Publishing Python Packages

Dane Hillard

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Publishing Python Packages
Version 02

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP of *Publishing Python Packages*. Your participation and support will be a big part of making the book as effective as possible for those looking to create and publish great Python packages!

Python packaging has historically been criticized, but with advancements over the past few years the story around it has improved significantly. Many people still haven't caught up to what's available and how it works, so this book is a chance to bring people together on the same page.

Whether you're brand new to Python packaging or you already maintain a package or two, there's something in this book for you. Although it covers some of the underpinnings of packaging, it also covers practical aspects of sustainable maintainership. After all, packaging is a means to an end—sharing valuable code with others. Understanding the core concepts of packaging, practicing with them, and then automating them to get them out of the way until you need to change them will help you focus on the real value you're giving others.

Packaging is a huge topic, and with this book I hope to cover in depth the right mix of parts to give most people a boost in productivity. You can help me best by telling me about your use cases, scenarios, and challenges in creating packages. If something isn't covered in the book but seems like a common problem, it's good evidence to support including it!

By the end of this book, you'll learn to:

- Build a package using the latest standards for Python packaging
- Include code written as a non-Python extension for better performance
- Provide a command-line interface for ease of use
- Automate your package build, testing, and publishing using GitHub Actions
- Publish your package documentation to Read the Docs

By purchasing this MEAP, you're joining a partnership that I hope we'll both benefit from. I appreciate your grace as you find the inevitable typo, confusing passage, or missing topic. In return I hope to provide you iteratively with stronger examples,clearer and more accessible language, and of course a great path to maintainable packaging. I look forward to seeing you in the liveBook Discussion forums.

— Dane Hillard

brief contents

PART 1: FOUNDATIONS

1 The what and why of Python package

2 Preparing for package development

3 The anatomy of a minimal Python package

PART 2: CREATING A VIABLE PACKAGE

4 Handling package dependencies, entry points, and extensions

5 Building and maintaining a test suite

6 Automating code quality tooling

PART 3: GOING PUBLIC

7 Automating work through continuous integration

8 Authoring and maintaining documentation

9 Making a package evergreen

PART 4: THE LONG HAUL

10 Creating a repeatable process

11 Building an open source community

APPENDIXES

A Installing asdf and python-launcher

B Installing pipx, build, and tox

1 The what and why of Python packages

This chapter covers

- Packaging code to make it more accessible for others
- Using packages to make your own projects more manageable
- Building Python packages for different platforms

Imagine that you've written a groundbreaking piece of Python software for use in self-driving cars. Your latest work is going to change the world, and you want as many people using it as possible. You've convinced CarCorp to use your solution, and they want to retrieve the code to get started with it.

When CarCorp calls asking how to install and use your code, you go through all the gory details of copying each file to the right directory, making some files executable so they can be run as commands, and so on. Because you wrote the software, this is all second nature for you. To your surprise, the developers on the other end of the phone are a bit lost. What happened?

You've discovered the chasm that often exists between those who create software and those who use it. These days, people are used to visiting the app store on their iPhone when they need something new. You have a bit of work to do if you want to improve the user experience of your software!

In this book, you'll learn how distributing your Python project as an installable package can make it more accessible to others. You'll also learn to create a repeatable process for managing your projects, reducing the effort you'll spend maintaining them so you can focus on your real aspiration to change the world. You'll do all this by building a real project using some popular

packaging tools and automating several aspects of the process. Although the Python community has developed standards for some areas of packaging, the One True Way© of doing things has not yet emerged. Nor may it ever do so!

Even if you've created or published a Python package before, there's something in this book for you. The suggestions and tools you'll learn in this book are time-tested approaches to some of the more loosely-defined packaging practices. Importantly, you can use them to ensure you don't run into some of the following common pitfalls:

- You add a new directory or file to your package's source code, but forget to include it in the package.
- You remember to add new files to the package, but forget to specify that they should be installed with the rest of the code.
- You succeed in including and installing all the code, but it breaks when you try to use it in an application.
- Your tests run successfully when developing the code, but fail after packaging and installing the code.

Python packaging has a messy history, so in addition to using the tools available now you'll also learn the methodology behind how they work to continue adapting as the landscape matures. To that end, it's important first to understand why software is packaged at all.

1.1 What is a package, anyway?

To save your relationship with CarCorp, you promise to come back in a few weeks with an overhauled process that will help them install your software in a snap. You know that some of your favorite Python code, like `pandas` and `requests`, are available as packages online and you want to provide the same ease of installation to your own consumers.

Packaging is the act of archiving software along with metadata that describes those files. Developers usually create these archives, or *packages*, with the intent of sharing or publishing them.

CAUTION The Python ecosystem overloads the word "package." The Python Packaging Authority (PyPA) differentiates the terms in the Python Packaging User Guide (packaging.python.org):

- Import packages organize multiple Python modules into a directory for discovery purposes.
- Distribution packages archive Python projects to be published for others to install.

Import packages aren't always distributed in an archive, though distribution packages often contain one or more import packages. Distribution packages are the main subject of this book, and will be disambiguated from import packages where necessary to avoid confusion.

1.1.1 The contents of a distribution package

Figure 1.1 shows some of the files you might choose to put in a distribution package. Developers often include the source code files in a package, but they can also provide compiled artifacts, test data, and whatever else a consumer or colleague might need. By distributing a package, your consumers will have a one-stop shop to grab all the pieces they need to get started with your software.

