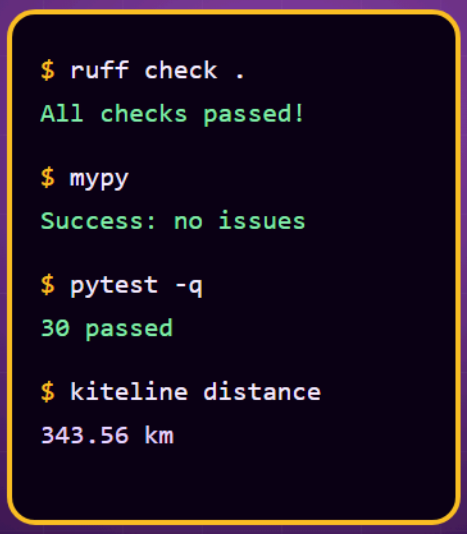
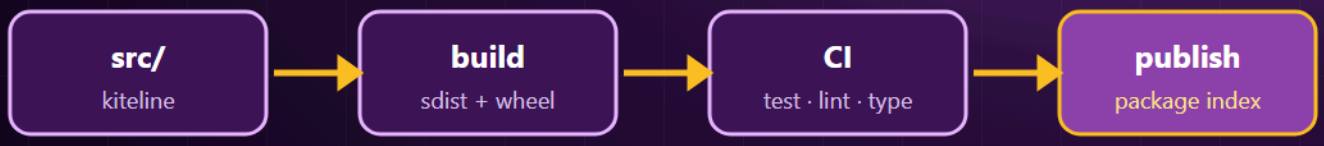
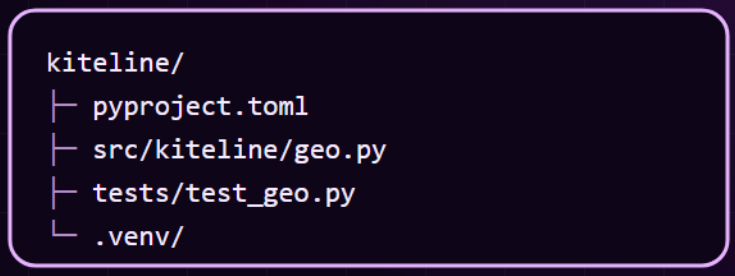
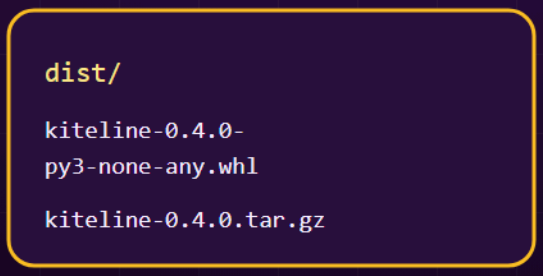










FROM SCRIPT TO RELEASE

Modern Python Project Tooling

Virtual environments, pyproject.toml, packaging, publishing, linting, and CI for professional projects

venv · pip · lock files · uv · Poetry · PDM · Hatch · wheels
pytest · Ruff · mypy · GitHub Actions · PyPI · Docker

ENVIRONMENTS · PACKAGING · QUALITY · CI · PUBLISHING

THENEXBOOKS

Modern Python Project Tooling

Virtual Environments, pyproject.toml, Packaging, Publishing, Linting, and CI for Professional Python Projects

From a single script to a tested, typed, packaged, and published 1.0 release — built step by step

First Edition

THENEXBOOKS

Modern Python Project Tooling
Virtual Environments, pyproject.toml, Packaging, Publishing, Linting, and CI for Professional Python Projects

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

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 publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples and configuration files may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Python packaging tools change quickly; commands, options, defaults, and output can differ between versions of Python, pip, and the other tools described. Always consult the official documentation for the versions you use. Nothing in this book is legal advice; the licensing discussion is a practical overview only.

