.3 Managing Conflicting Dependencies

When facing difficult dependency conflicts:

```
Create environment with strict solver
conda create --name myenv python=3.10 --strict-channel-priority

Or use the libmamba solver for better resolution
conda install -n base conda-libmamba-solver
conda create --name myenv python=3.10 --solver=libmamba
```

### K.6.4 Combining Conda with pip

While conda can install most packages, some are only available on PyPI. The recommended approach:

1. Install all conda-available packages first using conda
2. Then install PyPI-only packages using pip

This approach is implemented automatically when using an `environment.yml` file with a `pip` section.

### K.6.5 Environment Isolation from System Python

Avoid using your system Python installation with conda. Instead:

```
Explicitly create all environments with a specific Python version
conda create --name myenv python=3.10
```

## K.7 Integration with Development Workflows

### K.7.1 Using Conda with VS Code

VS Code can automatically detect and use conda environments:

1. Install the Python extension
2. Open the Command Palette (Ctrl+Shift+P)
3. Select “Python: Select Interpreter”
4. Choose your conda environment from the list

### K.7.2 Using Conda with Jupyter

Conda integrates well with Jupyter notebooks:

```
Install Jupyter in your environment
conda install -c conda-forge jupyter

Register your conda environment as a Jupyter kernel
conda install -c conda-forge ipykernel
```

```
python -m ipykernel install --user --name=myenv --display-name="Python
(myenv)"
```

### K.7.3 CI/CD with Conda

For GitHub Actions, you can use conda environments:

```
name: Python CI with Conda

on: [push, pull_request]

jobs:
 build:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3
 - name: Set up conda
 uses: conda-incubator/setup-miniconda@v2
 with:
 python-version: 3.10
 environment-file: environment.yml
 auto-activate-base: false
 - name: Run tests
 shell: bash -l {0}
 run: |
 conda activate myenv
 pytest
```

## K.8 Common Pitfalls and Solutions

### K.8.1 Slow Environment Creation

Conda environments can take time to create due to dependency resolution:

```
Use the faster libmamba solver
conda install -n base conda-libmamba-solver
conda create --name myenv python=3.10 numpy pandas --solver=libmamba
```

### K.8.2 Conflicting Channels

Mixing packages from different channels can cause conflicts:

```
Use strict channel priority
conda config --set channel_priority strict
```

### K.8.3 Large Environment Sizes

Conda environments can grow large, especially with the Anaconda distribution:

```
Start minimal and add only what you need
conda create --name myenv python=3.10
```

```
conda install -n myenv numpy pandas

Or use mamba for more efficient installations
conda install -c conda-forge mamba
mamba create --name myenv python=3.10 numpy pandas
```

## K.9 Mamba: A Faster Alternative

For large or complex environments, consider mamba, a reimplementa-tion of conda’s package manager in C++:

```
Install mamba
conda install -c conda-forge mamba

Use mamba with the same syntax as conda
mamba create --name myenv python=3.10 numpy pandas
mamba install -n myenv scikit-learn
```

Mamba offers significant speed improvements for environment creation and package installation while maintaining compatibility with conda commands.

## K.10 Conclusion

Conda provides a robust solution for environment management, particularly valuable for scientific computing, data science, and research applications. While more complex than venv, it solves specific problems that other tools cannot easily address, especially when dealing with non-Python dependencies or cross-platform binary distribution.

For projects focusing purely on Python dependencies without complex binary requirements, the venv and uv approaches covered in the main text may provide simpler workflows. However, understanding conda remains valuable for many Python practitioners, especially those working in scientific domains.



# Appendix L

## Getting Started with venv

### L.1 Introduction to venv

The `venv` module is Python’s built-in tool for creating virtual environments. Introduced in Python 3.3 and standardized in PEP 405, it has become the official recommended way to create isolated Python environments. As a module in the standard library, `venv` is immediately available with any Python installation, requiring no additional installation step.

Virtual environments created with `venv` provide isolated spaces where Python projects can have their own dependencies, regardless of what dependencies other projects may have. This solves the common problem of conflicting package requirements across different projects and prevents changes to one project from affecting others.