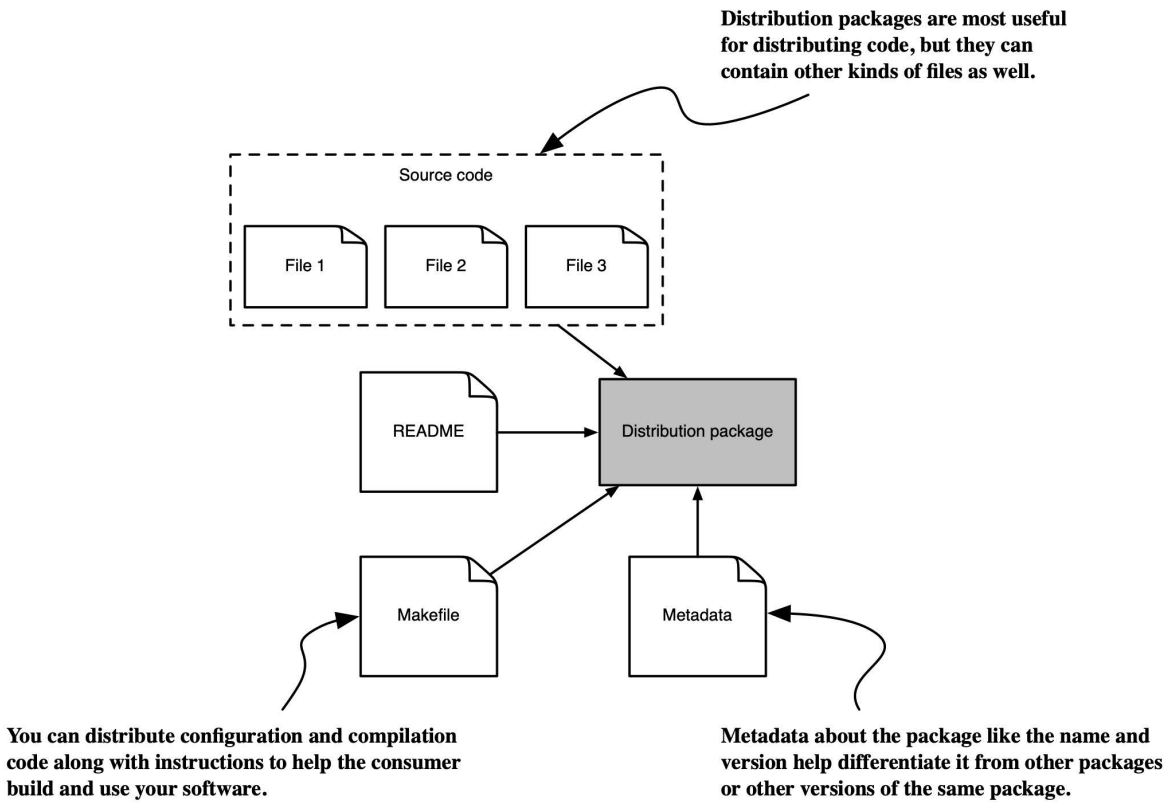

Figure 1.1 A package often includes source code, a Makefile for compiling the code, metadata about the code, and instructions for the consumer.

Distributing non-code files is an important capability. Although the code is often the reason to distribute anything in the first place, many users and tools depend on the metadata about the code to differentiate it from other code. Developers usually specify the name of a software project, its creator(s), the license under which it can be reused, and so on in the metadata. Importantly, the metadata often includes the *version* of the archive to distinguish it from previous and future publications of the project.

SIDEBAR **The early days of sharing software**

For more than a decade after the Unix operating system first became available, sharing software between teams and individuals remained a largely manual process. Downloading source code, compiling it, and contending with the artifacts of the compilation were all left up to the person trying to use the code. Each step in this process introduced opportunities for failure due to human error and architectural or environmental differences between systems. Tools like Make (www.gnu.org/software/make/) removed some of this variation from the process, but stop shy of dependency or version management.

Now that you're familiar with what goes into a package, you'll learn how this approach to sharing software solves specific problems in practice.

1.1.2 The challenges of sharing software

Your call with CarCorp is growing tense, and you realize you forgot to have them install all your project's dependencies first. You back up a few steps and navigate them through the dependency installation. Unfortunately, you forgot to check which version you've been using for one of your major dependencies and the latest version doesn't seem to work. You walk them through installing each previous version until you finally find one that works. Close one.

As you develop increasingly complex systems, the effort to make sure you've installed the required version of each dependency correctly grows quickly. In the worst cases you might reach a point where you need two different versions of the same dependency, and they can't coexist. This is affectionally known as "dependency hell." Detangling a project from this point can prove challenging.

Even without running into dependency hell, without a standardized approach to packaging it can be difficult to share software in a standard way that anyone, anywhere knows what other dependencies they need to install for your project. The next section covers how software communities create conventions and standards for managing packages.

1.1.3 Standardizing packaging for automation

Package management systems, or *package managers*, standardize the archive and metadata format for software packages in a particular domain. Package managers provide tools to help consumers install dependencies at the project, programming language, framework, or operating system levels. Most package managers ship with a familiar set of instructions to install, uninstall, or update packages. You may have used some of the following package managers:

- pip (pip.pypa.io/en/stable/)
- conda (docs.conda.io/en/latest/)
- Homebrew (brew.sh/)
- NPM (www.npmjs.com/)

SIDEBAR **The early days of package management**

Although developers had been packaging their code informally for some time, it wasn't until package management systems became widely available in the early 1990s that this approach took off (see Jeremy Katz, "A brief history of package management," Tidelift, blog.tidelift.com/a-brief-history-of-package-management).

The ability to declaratively define project dependencies proved a boon to developer productivity by abstracting away a major area of leg work in managing software projects.

Software repositories standardize packaging further by acting as centralized marketplaces to

publish and host packages others can install (figure 1.2). Many programming language communities provide an official or de facto standard repository for installing packages. PyPI (pypi.org), RubyGems (rubygems.org/), and Docker Hub (hub.docker.com/) are a few popular software repositories.

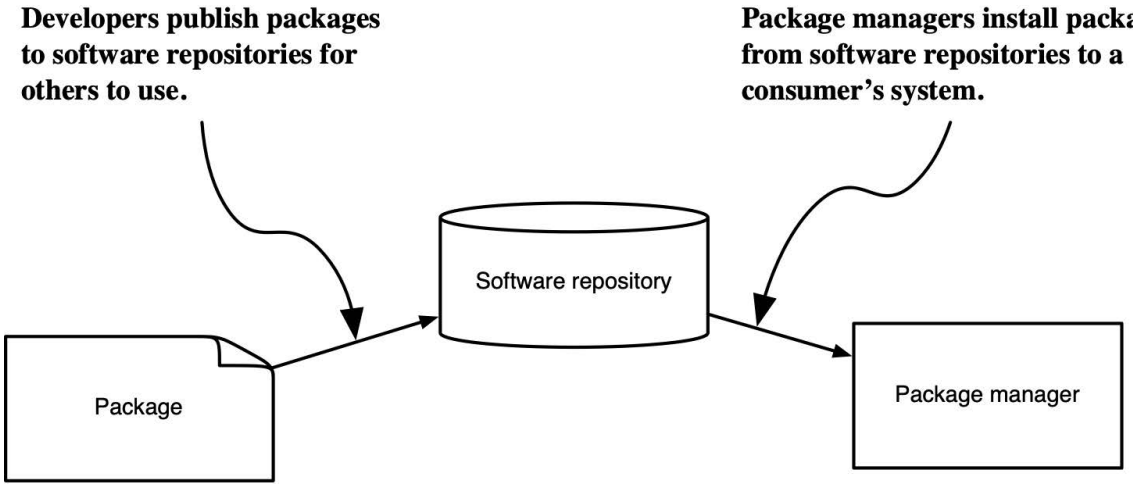

Figure 1.2 Packages, package managers, and software repositories are all critical in sharing software.

If you own a smartphone, tablet, or desktop computer and you've installed apps from an app store, that's packaging at work. Packages are software bundled together with metadata about that software, and that's precisely what an app is. Software repositories host software that people can install, and that's what the app store is.

Now that you understand why packaging is good for sharing software, read on to learn some of the advantages packaging can provide even if you aren't always making your software publicly available.