Python and the Python logos are trademarks or registered trademarks of the Python Software Foundation. PyPI and the Python Package Index are associated with the Python Software Foundation. GitHub and GitHub Actions are trademarks of GitHub, Inc. Docker is a trademark of Docker, Inc. Windows, Visual Studio Code, and PowerShell are trademarks of the Microsoft group of companies. macOS is a trademark of Apple Inc. Linux is a registered trademark of Linus Torvalds. uv, Ruff, Poetry, PDM, Hatch, pytest, mypy, and other tool names belong to their respective owners and projects. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the Python Software Foundation or any of the projects or companies mentioned.

kiteline, the open-source geospatial library used as the running example in this book, is fictional, as are Kiteline OSS, its maintainers and contributors, its release history, and its repositories and web addresses. Email addresses and domains use reserved example names. Map coordinates are used only as sample data.

Preface

Most Python projects begin as a script. It solves one problem, it lives in one folder, and it runs on one machine. Then other people start to depend on it. A colleague asks how to install it. A new laptop has a different Python version. A dependency releases an update and something breaks. Someone wants to use the code as a library, someone else wants a command they can run from anywhere, and soon you are wondering whether you need a `setup.py`, a `requirements.txt`, a `pyproject.toml`, or all three — and which of the many tools everyone recommends you should actually use.

Python's tooling has improved enormously, but it has also become crowded. Standards such as `pyproject.toml`, wheels, and dependency groups now give every tool a common foundation, while project managers such as uv, Poetry, PDM, and Hatch compete to make daily work smoother. The result is powerful, and confusing if nobody explains how the pieces fit together.

This book is that explanation. We start with the interpreter and the virtual environment, work up through dependencies, lock files, project layout, building, and publishing, and finish with the habits of a professionally maintained project: tests, linting, type checking, continuous integration, security, and releases. We explain *why* each piece exists before we show *how* to use it, so that when tools change — and they will — you still understand what they are doing for you.

Who This Book Is For

- **Developers moving from scripts to projects** who can write Python but have never packaged, published, or maintained a library or tool.
- **Team developers** who inherited a Python codebase with a confusing setup and want reproducible environments, consistent checks, and a clean release process.
- **Data, ops, and automation people** whose scripts became tools that other people rely on.
- **New open-source maintainers** who want to ship a library to PyPI with confidence and keep it healthy over time.

You should already be comfortable with Python itself: functions, modules, classes, and running code from a terminal. We do not teach the language.

What You Will Learn

- How Python interpreters, versions, and the Windows `py` launcher work, and how to manage several Pythons side by side.
- How virtual environments isolate projects, and how pip, requirements files, constraints, hashes, and lock files make installs reproducible.
- What uv, Poetry, PDM, and Hatch each offer compared with plain pip, and how to choose between them.

- How to write a `pyproject.toml` by hand, structure a project with the src layout, and declare dependencies, extras, and dependency groups correctly.
- How packaging really works: source distributions, wheels, build backends such as setuptools and hatchling, entry points, versioning, metadata, and licensing.
- How to keep code healthy with pytest, Ruff, mypy, and pre-commit, and how to run the same checks locally and in GitHub Actions.
- How to document a project, publish to TestPyPI and PyPI with Trusted Publishing, and automate releases.
- How to protect your project's supply chain, organize monorepos, build good Docker images for Python applications, and maintain an open-source project over the long term.

How This Book Is Organized

Part I — From Scripts to Projects (Chapters 1–5) covers interpreters, virtual environments, pip, requirements files, and reproducible installs with hashes and lock files. **Part II — Project Structure and pyproject.toml** (Chapters 6–10) compares the modern project managers and builds a real project: `pyproject.toml`, the src layout, dependencies and extras, and the everyday development loop. **Part III — Building and Packaging** (Chapters 11–15) explains sdists and wheels, build backends, entry points and console scripts, versioning, and metadata. **Part IV — Code Quality** (Chapters 16–19) covers pytest, Ruff, mypy, and automating checks. **Part V — Documentation, CI, and Publishing** (Chapters 20–23) covers documentation, GitHub Actions, publishing to PyPI, and release engineering. **Part VI — Production and Maintenance** (Chapters 24–28) covers supply-chain security, monorepos, Docker, open-source maintenance, and a complete case study. The appendices provide a cheat sheet, templates and reference material, a release checklist, and a glossary.

Each chapter includes exercises under **Try It Yourself** and ends with a **Key Takeaways** box for quick review.