### L.2 Why Use venv?

Virtual environments are essential in Python development for several reasons:

1. **Dependency Isolation:** Each project can have its own dependencies, regardless of other projects’ requirements
2. **Consistent Environments:** Ensures reproducible development and deployment environments
3. **Clean Testing:** Test against specific package versions without affecting the system Python
4. **Conflict Prevention:** Avoids “dependency hell” where different projects need different versions of the same package
5. **Project organisation:** Clearly separates project dependencies from system or global packages

### L.3 Getting Started with venv

#### L.3.1 Creating a Virtual Environment

To create a virtual environment using `venv`, open a terminal and run:

```
Basic syntax
python -m venv /path/to/new/virtual/environment

Common usage (create a .venv directory in your project)
python -m venv .venv
```

The command creates a directory containing: - A Python interpreter copy - The `pip` package manager - A basic set of installed libraries - Scripts to activate the environment

## L.3.2 Activating the Environment

Before using the virtual environment, you need to activate it. The activation process adjusts your shell's `PATH` to prioritize the virtual environment's Python interpreter and tools.

### L.3.2.1 On Windows:

```
Command Prompt
.venv\Scripts\activate.bat

PowerShell
.venv\Scripts\Activate.ps1
```

### L.3.2.2 On macOS and Linux:

```
source .venv/bin/activate
```

After activation, your shell prompt typically changes to indicate the active environment:

```
(.venv) user@computer:~/project$
```

All Python and `pip` commands now use the virtual environment's versions instead of the system ones.

## L.3.3 Deactivating the Environment

When you're done working on the project, deactivate the environment:

```
deactivate
```

This restores your shell to its original state, using the system Python interpreter.

## L.4 Advanced venv Options

### L.4.1 Creating Environments with Specific Python Versions

To create an environment with a specific Python version, use that version's interpreter:

```
Using Python 3.8
python3.8 -m venv .venv
```

```
On Windows with py launcher
py -3.8 -m venv .venv
```

### L.4.2 Creating Environments Without pip

By default, `venv` installs `pip` in new environments. To create one without `pip`:

```
python -m venv --without-pip .venv
```

### L.4.3 Creating System Site-packages Access

Normally, virtual environments are isolated from system site-packages. To allow access:

```
python -m venv --system-site-packages .venv
```

This creates an environment that can see system packages, but newly installed packages still go into the virtual environment.

### L.4.4 Upgrading pip in a New Environment

Virtual environments often include an older `pip` version. It's good practice to upgrade:

```
After activating the environment
pip install --upgrade pip
```

## L.5 Managing Dependencies with venv

While `venv` creates the environment, you'll use `pip` to manage packages within it.

### L.5.1 Installing Packages

With your environment activated:

```
Install individual packages
pip install requests

Install with version constraints
pip install "django>=4.0,<5.0"
```

### L.5.2 Tracking Dependencies

To track installed packages:

```
Generate a requirements file
pip freeze > requirements.txt
```

This creates a text file listing all installed packages and their versions.

### L.5.3 Installing from Requirements

To recreate an environment elsewhere:

```
Create and activate a new environment
python -m venv .venv
source .venv/bin/activate # or Windows equivalent

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

## L.6 Best Practices with venv

### L.6.1 Directory Naming Conventions

Common virtual environment directory names include:

- `.venv`: Hidden directory (less visible clutter)
- `venv`: Explicit directory name
- `env`: Shorter alternative

The `.venv` name is increasingly popular as it: - Keeps it hidden in file browsers - Makes it easy to add to `.gitignore` - Is recognised by many IDEs and tools

### L.6.2 Version Control Integration

Never commit virtual environment directories to version control. Add them to `.gitignore`:

```
.gitignore
.venv/
venv/
env/
```

### L.6.3 Environment Management Across Projects

Create a new virtual environment for each project:

```
Project A
cd project_a
python -m venv .venv

Project B
cd ../project_b
python -m venv .venv
```

### L.6.4 IDE Integration

Most Python IDEs integrate well with venv environments:

#### L.6.4.1 VS Code

1. Open your project folder
2. Press `Ctrl+Shift+P`
3. Select “Python: Select Interpreter”
4. Choose the environment from the list

### L.6.4.2 PyCharm

1. Go to Settings → Project → Python Interpreter
2. Click the gear icon → Add
3. Select “Existing Environment” and navigate to the environment’s Python

## L.7 Comparing venv with Other Tools

### L.7.1 venv vs. virtualenv

`virtualenv` is a third-party package that inspired the creation of `venv`.

- **venv**: Built into Python, no installation needed, slightly fewer features
- **virtualenv**: Third-party package, more features, better backwards compatibility

For most modern Python projects, `venv` is sufficient, but `virtualenv` offers some advanced options and supports older Python versions.

### L.7.2 venv vs. conda

While both create isolated environments, they serve different purposes:

- **venv**: Python-specific, lightweight, manages only Python packages
- **conda**: Cross-language package manager, handles non-Python dependencies, preferred for scientific computing

### L.7.3 venv vs. Poetry/PDM

These are newer tools that combine dependency management with virtual environments:

- **venv+pip**: Separate tools for environments and package management
- **Poetry/PDM**: All-in-one solutions with lock files, dependency resolution, packaging

## L.8 Troubleshooting Common Issues

### L.8.1 Activation Script Not Found

If you can’t find the activation script:

```
List environment directory contents
ls -la .venv/bin # macOS/Linux
dir .venv\Scripts # Windows
```

Make sure the environment was created successfully and you’re using the correct path.

### L.8.2 Packages Not Found After Installation

If packages are installed but not importable:

1. Verify the environment is activated (check prompt prefix)
2. Check if you have multiple Python installations
3. Reinstall the package in the active environment

### L.8.3 Permission Issues

If you encounter permission errors:

```
On macOS/Linux
python -m venv --prompt myproject .venv

On Windows, try running as administrator or using user directory
```

## L.9 Script Examples for venv Workflows

### L.9.1 Project Setup Script

```
#!/bin/bash
setup_project.sh

Create project directory
mkdir -p my_project
cd my_project

Create basic structure
mkdir -p src/my_package tests docs

Create virtual environment
python -m venv .venv

Activate environment (adjust for your shell)
source .venv/bin/activate

Upgrade pip
pip install --upgrade pip

Install initial dev packages
pip install pytest black

Create initial requirements
pip freeze > requirements.txt

echo "Project setup complete! Activate with: source .venv/bin/activate"
```

### L.9.2 Environment Recreation Script

```
#!/bin/bash
recreate_env.sh

Remove old environment if it exists
rm -rf .venv

Create fresh environment
python -m venv .venv
```

```
Activate
source .venv/bin/activate

Upgrade pip
pip install --upgrade pip

Install dependencies
pip install -r requirements.txt

echo "Environment recreated successfully!"
```

## L.10 Conclusion

The `venv` module provides a simple, reliable way to create isolated Python environments directly from the standard library. While newer tools offer more features and automation, `venv` remains a fundamental building block of Python development workflows, offering an excellent balance of simplicity and utility.

For most Python projects, the combination of `venv` and `pip` provides a solid foundation for environment management. As projects grow in complexity, you can build upon this foundation with additional tools while maintaining the same core principles of isolation and reproducibility.



# Appendix M

## UV - High-Performance Python Package Management

### M.1 Introduction to uv

uv is a modern, high-performance Python package installer and resolver written in Rust. Developed by Astral, it represents a significant evolution in Python tooling, designed to address the performance limitations of traditional Python package management tools while maintaining compatibility with the existing Python packaging ecosystem.

Unlike older tools that are written in Python itself, uv's implementation in Rust gives it exceptional speed advantages—often 10-100x faster than traditional tools for common operations. This performance boost is particularly noticeable in larger projects with complex dependency graphs.

### M.2 Key Features and Benefits

#### M.2.1 Performance

Performance is uv's most distinctive feature:

- **Parallel Downloads:** Downloads and installs packages in parallel
- **optimised Dependency Resolution:** Efficiently resolves dependencies with a modern algorithm
- **Cached Builds:** Maintains a build artifact cache to avoid redundant work
- **Rust Implementation:** Low memory usage and high computational efficiency

In practical terms, this means environments that might take minutes to create with traditional tools can be ready in seconds with uv.

#### M.2.2 Compatibility

Despite its modern architecture, uv maintains compatibility with Python's ecosystem:

- **Standard Wheel Support:** Installs standard Python wheel distributions
- **PEP Compliance:** Follows relevant Python Enhancement Proposals for packaging
- **Requirements.txt Support:** Works with traditional requirements files

- **pyproject.toml Support:** Compatible with modern project configurations

### M.2.3 Unified Functionality

uv combines features from several traditional tools:

- **Environment Management:** Similar to venv but faster
- **Package Installation:** Like pip but with parallel processing
- **Dependency Resolution:** Similar to pip-tools but more efficient
- **Lockfile Generation:** Creates deterministic environments like pip-compile

## M.3 Getting Started with uv

### M.3.1 Installation

uv can be installed in several ways:

```
Using pipx (recommended for CLI usage)
pipx install uv

Using pip
pip install uv

Using curl (Unix systems)
curl -LsSf https://astral.sh/uv/install.sh | sh

Using PowerShell (Windows)
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"
```

### M.3.2 Basic Commands

uv has an intuitive command structure that will feel familiar to pip users:

```
Create a virtual environment
uv venv

Install a package
uv pip install requests

Install from requirements file
uv pip install -r requirements.txt

Install a package in development mode
uv pip install -e .
```

### M.3.3 Working with Virtual Environments

uv integrates environment management with package installation:

```
Create and activate a virtual environment
uv venv
source .venv/bin/activate # On Unix
```

```
.venv\Scripts\activate # On Windows

Or install directly into an environment
uv pip install --venv .venv numpy pandas
```

## M.4 Dependency Management with uv

### M.4.1 Compiling Requirements

uv offers an efficient workflow for managing dependencies using a two-file approach similar to pip-tools:

```
Create a simple requirements.in file
echo "requests>=2.28.0" > requirements.in

Compile to a locked requirements.txt
uv pip compile requirements.in -o requirements.txt

Install the locked dependencies
uv pip sync requirements.txt
```

The generated requirements.txt will contain exact versions of all dependencies (including transitive ones), ensuring reproducible environments.

### M.4.2 Development Dependencies

For more complex projects, you can separate production and development dependencies:

```
Create a dev-requirements.in file
echo "-c requirements.txt" > dev-requirements.in
echo "pytest" >> dev-requirements.in
echo "black" >> dev-requirements.in

Compile development dependencies
uv pip compile dev-requirements.in -o dev-requirements.txt

Install all dependencies
uv pip sync requirements.txt dev-requirements.txt
```

The `-c requirements.txt` constraint ensures compatible versions between production and development dependencies.

### M.4.3 Updating Dependencies

When you need to update packages:

```
Update all packages to their latest allowed versions
uv pip compile --upgrade requirements.in

Update a specific package
uv pip compile --upgrade-package requests requirements.in
```

## M.5 Advanced uv Features

### M.5.1 Offline Mode

uv supports working in environments without internet access:

```
Install using only cached packages
uv pip install --offline numpy
```

### M.5.2 Direct URLs and Git Dependencies

uv can install packages from various sources:

```
Install from GitHub
uv pip install git+https://github.com/user/repo.git@branch

Install from local directory
uv pip install /path/to/local/package
```

### M.5.3 Configuration Options

uv allows configuration through command-line options:

```
Set global options
uv pip install --no-binary :all: numpy # Force source builds
uv pip install --only-binary numpy pandas # Force binary installations
```

### M.5.4 Performance optimisation

To maximise uv's performance:

```
Use concurrent installations
uv pip install --concurrent-installs numpy pandas matplotlib

Reuse the build environment
uv pip install --no-build-isolation package-name
```

## M.6 Integration with Workflows

### M.6.1 CI/CD Integration

uv is particularly valuable in CI/CD pipelines where speed matters:

```
GitHub Actions example
- name: Set up Python
 uses: actions/setup-python@v4
 with:
 python-version: "3.10"