1.2 How packaging helps you

If you're new to packaging, so far it may seem like it's mainly useful for sharing software with people across the globe. Although that's certainly a good reason to package your code, you may also like some of the other benefits it brings:

- Stronger cohesion and encapsulation
- Clearer definition of ownership
- Looser coupling between areas of the code
- More opportunity for composition

The following sections cover these benefits in detail.

1.2.1 Enforcing cohesion and encapsulation through packaging

A particular area of code should generally have one job. *Cohesion* measures how dutifully the code sticks to that job. The more stray functionality floating around, the less cohesive the code is.

You've probably used functions, classes, modules, and import packages to organize your Python code (see Dane Hillard, "The hierarchy of separation in Python," *Practices of the Python Pro*, *Manning Publications*, 2020, pp. 25–39). These constructs each place a kind of named boundary around areas of code that have a particular job. When done well, naming communicates to developers what belongs inside the boundary and, importantly, what doesn't.

Despite best efforts, names and people are rarely perfect. If you put all your Python code in a single application, chances are some code will eventually seep into areas it doesn't belong. Think about some of the larger projects you've developed. How many times did you create a `utils.py` or `helpers.py` module containing a grab bag of functionality? The boundaries you create with a function or a module are readily overcome. These "utility" areas of the code tend to attract new "utilities," with the cohesion trending down over time.

Although assessing naming and regularly refactoring the code base can keep cohesion higher, it's also a maintenance burden. Distribution packages increase the barrier to adding code where it may not belong in the first place. Because updating a package necessitates going through a cycle of packaging, publishing, and installing the update, it prompts developers to think more deeply about the changes they make. You will be less likely to add code to a package without explicit intent that's worth the investment of the update cycle.

Creating cohesion and packaging a cohesive area of code is a gateway into *encapsulation*. Encapsulation helps you build the right expectations with your consumers about how to interact with your code by defining if and how the code's behavior is exposed. Think of a project you built and shared with someone to use. Now think about how many times you changed your code, and how many times they had to change their code in turn. How frustrating was it for them? How about for you? Encapsulation can reduce this kind of churn by better defining the API contract that's less subject to change. Figure 1.3 shows how you might create multiple packages out of cohesive areas of code.

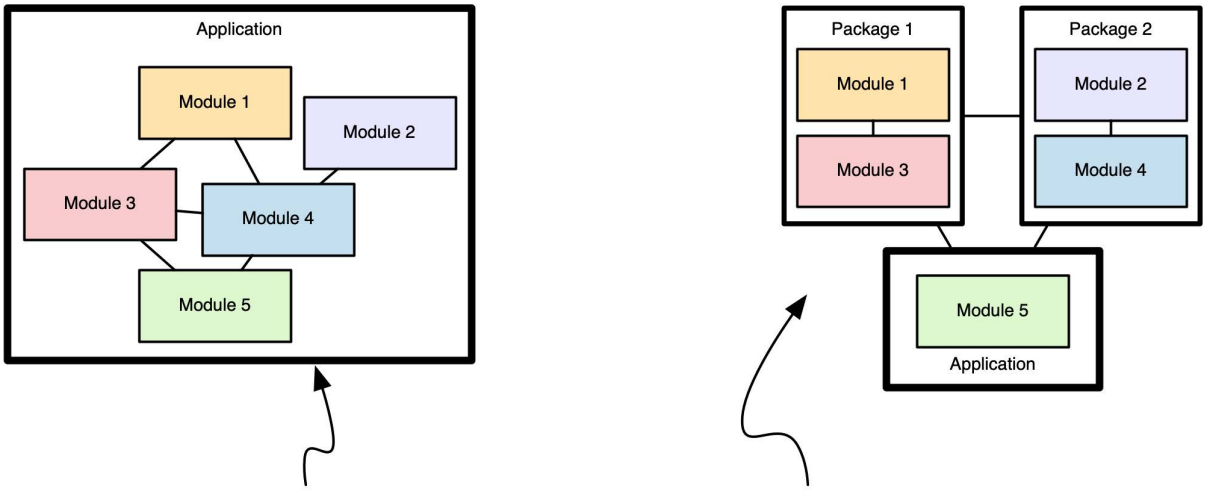

Figure 1.3 Packaging can reduce unexpected interdependence between areas of code by introducing stronger boundaries.

You might've felt frustration in the past when you found out a piece of code meant only for use internal to a module was being used widely throughout the code. Each time you update that "internal" code you need to go update usages elsewhere. This high churn environment can lead to bugs when you don't propagate a change everywhere, leaving you or your team that much less productive.

Well-encapsulated, highly-cohesive code will change rarely even when used widely. This kind of code is sometimes labeled "mature." Mature code is a great candidate for distributing as a package because you won't need to republish it frequently. You can get a start in packaging by extracting some of the more mature code from your code base, then use what you know about cohesion and encapsulation to bring less mature code up to snuff.

1.2.2 Promoting clear ownership of code

Teams benefit from clear ownership over areas of code. Ownership often goes beyond maintaining the behavior of the code itself. Teams build automation to streamline unit testing, deployment, integration testing, performance testing, and more. That's a lot of plates to keep spinning at once. Keeping the scope of a bounded area of code small so that a team can own all these aspects will ensure the code's longevity. Packaging is one tool for managing scope.

The encapsulation you create through packaging code enables you to develop automation independent of other code. As an example, automation for a big ball of mud may require you to write conditional logic to determine which tests to run based on which files changed. Alternatively, you might run all the tests for every change, which can be slow. Creating packages that you can test and publish independently of other code will result in clearer mappings from source code to test code to publication code (figure 1.4).