The Running Example: kiteline

Throughout the book we follow **kiteline**, a small open-source geospatial library and command-line tool maintained by a fictional volunteer group called **Kiteline OSS**. It calculates great-circle distances and bearings between points, converts between kilometers, miles, and nautical miles, and summarizes GPS tracks stored in GPX files. It begins life as a forty-line script that its lead maintainer wrote to plan kite-surfing trips, and by the end of the book it is a typed, tested, documented package with a `kiteline` command, continuous integration on three operating systems, an automated release workflow, and a 1.0 release.

kiteline is deliberately small, so the tooling — not the geometry — stays in the spotlight, and every number in the book can be checked: the distance from London to Paris is 343.56 km whether you compute it in a test, from the command line, or by hand.

Versions and Conventions

Examples target **Python 3.12 and newer**. Commands and outputs in this book were run on Windows with CPython 3.12 and the tool versions noted in each chapter; where a command needs the network or a tool that changes quickly — uploading to PyPI, installing new packages, or running uv, Poetry, PDM, Hatch, pre-commit, or Docker — we show it with output labeled "you should see something like", because the exact text differs between versions. Tools in this space release often; when a flag or default seems different on your machine, check the tool's documentation for your version.

We show Windows PowerShell commands first, with a `PS C:\work\kiteline>` prompt, and macOS or Linux commands with a `$` prompt where they differ. We prefer `python -m pip`, `python -m pytest`, and similar `-m` forms because they always run the tool that belongs to the interpreter you mean.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can break installs, leak secrets, or publish the wrong thing.

File names appear above code listings, such as `pyproject.toml` or `src/kiteline/geo.py`. Command output appears in a separate block labeled **Output**. Long lines are wrapped to fit the page.

Contents

Preface

PART I — FROM SCRIPTS TO PROJECTS

1 From Script to Project: What Modern Python Tooling Solves

2 Python Interpreters and Versions: The py Launcher, PATH, and Version Managers

3 Virtual Environments in Depth: venv, Activation, and Isolation

4 pip and Requirements Files: Installing, Pinning, and Constraints

5 Reproducible Environments: Lock Files, Hashes, and Dependency Resolution

PART II — PROJECT STRUCTURE AND PYPROJECT.TOML

6 Project Managers Compared: uv, Poetry, PDM, Hatch, and Plain pip

7 pyproject.toml Anatomy: build-system, project, and tool Tables

8 Project Layout: src Layout, Packages, Modules, and Data Files

9 Dependencies Done Right: Version Specifiers, Extras, and Dependency Groups

10 Editable Installs and the Everyday Development Loop

PART III — BUILDING AND PACKAGING

11 How Packaging Works: Sdists, Wheels, Build Backends, and Frontends

12 Building with setuptools and hatchling

13 Entry Points and Console Scripts: Shipping the kiteline CLI

14 Versioning: Semantic Versioning, PEP 440, Single-Sourcing, and Changelogs

15 Metadata, Licensing, READMEs, and Typed Packages

PART IV — CODE QUALITY

16 Testing with pytest: Layout, Fixtures, Parametrize, and Coverage

17 Linting and Formatting with Ruff

18 Static Typing with mypy

19 Automating Checks: pre-commit, nox, and Task Runners

PART V — DOCUMENTATION, CI, AND PUBLISHING

20 Documentation: Docstrings, READMEs, and Documentation Sites

21 Continuous Integration with GitHub Actions for Python Projects

22 Publishing to TestPyPI and PyPI: Tokens and Trusted Publishing

23 Release Engineering: Tags, Changelogs, and Automated Releases

PART VI — PRODUCTION AND MAINTENANCE

24 Security and the Software Supply Chain

25 Monorepos and Multi-Package Projects

26 Docker for Python Applications

27 Maintaining an Open-Source Project: Contributors, Deprecations, and Upgrades

28 Case Study: kiteline from Script to 1.0

APPENDICES

A Tooling Cheat Sheet

B Templates and Reference

C Release and Production Checklist

D Glossary

PART I

From Scripts to Projects

Why a script stops being enough, how Python interpreters and versions are installed and selected, how virtual environments isolate each project, and how pip, requirements files, hashes, and lock files make installs repeatable on every machine.