- name: Install uv
 run: pip install uv
```

```
- name: Install dependencies
 run: uv pip sync requirements.txt dev-requirements.txt
```

## M.6.2 IDE Integration

While IDEs typically detect standard virtual environments, you can explicitly configure them:

### M.6.2.1 VS Code

1. Create an environment: `uv venv`
2. Select the interpreter at `.venv/bin/python` (Unix) or `.venv\Scripts\python.exe` (Windows)

### M.6.2.2 PyCharm

1. Create an environment: `uv venv`
2. In Settings → Project → Python Interpreter, add the interpreter from the `.venv` directory

## M.7 Comparing uv with Other Tools

### M.7.1 uv vs. pip

Feature	uv	pip
<b>Installation Speed</b>	Very fast (parallel)	Slower (sequential)
<b>Dependency Resolution</b>	Fast, efficient	Slower, sometimes problematic
<b>Environment Management</b>	Built-in	Requires separate tool (venv)
<b>Lock Files</b>	Native support	Requires pip-tools
<b>Caching</b>	Global, efficient	More limited
<b>Compatibility</b>	High with standard packages	Universal

### M.7.2 uv vs. pip-tools

Feature	uv	pip-tools
<b>Speed</b>	Very fast	Moderate
<b>Implementation</b>	Rust	Python
<b>Environment Management</b>	Integrated	Separate (needs venv)
<b>Command Structure</b>	<code>uv pip compile/sync</code>	<code>pip-compile/pip-sync</code>
<b>Hash Generation</b>	Supported	Supported

### M.7.3 uv vs. Poetry/PDM

Feature	uv	Poetry/PDM
<b>Focus</b>	Performance	Project management
<b>Configuration</b>	Minimal (uses standard files)	More extensive
<b>Learning Curve</b>	Gentle (similar to pip)	Steeper
<b>Project Structure</b>	Flexible	More opinionated
<b>Publishing to PyPI</b>	Basic support	Comprehensive support

## M.8 Best Practices with uv

### M.8.1 Dependency Management Workflow

A recommended workflow using uv for dependency management:

1. **Define direct dependencies** in a `requirements.in` file with minimal version constraints
2. **Compile locked requirements** with `uv pip compile requirements.in -o requirements.txt`
3. **Install dependencies** with `uv pip sync requirements.txt`
4. **Update dependencies** periodically with `uv pip compile --upgrade requirements.in`

### M.8.2 Optimal Project Structure

A simple project structure that works well with uv:

```
my_project/
 .venv/ # Created by uv venv
 src/ # Source code
 my_package/
 tests/ # Test files
 requirements.in # Direct dependencies
 requirements.txt # Locked dependencies (generated)
 dev-requirements.in # Development dependencies
 dev-requirements.txt # Locked dev dependencies (generated)
 pyproject.toml # Project configuration
```

### M.8.3 Version Control Considerations

When using version control with uv:

- **Commit both `.in` and `.txt` files** to ensure reproducible builds
- **Add `.venv/` to your `.gitignore`**
- **Consider committing hash-verified requirements** for security

## M.9 Troubleshooting uv

### M.9.1 Common Issues and Solutions

#### M.9.1.1 Missing Binary Wheels

If you encounter issues with packages requiring compilation:

```
Try forcing binary wheels
uv pip install --only-binary :all: package-name

Or for a specific package
uv pip install --only-binary package-name package-name
```

#### M.9.1.2 Dependency Conflicts

For dependency resolution issues:

```
Get detailed information about conflicts
uv pip install --verbose package-name

Try installing with more permissive constraints
uv pip install --no-deps package-name
Then fix specific dependencies
```

#### M.9.1.3 Environment Problems

If environments aren't working properly:

```
Create a fresh environment
rm -rf .venv
uv venv

Or use a specific Python version
uv venv --python 3.9
```

## M.10 Conclusion

uv represents an exciting advancement in Python tooling, offering significant performance improvements while maintaining compatibility with existing workflows. Its speed benefits are particularly valuable for:

- CI/CD pipelines where build time matters
- Large projects with many dependencies
- Development environments with frequent updates
- Teams looking to improve developer experience

While newer than some traditional tools, uv's compatibility with standard Python packaging conventions makes it a relatively low-risk adoption with potentially high rewards in terms of productivity and performance. As it continues to mature, uv is positioned to become an increasingly important part of the Python development ecosystem.

For most projects, uv can be a drop-in replacement for pip and pip-tools, offering an immediate performance boost without requiring significant workflow changes—a rare combination of revolutionary performance with evolutionary adoption requirements.

# Appendix N

## Poetry - Modern Python Packaging and Dependency Management

### N.1 Introduction to Poetry

Poetry is a modern Python package management tool designed to simplify dependency management and packaging in Python projects. Developed by Sébastien Eustace and released in 2018, Poetry aims to solve common problems in the Python ecosystem by providing a single tool to handle dependency installation, package building, and publishing.

Poetry's core philosophy is to make Python packaging more deterministic and user-friendly through declarative dependency specification, lock files for reproducible environments, and simplified commands for common workflows. By combining capabilities that traditionally required multiple tools (pip, setuptools, twine, etc.), Poetry offers a more cohesive development experience.

### N.2 Key Features of Poetry

#### N.2.1 Dependency Management

Poetry's dependency resolution is one of its strongest features:

- **Deterministic builds:** Poetry resolves dependencies considering the entire dependency graph, preventing many common conflicts
- **Lock file:** The `poetry.lock` file ensures consistent installations across different environments
- **Easy version specification:** Simple syntax for version constraints
- **Dependency groups:** organise dependencies into development, testing, and other logical groups

## N.2.2 Project Setup and Configuration

Poetry uses a single configuration file for project metadata and dependencies:

- **pyproject.toml:** All project configuration lives in one standard-compliant file
- **Project scaffolding:** `poetry new` command creates a standardized project structure
- **Environment management:** Automatic handling of virtual environments

## N.2.3 Build and Publish Workflow

Poetry streamlines the package distribution process:

- **Unified build command:** `poetry build` creates both source and wheel distributions
- **Simplified publishing:** `poetry publish` handles uploading to PyPI
- **Version management:** Tools to bump version numbers according to semantic versioning

# N.3 Getting Started with Poetry

## N.3.1 Installation

Poetry can be installed in several ways:

```
Using the official installer (recommended)
curl -sSL https://install.python-poetry.org | python3 -

Using pipx
pipx install poetry

Using pip (not recommended for most cases)
pip install poetry
```

After installation, verify that Poetry is working:

```
poetry --version
```

## N.3.2 Creating a New Project

To create a new project with Poetry:

```
Create a new project
poetry new my-project

Project structure created:
my-project/
my_project/
__init__.py
tests/
__init__.py
pyproject.toml
README.md
```

Alternatively, initialize Poetry in an existing project:

```
Navigate to existing project
cd existing-project

Initialize Poetry
poetry init
```

This interactive command helps you create a `pyproject.toml` file with your project's metadata and dependencies.

### N.3.3 Basic Configuration

The `pyproject.toml` file is the heart of a Poetry project. Here's a sample:

```
[tool.poetry]
name = "my-project"
version = "0.1.0"
description = "A sample Python project"
authors = ["Your Name <your.email@example.com>"]
readme = "README.md"
packages = [{"include = "my_project"}]

[tool.poetry.dependencies]
python = "^3.8"
requests = "^2.28.0"
pandas = "^2.0.0"

[tool.poetry.group.dev.dependencies]
pytest = "^7.0.0"
black = "^23.0.0"
mypy = "^1.0.0"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

## N.4 Essential Poetry Commands

### N.4.1 Managing Dependencies

```
Install all dependencies
poetry install

Install only main dependencies (no dev dependencies)
poetry install --without dev

Add a new dependency
poetry add requests

Add a development dependency
```

```
poetry add pytest --group dev

Update all dependencies
poetry update

Update specific packages
poetry update requests pandas

Show installed packages
poetry show

Show dependency tree
poetry show --tree
```