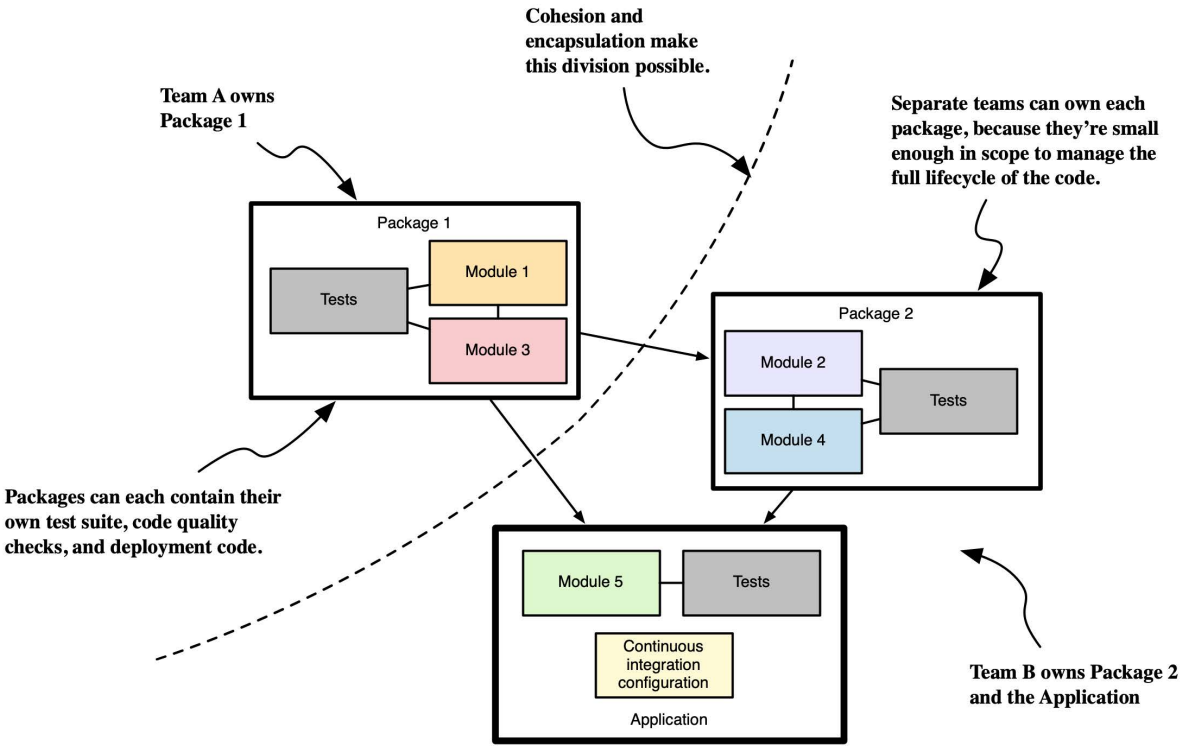

Figure 1.4 Teams can take full ownership over individual packages, defining how they want to manage the development, testing, and publishing lifecycle.

A clear delineation of purpose for a package makes it likelier to have a clear delineation of ownership. If a team isn't sure what they're committing to by taking ownership of some code, they're going to be wary. Try providing a package with a clear scope, story, and operator's manual to see how the mood shifts.

1.2.3 Decoupling implementation from usage

You may have heard the term *loose coupling* to describe the level of interdependence between areas of code.

IMPORTANT **Definition**

Coupling is a measure of the interdependence between areas of code. Loosely coupled code provides multiple avenues of flexibility so you can implement and choose from a variety of execution strategies instead of being forced down a particular path. Two pieces of code with low coupling have little or no dependence on each other, and they can be changed at different rates.

The cohesion and encapsulation practices you read about earlier in this chapter are a way to reduce the likelihood of tight coupling due to poor code organization. Highly cohesive code will have tight coupling within itself, and loose coupling to anything outside its boundary. Encapsulation exposes an intentional API, limiting any coupling to that API. Your choices about packaging and encapsulation, then, help you decouple your consumers from implementation

details in your code. Packaging also makes it possible to decouple consumers from implementation through versioning, namespacing, and even the programming language in which software is written.

In a big ball of mud, you're stuck running whatever code is in each module. If you or someone on your team updates a module, all code using that module needs to accommodate the change immediately. If the update changes a call signature or a return value, it may have a wide blast radius. Packaging significantly reduces this restriction (figure 1.5).

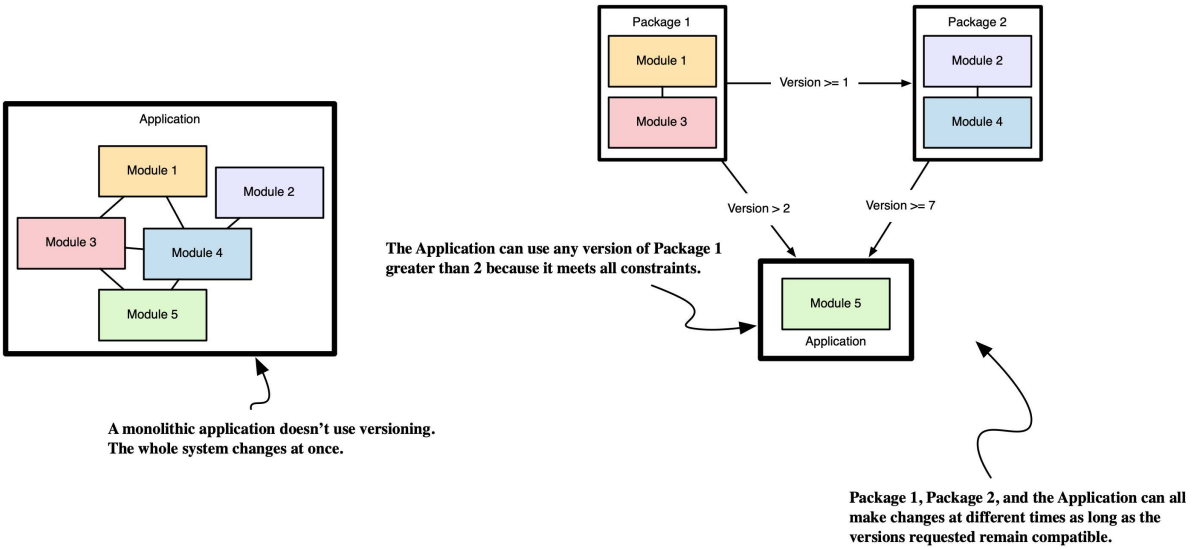

Figure 1.5 Packaging provides flexibility so two areas of code can evolve at different rates

Imagine if each update to the `requests` package required you to react immediately by updating your own code. That would be a nightmare! Because packages version the code they contain, and because consumers can specify which version they want to install, a package can be updated many times without impacting consuming code. Developers can choose precisely when to incur the effort of updating their code to accommodate a change in a more recent version of the package.

Another point at which you can decouple code is *namespacing*. Namespaces attach values and behavior to human readable names. When you install a package, you make it available at the namespace it specifies. As an example, the `requests` package is available in the `requests` namespace.

Different packages can have the same namespace. This means they could conflict if you install more than one of them, but it also makes something interesting possible: this flexibility in namespaces means packages can act as full alternatives to one another. If a developer creates an alternative to a popular package that's faster, safer, or more maintainable, you can install it in place of the original as long as the API is the same. As an example, the following packages all provide roughly equivalent MySQL (www.mysql.com) client functionality (specifically, they implement some level of compatibility with PEP 249, www.python.org/dev/peps/pep-0249/):

- mysqlclient (github.com/PyMySQL/mysqlclient)
- PyMySQL (github.com/PyMySQL/PyMySQL)
- mysql-python (github.com/arnaudsj/mysql-python)
- oursql (github.com/python-oursql/oursql)

Finally, Python packaging can even decouple usage in Python from the language in which a package is written! Many Python packages are written in C and even Fortran for improved performance. Package authors can provide pre-compiled versions of these packages alongside versions that can be built from source by the consumer if needed. This also makes packages more portable, decoupling developers somewhat from the details of the computer or server they're using. You'll learn more about packaging build targets in a later chapter.