1 · From Script to Project
2 · Python Interpreters and Versions
3 · Virtual Environments in Depth
4 · pip and Requirements Files
5 · Reproducible Environments

CHAPTER 1 · PART I — FROM SCRIPTS TO PROJECTS

From Script to Project: What Modern Python Tooling Solves

Every Python project starts as a file that works on one machine. This chapter shows exactly what goes wrong when that file has to work for other people, maps the tools that fix each problem, and introduces kiteline, the small library we will grow from a 39-line script into a tested, typed, published 1.0 release.

The Script That Became a Tool

Mara Lindqvist kite-surfs. Before every trip she wanted a quick answer to a simple question: how far is the next spot from here, as the crow flies? She knew the haversine formula from a navigation course, so one evening she wrote a short Python script, saved it as `kite.py`, and ran it from a terminal. It printed a distance in kilometers. Problem solved.

Then a friend asked for a copy. Then the friend's club started using it to plan group trips and asked for miles. Somebody wanted the compass direction as well. Somebody else exported a GPS track from a watch and asked whether the script could total the distance of a whole session. Within a few months the script was "the tool everyone uses", and Mara was answering messages that started with "it doesn't work on my laptop".

That story is the reason this book exists. Mara and a few volunteers eventually formed **Kiteline OSS**, a small open-source group, and turned the script into **kiteline**: a dependency-free geospatial library plus a `kiteline` command-line tool. Along the way they had to answer every question that modern Python tooling was invented to answer. We will follow the same path, one tool at a time, and by the end you will have done everything they did on your own machine.

Here is the script as Mara first shared it. It is ordinary, reasonable Python. Nothing about it is wrong for a single person on a single computer.

kite.py

```
"""kite.py - how far is the next kite spot? (Mara's original script)"""

import math
import sys

EARTH_RADIUS_KM = 6371.0088

# Favorite spots, typed in by hand: name -> (latitude, longitude)
SPOTS = {
    "london": (51.5074, -0.1278),
    "paris": (48.8566, 2.3522),
```

```
    "newyork": (40.7128, -74.0060),
    "tokyo": (35.6762, 139.6503),
}


def haversine(lat1, lon1, lat2, lon2):
    """Great-circle distance between two points, in kilometers."""
    lat1, lon1, lat2, lon2 = map(math.radians, (lat1, lon1, lat2, lon2))
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    h = (
        math.sin(dlat / 2) ** 2
        + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2) ** 2
    )
    return 2 * EARTH_RADIUS_KM * math.asin(math.sqrt(h))


if __name__ == "__main__":
    args = sys.argv[1:]
    if len(args) == 2:
        # kite.py london paris
        args = [*SPOTS[args[0]], *SPOTS[args[1]]]
    if len(args) != 4:
        print("usage: kite.py LAT1 LON1 LAT2 LON2  (or: kite.py SPOT1 SPOT2)")
        sys.exit(2)
    lat1, lon1, lat2, lon2 = (float(x) for x in args)
    km = haversine(lat1, lon1, lat2, lon2)
    print(f"{km:.2f} km")
```

The file has one function, `haversine(lat1, lon1, lat2, lon2)`, a dictionary of favorite spots, and a `__main__` block that reads `sys.argv`. The constant `6371.0088` is the mean radius of the Earth in kilometers, the same value the finished library uses. Let us run it with Python 3.12 using the Windows `py` launcher (Chapter 2 explains the launcher in detail).

Terminal (PowerShell)

```
PS C:\work\kiteline> py -3.12 kite.py 51.5074 -0.1278 48.8566 2.3522
PS C:\work\kiteline> py -3.12 kite.py london paris
PS C:\work\kiteline> py -3.12 kite.py newyork tokyo
```

Output

```
343.56 km
343.56 km
10851.75 km
```

London to Paris is 343.56 km along a great circle, and New York to Tokyo is 10,851.75 km. Those numbers will appear again and again in this book, because the finished library produces exactly the same results. The formula never changes; everything around it does.

"Works on My Machine": The Four Questions

The script works for Mara. The trouble starts the moment anyone else is involved, including Mara herself six months later on a new laptop. Almost every complaint the kiteline volunteers received falls into one of four questions. Keep these in mind, because each part of this book answers one or more of them.

Question 1: Which Python?