## N.4.2 Environment Management

```
Create/use virtual environment
poetry env use python3.10

List available environments
poetry env list

Get information about the current environment
poetry env info

Remove an environment
poetry env remove python3.9
```

## N.4.3 Building and Publishing

```
Build source and wheel distributions
poetry build

Publish to PyPI
poetry publish

Build and publish in one step
poetry publish --build

Publish to a custom repository
poetry publish -r my-repository
```

## N.4.4 Running Scripts

```
Run a Python script in the Poetry environment
poetry run python script.py

Run a command defined in pyproject.toml
```

```
poetry run my-command

Activate the shell in the Poetry environment
poetry shell
```

## N.5 Advanced Poetry Features

### N.5.1 Dependency Groups

Poetry allows organising dependencies into logical groups:

```
[tool.poetry.dependencies]
python = "^3.8"
requests = "^2.28.0"

[tool.poetry.group.dev.dependencies]
pytest = "^7.0.0"
black = "^23.0.0"

[tool.poetry.group.docs.dependencies]
sphinx = "^5.0.0"
sphinx-rtd-theme = "^1.0.0"
```

Install specific groups:

```
Install only production and docs dependencies
poetry install --without dev

Install with specific groups
poetry install --only main,dev
```

### N.5.2 Version Constraints

Poetry supports various version constraint syntaxes:

- `^1.2.3`: Compatible with 1.2.3  $\leq$  version  $<$  2.0.0
- `~1.2.3`: Compatible with 1.2.3  $\leq$  version  $<$  1.3.0
- `>=1.2.3,<1.5.0`: Version between 1.2.3 (inclusive) and 1.5.0 (exclusive)
- `1.2.3`: Exactly version 1.2.3
- `*`: Any version

### N.5.3 Private Repositories

Configure private package repositories:

```
Add a repository
poetry config repositories.my-repo
https://my-repository.example.com/simple/

Add credentials
poetry config http-basic.my-repo username password
```

```
Install from the repository
poetry add package-name --source my-repo
```

## N.5.4 Script Commands

Define custom commands in your `pyproject.toml`:

```
[tool.poetry.scripts]
my-command = "my_package.cli:main"
start-server = "my_package.server:start"
```

These commands become available through `poetry run` or when the package is installed.

## N.6 Best Practices with Poetry

### N.6.1 Project Structure

A recommended project structure for Poetry projects:

```
my_project/
 src/
 my_package/ # Main package code
 __init__.py
 module.py
 tests/ # Test files
 __init__.py
 test_module.py
 docs/ # Documentation
 pyproject.toml # Poetry configuration
 poetry.lock # Lock file (auto-generated)
 README.md # Project documentation
```

To use the `src` layout with Poetry:

```
[tool.poetry]
...
packages = [{include = "my_package", from = "src"}]
```

### N.6.2 Dependency Management Strategies

1. **Minimal Version Specification:** Use `^` (caret) constraint to allow compatible updates

```
[tool.poetry.dependencies]
requests = "^2.28.0" # Allows any 2.x.y version >= 2.28.0
```

2. **Development vs. Production Dependencies:** Use groups to separate dependencies

```
[tool.poetry.dependencies]
Production dependencies
```

```
[tool.poetry.group.dev.dependencies]
Development-only dependencies
```

### 3. Update Strategy: Regularly update the lock file

```
Update dependencies and lock file
poetry update

Regenerate lock file based on pyproject.toml
poetry lock --no-update
```

## N.6.3 Version Control Practices

1. **Always commit the lock file:** The `poetry.lock` file ensures reproducible builds
2. **Consider a CI step to verify lock file consistency:**

```
In GitHub Actions
- name: Verify poetry.lock is up to date
 run: poetry lock --check
```

## N.6.4 Integration with Development Tools

### N.6.4.1 Code Formatting and Linting

Configure tools like Black and Ruff in `pyproject.toml`:

```
[tool.black]
line-length = 88
target-version = ["py39"]

[tool.ruff]
select = ["E", "F", "I"]
line-length = 88
```

### N.6.4.2 Type Checking

Configure mypy in `pyproject.toml`:

```
[tool.mypy]
python_version = "3.9"
warn_return_any = true
disallow_untyped_defs = true
```