If for no other reason, you might like to package some of your code to experiment with the freedom of version decoupling. See how your versioned packages evolve over time. Those that change quickly may point to low cohesion because the code has many reasons to change. On the other hand, it may indicate only that the code is still maturing. At the least, these data points will be observable! You'll learn more about versioning in a later chapter.

1.2.4 Filling roles by composing small packages

The act of extracting code into multiple packages is a bit like *decomposition*. Successful decomposition requires a good handle on loose coupling. Decomposing code is an art that separates pieces of code so they can be recombined in new ways (for a wonderfully concise rundown of decomposition and coupling, see Josh Justice, "Breaking Up Is Hard To Do: How to Decompose Your Code", *Big Nerd Ranch*, www.bignerdranch.com/blog/breaking-up-is-hard-to-do-how-to-decompose-your-code/).

By packaging smaller areas of your code, you'll uncover opportunities to move code from accomplishing a very specific calculation to *fulfilling a role*. As an example, the `requests` package isn't built to make just one, specific HTTP request. It fills a general role as an HTTP client. Some of your code may be very specific now, but as you package it up and find new areas where you need similar behavior, you may see that your newly-minted package can be generalized a bit to fill a role.

As you work on revamping your software for CarCorp, you remember that a major portion of the code deals with the car's navigation systems. You realize that with a bit of tweaking, the navigation code will also work for Acme Auto's vehicles. This code could fill the role of communicating with vehicle navigation systems. Because you've learned that packages can depend on other packages, and because your navigation system code is already fairly cohesive, you commit yourself to creating not one but two packages before your next CarCorp meeting.

SIDEBAR **A composition success story**

You can see great examples of composition at play in packaging through Python frameworks like Django (www.djangoproject.com). Django is itself a package, and because it's built as a plugin-based architecture, you can extend its functionality by installing and configuring additional packages. Peruse the hundreds of packages listed on Django Packages (djangopackages.org) to see the kind of wide adoption the packaging approach enjoys.

Thinking about composition and decomposition highlights the fact that distribution packages can exist at any size, just as functions, classes, modules, and import packages do. Look to cohesion and decoupling as guiding lights to strike the right balance. One hundred distribution packages that each provide a single function would be a maintenance burden, and one distribution package that provides one hundred import packages would be about the same as having no package at all. If all else fails, always ask yourself, "What role do I want this code to fill?"

Now that you've learned that packaging can help you write cohesive, loosely coupled code with clear ownership that you can deliver to consumers in an accessible way, I hope you're rolling up your sleeves to dive into the details.

1.3 Summary

- Packages archive software files and metadata about the software, such as the name, creator, license, and version.
- Package managers automate installing packages and managing the interdependencies between them.
- The packaging process has a number of pitfalls that can be overcome with tools and a repeatable process.
- Software repositories host published packages for others to install.
- Packaging is a great way to separate and encapsulate code with high cohesion.
- Packaging can be used as a decoupling tool to gain flexibility in developing and maintaining code.
- Versioned packages are a great way to reduce churn across the code base for each individual update.

2 Preparing for package development

This chapter covers

- Managing multiple Python versions using asdf
- Managing virtual environments using venv
- Using multiple Python versions and virtual environments at the same time

Getting started with a new project can be daunting. Like any creative work, staring at a blank screen with all the decisions ahead of you can quickly lead to analysis paralysis. Taking the first few steps can help you develop some comfort and a willingness to explore. Practicing early with the tools you'll use most frequently builds some comfort as well. In this chapter you'll use asdf and venv to create a development environment for the package you'll work on for the rest of this book.

If you haven't already, visit appendix A to install the tools you'll need for this chapter.

2.1 Managing multiple base Python versions with asdf and python-launcher

As you think more about the potential success of your CarCorp engagement, you realize that if the package you're working to publish becomes popular, people using a variety of Python versions might want to install and use it. It isn't likely they'll always be running the latest version of Python on their production systems. It's a good practice to explicitly state the range of Python versions your package supports, and test your package across all those versions. The first step is to install a few recent versions of Python.

asdf is a tool for installing different versions of software and switching between them. Now that you have asdf and its Python plugin installed and at your disposal, you can use it readily to

install new base Python versions.

Now that you've installed several base Python versions, verify that asdf knows about them by running the following command:

```
$ asdf list python
```

You should see output similar to the following, with slightly different versions based on what you installed:

```
3.11.0
3.10.2
3.9.5
```

Now, see the versions that python-launcher knows about using the following command:

```
$ py --list
```

You should see output similar to the following, with slightly different versions based on your operating system and what you installed:

```
3.10  /Users/<you>/.asdf/shims/python3.10
3.9   /Users/<you>/.asdf/shims/python3.9
3.8   /Users/<you>/.asdf/shims/python3.8
3.7   /Users/<you>/.asdf/shims/python3.7
2.7   /usr/bin/python2.7
```

Note that most of these versions mention asdf, which you made available through the `asdf global python` command. But the last version points somewhere else—that's the *system* Python. asdf interacts with your shell's `$PATH` variable, which is how it can switch where the `python` command resolves.

python-launcher will, by default, use the highest available version of Python it can find. As an example, if you installed Python 3.10, 3.9, and 3.8, python-launcher would prefer to use Python 3.10 by default. You can control which base Python version you get using python-launcher's version flag, but you'll see how to use it more smartly for projects in the following section.

SIDEBAR **Exercise**

If you used asdf to install Python 3.10, 3.9, and 3.8, and ran `asdf global python 3.9 3.8`, which version does the following command return?

```
$ py -V
```

Now that you understand how to install and switch between base Python versions, you'll learn how to create isolated environments from them using venv.

2.2 Managing Python virtual environments

When you install Python, it ships with the packages that are available in Python's *standard library*.

IMPORTANT **Definition**

A standard library defines which functionality is considered the core part of a programming language. The standard library of a language is built into the language or its installation process and is available by default after installing the language's software on your system.

Python's standard library is extensive compared with some languages, but even then it doesn't provide all the functionality you might need for your projects. Python packages, the Python Package Index (PyPI), and the pip package manager exist to share software that extends beyond, or provides alternatives to, the Python standard library.

Imagine that when you first started your project with CarCorp, you used pip to install a few packages like `requests`. You also had some other packages installed from an earlier project for Vehicle Ventures. Did you notice that all these packages ended up together in one place, regardless of the project you were using them for?

By default, pip installs packages in a location related to the Python version with which pip itself was installed, known as the *site packages* directory. That is, when you install Python 3.7 and use the copy of pip that comes with it, packages you install will be stored in Python 3.7's site packages directory. Installing all packages to this site packages directory might be manageable enough for a while, but what happens when you need different package versions for different projects? What happens if you need to list the minimum dependencies required for a single project? With the site packages directory full of packages from any and every project, these hurdles become difficult or impossible to address.