`kite.py` uses the `*` unpacking syntax inside a list display and an f-string, both of which are fine on any current Python. The later library uses `itertools.pairwise` (added in Python 3.10), the `X | None` union syntax in annotations, and dataclasses with `slots=True`. A friend who runs it with an old interpreter that happens to be first on their PATH gets a confusing error that has nothing to do with geography. Worse, many computers have several Pythons installed at once: one from python.org, one from a package manager, one bundled with some other application. Typing `python` picks whichever one the shell finds first.

A project needs to *declare* which Python versions it supports and the developer needs a reliable way to *choose* an interpreter. Chapter 2 covers the `py` launcher, PATH, version managers, and the `requires-python` field.

Question 2: Which Packages?

Mara's script uses only the standard library, which is lucky. The moment the group added a pretty table output with the third-party `rich` library, the instructions grew a line: "first run pip install rich". Which version? Installed where? Into the same Python that runs the script, or a different one? Installed globally, where it might break another tool that needs a different version of the same library?

The answers are **virtual environments** (an isolated set of installed packages per project, Chapter 3), an **installer** that understands version requirements (pip, Chapter 4), and, for applications that need the exact same versions everywhere, **lock files** (Chapter 5).

Question 3: How Do Other People Install It?

"Download `kite.py` from the chat and put it somewhere" does not scale. People expect to type `pip install kiteline` and then run `kiteline` as a command, on Windows, macOS, and Linux alike. Making that work requires a project description that tools can read (`pyproject.toml`, Chapter 7), a sensible folder layout (Chapter 8), declared dependencies (Chapter 9), a **build** step that turns the source tree into installable files called **wheels** and **sdists** (Chapters 11 and 12), a console-script entry point (Chapter 13), a version number with meaning (Chapter 14), correct metadata and a license (Chapter 15), and finally an upload to the Python Package Index (Chapter 22).

Question 4: How Do We Know It Still Works?

Look at what the script does with bad input. Here are two runs that real users could easily type:

Terminal (PowerShell)

```
PS C:\work\kiteline> py -3.12 kite.py 95 0 0 0
PS C:\work\kiteline> py -3.12 kite.py london rome
```

Output

```
10563.53 km
Traceback (most recent call last):
  File "C:\work\kiteline\kite.py", line 33, in <module>
    args = [*SPOTS[args[0]], *SPOTS[args[1]]]
                              ~~~~~^^^^^^^^^
KeyError: 'rome'
```

A latitude of 95 degrees does not exist, yet the script cheerfully reports a distance of 10,563.53 km. An unknown spot name produces a raw traceback that means nothing to a non-programmer. Neither bug is hard to fix. The hard part is knowing, after every change, that the fix did not break something else, on every supported Python, on every operating system. That is the job of automated **tests** (Chapter 16), **linters and formatters** (Chapter 17), a **type checker** (Chapter 18), a way to run all of them with one command (Chapter 19), and **continuous integration** that runs them on every push (Chapter 21).

Compare the finished library's behavior for the same impossible latitude. It refuses clearly and exits with status 1:

Terminal (PowerShell)

```
(.venv) PS C:\work\kiteline> kiteline distance 95 0 0 0
```

Output

```
kiteline: error: latitude out of range: 95.0
```

That one line is the product of a `Point` class that validates its inputs, a CLI that turns exceptions into friendly messages, tests that pin down both behaviors, and a CI pipeline that keeps them true. None of it is clever. All of it is tooling and habits, which is what this book teaches.

> **NOTE**
>
> We do not teach the Python language in this book. We assume you can write functions, modules, and classes and that you have at least seen a pytest test. If a code snippet uses a language feature you have not met, it is never the point of the example; the point is always how the project around the code is organized, checked, built, and shipped.

The Toolchain Map

The Python ecosystem has a reputation for having "too many tools". Part of that reputation is deserved, but most of the confusion disappears once you see that each tool does one job in a pipeline, and that several competing tools exist for some of the jobs. Here is the whole pipeline on one page. Read it from top to bottom: each layer depends on the ones above it.