## N.7 Integration with Development Workflows

### N.7.1 IDE Integration

Poetry integrates well with most Python IDEs:

#### N.7.1.1 VS Code

1. Install the Python extension

2. Configure VS Code to use Poetry's environment:
  - It should detect the Poetry environment automatically
  - Or set `python.poetryPath` in settings

### N.7.1.2 PyCharm

1. Go to Settings → Project → Python Interpreter
2. Add the Poetry-created interpreter (typically in `~/.cache/pypoetry/virtualenvs/`)
3. Or use PyCharm's Poetry plugin

## N.7.2 CI/CD Integration

### N.7.2.1 GitHub Actions Example

```

name: Python CI

on:
 push:
 branches: [main]
 pull_request:
 branches: [main]

jobs:
 test:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3

 - name: Set up Python
 uses: actions/setup-python@v4
 with:
 python-version: "3.10"

 - name: Install Poetry
 uses: snok/install-poetry@v1
 with:
 version: "1.5.1"

 - name: Install dependencies
 run: poetry install

 - name: Run tests
 run: poetry run pytest

```

## N.8 Troubleshooting Common Issues

### N.8.1 Dependency Resolution Errors

If Poetry can't resolve dependencies:

```
Show more detailed error information
poetry install -v

Try updating Poetry itself
poetry self update

Try with specific versions to identify the conflict
poetry add package-name==specific.version
```

## N.8.2 Virtual Environment Problems

For environment-related issues:

```
Get environment information
poetry env info

Create a fresh environment
poetry env remove --all
poetry install

Use a specific Python version
poetry env use /path/to/python
```

## N.8.3 Package Publishing Issues

When facing publishing problems:

```
Verify your PyPI credentials
poetry config pypi-token.pypi your-token

Check build before publishing
poetry build
Examine the resulting files in dist/

Publish with more information
poetry publish -v
```

# N.9 Comparison with Other Tools

## N.9.1 Poetry vs. pip + venv

- **Poetry:** Single tool for environment, dependencies, and packaging
- **pip + venv:** Separate tools for different aspects of the workflow
- **Key difference:** Poetry adds dependency resolution and lock file

## N.9.2 Poetry vs. Pipenv

- **Poetry:** Stronger focus on packaging and publishing
- **Pipenv:** Primarily focused on application development
- **Key difference:** Poetry's packaging capabilities make it more suitable for libraries

### N.9.3 Poetry vs. PDM

- **Poetry:** More opinionated, integrated experience
- **PDM:** More standards-compliant, supports PEP 582
- **Key difference:** Poetry's custom installer vs. PDM's closer adherence to PEP standards

### N.9.4 Poetry vs. Hatch

- **Poetry:** Focus on dependency management and packaging
- **Hatch:** Focus on project management and multi-environment workflows
- **Key difference:** Poetry's stronger dependency resolution vs. Hatch's project lifecycle features

## N.10 When to Use Poetry

Poetry is particularly well-suited for:

1. **Library Development:** Its packaging and publishing tools shine for creating distributable packages
2. **Team Projects:** The lock file ensures consistent environments across team members
3. **Projects with Complex Dependencies:** The resolver helps manage intricate dependency requirements
4. **Developers Wanting an All-in-One Solution:** The unified interface simplifies the development workflow

Poetry might not be ideal for:

1. **Simple Scripts:** May be overkill for very small projects
2. **Projects with Unusual Build Requirements:** Complex custom build processes might need more specialized tools
3. **Integration with Existing pip-Based Workflows:** Requires adapting established processes

## N.11 Conclusion

Poetry represents a significant evolution in Python package management, offering a more integrated and user-friendly approach to dependencies, environments, and packaging. Its focus on deterministic builds through the lock file mechanism and simplified workflow commands addresses many pain points in traditional Python development.

While Poetry introduces its own conventions and may require adaptation for teams used to traditional tools, the benefits in terms of reproducibility and developer experience make it worth considering for both new and existing Python projects. As the tool continues to mature and the ecosystem around it grows, Poetry is establishing itself as a standard part of the modern Python development toolkit.