One way to solve these problems is to *isolate* the packages for each project. In isolation, you can keep a list of each project's minimum required dependencies. What's more, one project is free to use `requests==2.1.0` even though another project uses `requests==2.24.0`. You learned about the value of decoupling in chapter 1. Isolation of package dependencies decouples your projects from each other. You can achieve this isolation in Python using *virtual environments*.

IMPORTANT **Definition**

Python virtual environments are an isolated copy of Python with an isolated site packages directory. The copy of pip in the virtual environment's Python installs packages to its isolated site packages directory, keeping them separate from other environments.

A virtual environment isn't all that different from a normal Python installation, conceptually. Instead of installing Python 3.7 and installing all your projects' dependencies into it, imagine installing Python 3.7 several times and giving each installation a unique name corresponding to each of your projects. You could then use each uniquely-named Python for its corresponding project (figure 2.1). This isn't far off from how virtual environments work in practice.

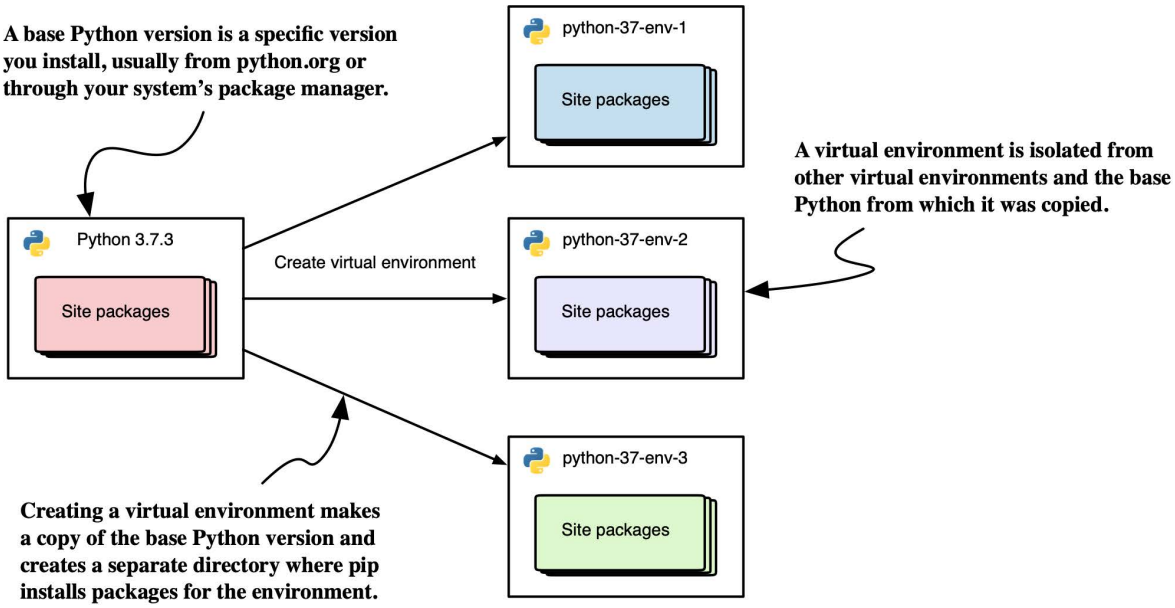

Figure 2.1 Virtual environments create isolated copies of Python and pip with their own installation directory for packages.

When you use Python within a virtual environment, you'll be using a *copy* of the base Python version that created the environment.

To test your package, you need to install packages not only in isolation from other projects, but also across many base versions of Python. As the number of Python versions your project supports grows, it can become tedious to manage all the virtual environments and their Python installations manually (figure 2.2).

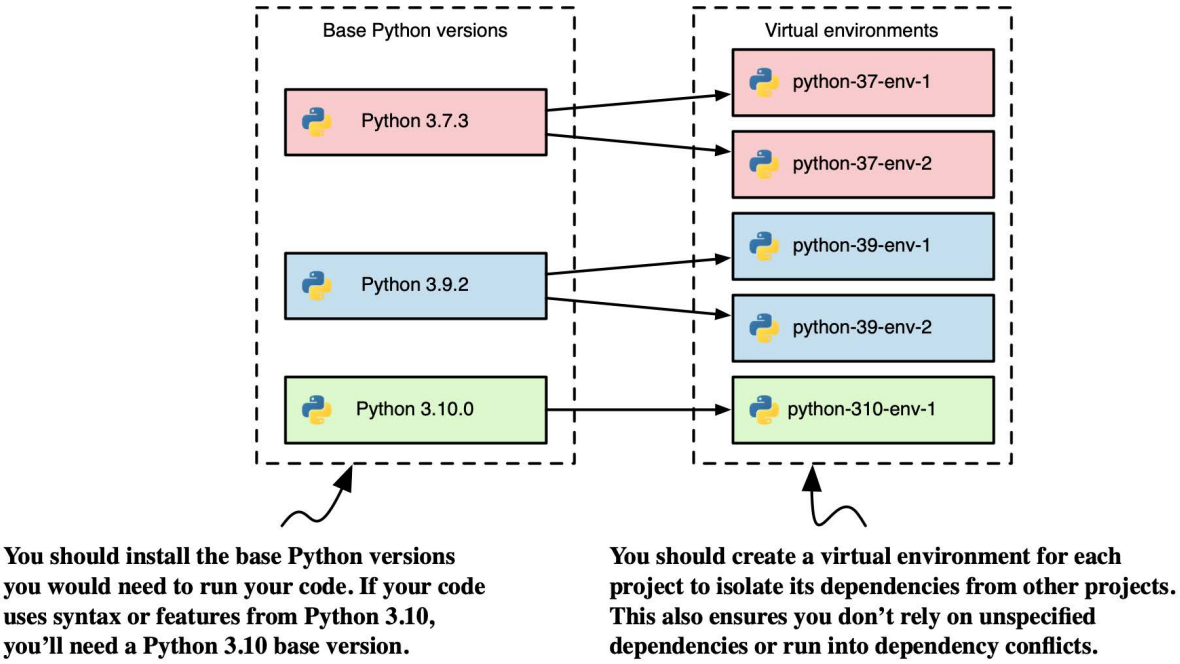

Figure 2.2 Many base Python versions, each with many virtual environments created from them, may exist on a single system.

You might be seeing the value tooling has in keeping all these things organized. Whereas asdf helps you install and manage base Python versions, venv helps you create virtual environments from those base Python versions.

2.2.1 Creating virtual environments with venv

To make a base Python version available on your system, you use asdf to install it from the source code on the internet. To create a virtual environment, you make a copy of an installed base Python version with a unique name.