The modern Python toolchain

```
interpreter      CPython 3.12, 3.13, 3.14 ...   py launcher, pyenv, uv python
     |
environment      an isolated site-packages      venv, virtualenv, uv venv
     |
installer        puts distributions into it     pip, uv pip
     |
resolver / lock  picks compatible versions,     pip's resolver, pip-tools,
                 records them exactly           uv.lock, poetry.lock, pylock.toml
     |
build backend    turns source into wheel/sdist  setuptools, hatchling, flit-core,
                                                poetry-core, pdm-backend, uv_build
     |
build frontend   calls the backend for you      python -m build, pip wheel, uv build
     |
index            where distributions live       PyPI, TestPyPI, private indexes
     |
quality tools    keep the code healthy          pytest, Ruff, mypy, pre-commit, nox
     |
CI               runs all of it on every push   GitHub Actions (and others)
```

A few observations make this map much less intimidating.

- **Each layer has a standard interface.** A build frontend talks to any backend through the same two or three functions defined in PEP 517. An installer reads the same wheel format regardless of which backend produced it. That is why you can switch from setuptools to hatchling without your users noticing (Chapter 12).
- **All-in-one "project managers" bundle several layers.** uv, Poetry, PDM, and Hatch each cover interpreter management, environments, installation, locking, building, and publishing to different degrees. They do not replace the standards; they implement them. Chapter 6 compares them side by side.
- **You do not need every layer on day one.** A personal script needs an interpreter and maybe a virtual environment. A shared internal tool adds a lock file. A published library adds builds, metadata, and an index. We add layers in exactly that order.

Throughout the book we use the plain, standard tools for the canonical path: `venv`, `pip`, setuptools as the backend, and the PyPA `build` and `twine` tools for publishing. Every project manager builds on the same concepts, so once you understand the plain path you can read any of them. Where uv, Poetry, PDM, or Hatch do something notably differently we show their commands next to the plain ones.

> **TIP**
>
> When a new tool appears and someone says it "replaces pip" or "replaces setuptools", place it on the map. Ask which layers it covers and which standard files it reads and writes. If it reads `pyproject.toml` and produces normal wheels, your project is not locked in, and switching later is mostly a matter of changing a few lines.

Standards versus Tools

The reason Python packaging works at all, across dozens of tools written by different groups, is a set of written standards. Most of them started life as **Python Enhancement Proposals** (PEPs). Once accepted, the packaging-related ones are maintained as living specifications on **packaging.python.org**, the website of the **Python Packaging Authority** (PyPA). The PyPA is a working group that maintains core projects such as pip, setuptools, `build`, `twine`, virtualenv, and the packaging documentation itself. Tools like uv, Poetry, and PDM are developed by other groups but implement the same specifications.

You do not need to read the PEPs to use this book, but knowing their numbers helps enormously when you search for answers or read a tool's changelog. These are the ones we refer to most often:

Standard	What it defines	Where we use it
PEP 517 / 518	Build backend interface; `[build-system]` requirements	Chapters 7, 11, 12
PEP 621	The `[project]` table in `pyproject.toml`	Chapter 7
PEP 440	Version numbers: `1.0.0`, `1.0.0rc1`, `1.0.0.post1`	Chapter 14
PEP 508	Dependency specifiers, extras, environment markers	Chapters 4, 9
PEP 427	The wheel file format and tags	Chapter 11
PEP 660	Editable installs through the backend	Chapter 10
PEP 561	`py.typed` for typed packages	Chapters 15, 18
PEP 639	SPDX `license` expressions and `license-files`	Chapter 15
PEP 668	"Externally managed" system Pythons	Chapters 2, 4
PEP 735	`[dependency-groups]` for dev tools	Chapter 9
PEP 751	The `pylock.toml` lock file format	Chapter 5

The distinction between a standard and a tool matters for one practical reason: **standards change slowly and tools change fast**. The `[project]` table you write today will be readable for many years. The exact flags of a specific tool may change in its next release. That is why we explain the standard

first, then the tool, and why we say "recent versions of pip" or "recent versions of uv" whenever a feature is new. When a feature is still experimental, such as pip's `pip lock` command at the time of writing, we label it clearly.

> **NOTE**
> A distribution's name on PyPI (what you `pip install`) and the package you `import` are two different things that happen to match for kiteline. Chapter 9 explains the difference and why names like `Kite_Line` and `kite-line` are treated as the same project on an index.

Where kiteline Ends Up

Before we start the journey, let us look at the destination. Here is the kiteline project at version 0.4.0, the version we use through most of the book. The listing comes from `tree /F /A` on Windows after removing temporary caches.

Terminal (PowerShell)

```
PS C:\work\kiteline> tree /F /A
```

Output

```
C:.
|   .gitignore
|   CHANGELOG.md
|   LICENSE
|   MANIFEST.in
|   pyproject.toml
|   README.md
|
+---.github
|   \---workflows
|           ci.yml
|
+---src
|   \---kiteline
|           cli.py
|           geo.py
|           gpx.py
|           py.typed
|           units.py
|           __init__.py
|           __main__.py
|
\---tests
    |   conftest.py
    |   test_cli.py
    |   test_geo.py
    |   test_gpx.py
```

```
    |
    \---data
            harbor-loop.gpx
```

Every file has a reason to exist, and every reason maps back to one of the four questions. Here is a quick tour, with the chapter where each file is explained in depth.

File or folder	Purpose	Chapter
`pyproject.toml`	Project metadata, build backend, and tool settings in one file	7, 9, 15
`src/kiteline/`	The importable package, kept under `src/` on purpose	8
`__main__.py`, `cli.py`	The `kiteline` command and `python -m kiteline`	13
`py.typed`	Marker telling type checkers the package ships type hints	15, 18
`tests/`	30 pytest tests, shared fixtures, and a sample GPX track	16
`MANIFEST.in`	Extra files to include in the source distribution	8, 12
`CHANGELOG.md`	Human-readable history in Keep a Changelog format	14, 23
`LICENSE`	The MIT license text	15
`.github/workflows/`	Continuous integration on three operating systems	21
`.gitignore`	Keeps environments, caches, and build output out of git	3, 11

Just as important is what is *not* in the tree. There is no `setup.py`, because modern setuptools reads everything it needs from `pyproject.toml`. There is no `.venv` folder in version control, because environments are created on each machine, never shared. There is no `requirements.txt` for the library itself, because a library declares ranges of compatible versions rather than exact pins (Chapter 5 explains the rule). And the library has **zero runtime dependencies**: everything it needs ships with Python. The `rich` library is an *optional extra*, used only by one command option.

The finished library answers the same question Mara's script did, but as a proper command with subcommands, unit options, and clear errors. With kiteline installed in a virtual environment (we set that up in Chapter 3), these are the commands and their real output:

Terminal (PowerShell)

```
(.venv) PS C:\work\kiteline> kiteline --version
(.venv) PS C:\work\kiteline> kiteline distance 51.5074 -0.1278 48.8566 2.3522
(.venv) PS C:\work\kiteline> kiteline bearing 51.5074 -0.1278 48.8566 2.3522
(.venv) PS C:\work\kiteline> kiteline stats tests/data/harbor-loop.gpx
```

Output

```
kiteline 0.4.0
343.56 km
148.1 deg (SE)
```

```
points:   6
distance: 3.92 km
climb:    21.5 m
duration: 0:35:00
```

The distance matches the script exactly. The bearing says that, leaving London for Paris, you start out heading 148.1 degrees, roughly south-east. The `stats` command reads a GPS track in the standard GPX format, here a short six-point "Harbor Loop" recorded on a kite session, and reports the number of points, the total distance, the elevation gained, and the elapsed time. Notice that the output uses the ASCII word "deg" rather than a degree symbol; that small choice keeps the tool working on Windows consoles that still use legacy code pages.

The test suite proves all of this automatically:

Terminal (PowerShell)

```
(.venv) PS C:\work\kiteline> python -m pytest -q
```

Output

```
..............................                                       [100%]
30 passed in 1.78s
```

Thirty tests, each a small promise about behavior: that London to Paris is 343.56 km, that latitude 95 is rejected, that the GPX loop has 21.5 m of climb, and so on. The time at the end varies from run to run and machine to machine; never treat it as a benchmark.

The People Behind kiteline

kiteline is fictional, but its maintainers behave like a real small open-source team, and we will meet them in the chapters where their roles matter.

- **Mara Lindqvist**, the lead maintainer, wrote `kite.py` and still writes most of the geometry code.
- **Tomás Reyes**, the release manager, owns version numbers, the CI configuration, and publishing to PyPI.
- **Aiko Tanaka** looks after documentation and type hints.
- **Sam Osei** is a first-time contributor whose first pull request we review in Chapter 27.