To create a virtual environment, use the `venv` module from the base Python version and pass it a name for the virtual environment's directory. It's a common convention to call this directory `.venv/`. Create a virtual environment in your project now by running the following commands:

```
$ cd $HOME/code/first-python-package/
$ py \          ❶
    -3.10 \     ❷
    -m venv \   ❸
    .venv       ❹
```

❶ Use python-launcher to choose a base Python version

❷ Pass a version flag

❸ Creates a new virtual environment using the builtin `venv` module

❹ Create `.venv/` directory in the current working directory

You won't see any output if the command is successful, but you should see a `.venv/` directory created. python-launcher will pick up on the presence of this new virtual environment and use it by default whenever you're in this directory or its child directories. You can verify this by running the `py` command with no arguments. The interpreter that starts will match the base Python version you used to create the virtual environment, and you can using the following code to be sure it's the virtual environment's copy of Python:

```
>>> import sys
>>> sys.executable
'/Users/<you>/code/first-python-package/.venv/bin/python'
```

If you pass a version flag to python-launcher, you'll still get the base version. As an example, you should see something similar to the following when using `py -3.9`:

```
>>> import sys
>>> sys.executable
'/Users/<you>/.asdf/installs/python/3.9.3/bin/python3.9'
```

To prove that your virtual environment is isolated from the base Python version from which it was created, first run the following commands from your project's directory to install the `requests` package in the virtual environment:

```
$ py -m pip install requests
$ py -m pip list
Package    Version
---------- -------
pip        21.1.2
requests   2.25.1
setuptools 57.0.0
wheel      0.36.2
```

Now, confirm that it isn't installed in the base Python version by explicitly passing the `-3.10` version flag:

```
$ py -3.7 -m pip list
Package    Version
---------- -------
pip        21.1.2
setuptools 57.0.0
wheel      0.36.2
```

Going forward, you'll be able to use the `py` command in your project and be sure you're always getting the copy of Python from your project's virtual environment unless you explictly ask for a different (base) Python. This can reduce your cognitive load, because you don't need to remember to activate or deactivate the virtual environment manually each time you start or stop work on the project.

You've learned the ins and outs of managing Python versions and virtual environments with asdf and venv. You're ready to move on to creating the contents of your first Python package.

2.3 Summary

- Virtual environments to decouple and isolate the dependencies of your different Python projects.
- asdf and venv are a complementary set of tools for managing base Python versions and virtual environments.
- Use python-launcher to reliably get the right version of Python.

3 The anatomy of a minimal Python package

This chapter covers

- The Python package build system
- Building a package using setuptools
- The directory structure of a Python package
- Building a package for multiple targets

Python package builds are the product of coordination between a few different tools driven by a standardized process. One of the biggest choices you have as a package author is which set of tools to use. It can be difficult to assess the nuances of each, especially if you're new to packaging. Fortunately, tools are standardizing around the same core workflow, so once you learn it you've got the agility to switch between tools with minimal effort. This chapter covers what each category of these tools accomplishes and how they work together to produce a package, as well as how package builds vary for different systems.

TIP **If you haven't yet installed `build`, pause here and head over to appendix B to do so now.**

3.1 The Python build workflow

The following sections cover what happens when you build a package, and what you need to do to build a package successfully. You first need to learn about the pieces of the Python build system itself.

3.1.1 Parts of the Python build system

In the root directory for your package, start by running `build` using the following command:

```
$ pyproject-build
```

Because your package has no content yet, you should see an error like the following:

```
ERROR Source /Users/<you>/code/first-python-package does not appear to be
<linearrow /> a Python project: no pyproject.toml or setup.py
```

The output makes two file suggestions. `pyproject.toml` is the newer standard file for configuring Python packaging introduced in PEP 518 (www.python.org/dev/peps/pep-0518/), and should be preferred unless a third-party tool you want to use is only compatible with `setup.py`. The file uses TOML (toml.io/en/), an INI-like language, to split configuration into relevant sections. Teaching TOML is beyond the scope of this book, but the pieces you need for your packaging will be included and explained where needed in this book. Create the `pyproject.toml` file using the following command to correct the error:

```
$ touch pyproject.toml
```

Run the `build` command again. This time the build should run successfully, and you should see a large amount of output with a few notable lines as shown in listing 3.1. What's happening here? At a high level, the build command consumes your source code and the metadata you supply, along with some files it generates, to create:

1. **A source distribution package**. A Python source distribution, or `sdist`, is a compressed archive file of the source code with a `.tgz` extension.
2. **A binary distribution package**. A Python built distribution package is a binary file. The current standard for built distributions is what's known as a wheel or `bdist_wheel`, a file with a `.whl` extension.

Listing 3.1 The result of building an empty Python package

```
...
Successfully installed setuptools-57.0.0 wheel-0.36.2                    ❶
...
running sdist                                                            ❷
...
warning: sdist: standard file not found:                                 ❸
<linearrow /> should have one of README, README.rst, README.txt, README.md

running check
warning: check: missing required meta-data: name, url                    ❹

warning: check: missing meta-data: either (author and author_email)      ❺
<linearrow /> or (maintainer and maintainer_email) should be supplied

creating UNKNOWN-0.0.0                                                   ❻
...
Creating tar archive                                                     ❼
...
Successfully installed setuptools-57.0.0 wheel-0.36.2
...
running bdist_wheel                                                      ❽
...
creating '/Users/<you>/code/first-python-package/dist/tmpgdfzly_7/'      ❾
<linearrow />UNKNOWN-0.0.0-py3-none-any.whl' and adding
<linearrow /> 'build/bdist.macosx-11.2-x86_64/wheel' to it
```

❶ Setuptools and the `wheel` package are used for the build backend

❷ The source distribution package is built by the `build_sdist` hook

❸ The build process expects a `README` file in one of a few formats

❹ The build process expects a name and a URL for the package

❺ The build process expects an author or maintainer of the package

❻ The package is called `UNKNOWN` because no name was specified

❼ The source distribution is a compressed archive file

❽ The binary wheel distribution package is built by the `build_wheel` hook

❾ The binary wheel distribution is a `.whl` file

Because you haven't supplied any metadata yet, the build process alerts you to the fact that it's missing some important information like a `README` file, the author, and so on. Adding this information is covered later in this chapter.

Notice that the build process installs the `setuptools` and `wheel` packages. Setuptools (setuptools.readthedocs.io) is a library that was, for a long time, one of the only ways to create Python packages. Now, Setuptools is one of a variety of available *build backends* for Python package builds.

IMPORTANT **Definition**

A build backend is a Python object that provides several required and optional hooks that implement packaging behavior. The core build backend interface is defined in PEP 517 (www.python.org/dev/peps/pep-0517/#build-backend-interface).

A build backend does the logistical work of creating package artifacts during the build process, namely through the `build_sdist` and `build_wheel` hooks. Setuptools uses the `wheel` package to build the wheel during the `build_wheel` step. The `build` tool uses Setuptools as a build backend by default when you don't specify one.