The group's repository lives at `kiteline-oss/kiteline` on GitHub, with `main` as the default branch. Contact addresses use the reserved `.example` domain, such as `maintainers@kiteline.example`, so nothing in the book points to a real mailbox.

The Release History in Brief

kiteline already has a short history by the time we join. Version 0.1.0 introduced the `Point` type, `haversine_km`, unit conversion, and the `kiteline distance` command. Version 0.2.0 added bearings and compass points. Version 0.3.0 added GPX parsing. Version 0.4.0, the one in our sample project, added track statistics, the optional pretty table, and raised the minimum Python to 3.12; it also deprecated an old helper called `km_to_miles()`. In Chapter 14 we bump to 0.5.0 to add a `midpoint()` function, and in Chapter 28 we ship 1.0.0. Every other chapter uses 0.4.0, so the outputs you see stay consistent from chapter to chapter.

Why Not Just Keep the Script?

It is fair to ask whether all of this is overkill. Plenty of useful Python lives as single files forever, and that is fine. A useful rule of thumb is to look at who depends on the code and what breaks if it changes.

Situation	What you need
Only you run it, occasionally	A script and a known Python version
It needs third-party packages	A virtual environment and a requirements file
Colleagues run it on their machines	Pinned versions or a lock file; tests
Others import it in their own code	A package with `pyproject.toml`, a version, a changelog
Strangers install it from PyPI	All of the above, plus CI, metadata, a license, and a release process

kiteline walks up every row of that table, which is why it makes a good running example. Your own project may stop at row two or three, and you can stop reading the corresponding chapters there too. But even if you never publish anything, the habits in Parts I and IV (isolated environments, reproducible installs, tests, linting, and type checking) pay off on every project that lives longer than a week.

There is also a middle ground worth knowing about. For a single-file script with one or two dependencies, a recent standard called **inline script metadata** (PEP 723) lets you declare the dependencies in a special comment block at the top of the file. Tools such as uv and pipx, among others, can read that block and run the script in a temporary environment with the right packages. We mention it again in Chapter 6. It is a great fit for personal utilities, but as soon as other people import your code, a real project wins.

A PEP 723 script header (for illustration)

```
# /// script
# requires-python = ">=3.12"
# dependencies = ["rich>=13.7"]
# ///
import rich
...
```

> **WARNING**
>
> Resist the temptation to fix "works on my machine" by emailing someone your entire Python folder or a zipped virtual environment. Virtual environments contain absolute paths to the interpreter that created them and are not relocatable, and a copied global Python drags along whatever happened to be installed in it. The fix is always to *describe* the environment (Python version plus dependency list) and let each machine *recreate* it. Chapter 3 shows how quickly that is done.

How to Follow Along

Every chapter is hands-on. You will create files, run commands, and compare your output with ours. Here is what you need and how we write commands.

What You Need

- **Python 3.12 or newer.** Our examples were verified on CPython 3.12.0 on Windows, and everything also applies to 3.13 and 3.14. Chapter 2 shows how to install and select a specific version.
- **A terminal.** On Windows we use PowerShell. On macOS and Linux any bash- or zsh-compatible shell works.
- **git**, for the chapters on versioning, releases, and CI.
- **A code editor.** Any editor works; VS Code and PyCharm both understand virtual environments and `pyproject.toml`.
- **Internet access** for installing packages from PyPI on your own machine. Several of our examples deliberately work offline, which is useful knowledge in its own right (Chapter 4).

Command Conventions

We show Windows first and macOS/Linux second whenever the commands differ. A PowerShell prompt looks like `PS C:\work\kiteline>`, and once a virtual environment is activated it looks like `(.venv) PS C:\work\kiteline>`. A bash or zsh prompt is shown as a plain `$`, and the project lives in `~/work/kiteline`. You do not type the prompt itself.

Terminal (PowerShell)

```
PS C:\work\kiteline> py -3.12 -m venv .venv
PS C:\work\kiteline> .venv\Scripts\Activate.ps1
(.venv) PS C:\work\kiteline> python -m pytest -q
```

Terminal (bash)

```
$ cd ~/work/kiteline
$ python3.12 -m venv .venv
$ source .venv/bin/activate
```