The presence of build *backends* may leave you wondering if there may be build *frontends* as well. As it turns out, you've been using a build frontend already. The `build` tool is a build frontend!

IMPORTANT **Definition**

A build frontend is a tool you run to initiate building a package from source code. The build frontend provides a user interface and integrates with the build backend via the hook interface.

To recap, you use a build frontend tool like `build` to trigger a build backend like Setuptools to create package artifacts from your source code and metadata (figure 3.1).

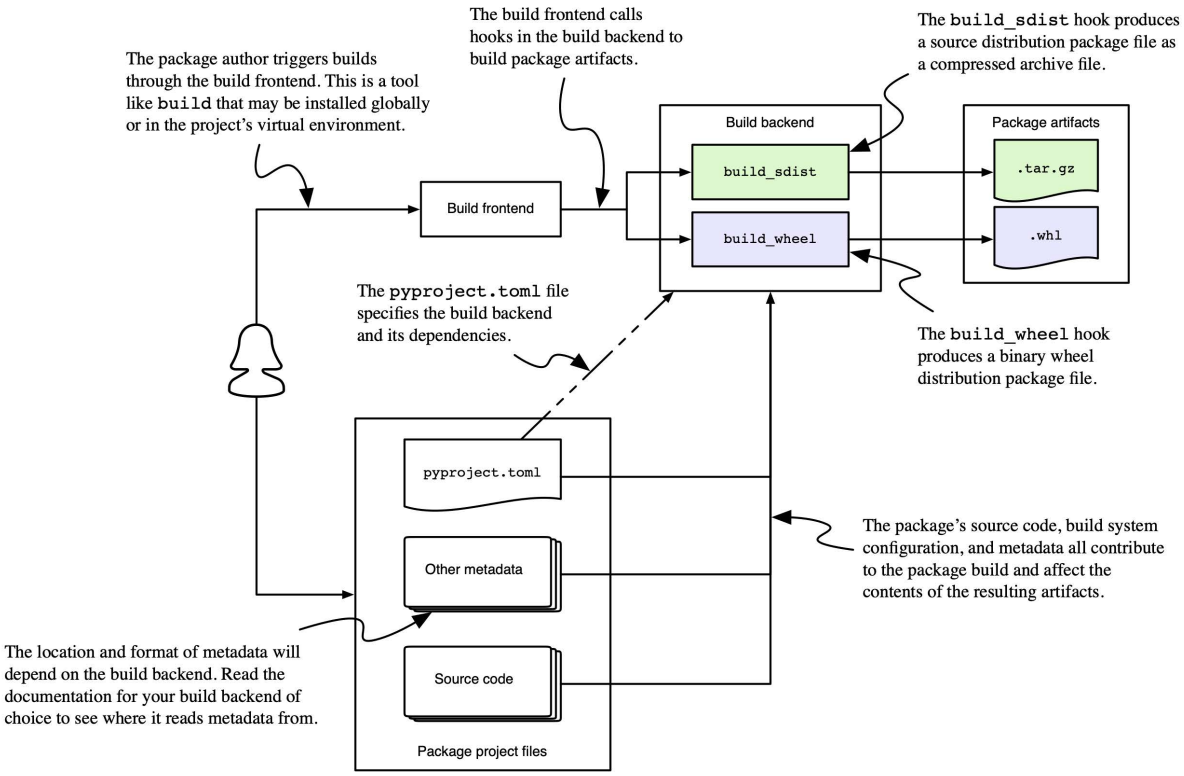

Figure 3.1 The Python build system consists of a frontend user interface that integrates with a backend to build package artifacts.

Because the build process creates package artifacts, you can now check the effect of running the build. List the contents of the root directory for your package now. You should see the following:

```
$ ls -a1 $HOME/code/first-python-package/
.
..
.venv/
UNKNOWN.egg-info/
build/
dist/
pyproject.toml
```

The `UKNOWN.egg-info/` and `build/` directories are intermediate artifacts. List the contents of the `dist/` directory, where you should see the source and binary wheel package files:

```
$ ls -a1 $HOME/code/first-python-package/dist/
UNKNOWN-0.0.0-py3-none-any.whl
UNKNOWN-0.0.0.tar.gz
```

SIDEBAR **Other build system tools**

As I mentioned earlier, other options exist for both build frontends and backends. Some packages provide both a frontend and a backend. Through the rest of this book, continue using `build` and Setuptools as the frontend and backend for your builds.

If you want to explore some alternative build tools, check out poetry (python-poetry.org/) and flit (flit.readthedocs.io). Each build system makes different trade-offs between ease of configuration, capability, and user interface. As an example, flit and poetry are geared toward pure-Python packages whereas Setuptools can support extensions in other languages. Chapter 4 covers this in more detail.

You can switch to another build system in a couple of steps:

1. Install the new build frontend package
2. Update `pyproject.toml` to specify the new build backend and its requirements
3. Move the metadata about the package to the location expected by the new build backend

Recall that `build` used Setuptools as the fallback build backend because you didn't specify one. You can specify Setuptools as the build backend for your package by adding the lines in listing 3.2 to `pyproject.toml`. These lines specify the following:

1. *`build-system`*—This section describes the package build system.
2. *`requires`*—These are a list of dependencies, as strings, which must be installed for the build system to work. A Setuptools build system needs `setuptools` and `wheel` as you saw earlier in this chapter.
3. *`build-backend`*—This identifies the entrypoint to the build backend object, using the dotted path as a string. The Setuptools build backend object is available at `setuptools.build_meta`.

These represent the complete configuration you need to specify the build backend.

Listing 3.2 A build system backend specification to use Setuptools

```
[build-system]                                          ❶
requires = ["setuptools", "wheel"]                      ❷
build-backend = "setuptools.build_meta"                 ❸
```

❶ Opens a new TOML section

❷ List of package names as strings

❸ Dotted path to the object as a string

Once you've added the build system information, run the build again. Nothing should change in the output: you've just locked in Setuptools as the explicit backend instead of letting `build` fall back to it as a default. Now that you've got a handle on the Python package build system, you need to add some metadata about your package.

3.2 Authoring package metadata

You learned that each build backend may look for package metadata in different places and formats. For the Setuptools backend, you can specify static metadata in an INI-style file called `setup.cfg` in the root directory of your package. You'll add sections of key-value pairs to this file that provide information about the package and its contents.

Some metadata is essential to build a package that can be identified properly. When you ran the build it resulted in files with `UNKNOWN-0.0.0` in the name, which is the result of some missing core metadata. Start by fixing these core metadata issues first.

3.2.1 Required core metadata

To fix the names of your package files, start by creating the `setup.cfg` file in the root directory of your package. Two fields are minimally required for a package—`name` and `version`. These distinguish a distributed version of your package from other packages and other versions of your own package. Add the fields to `setup.cfg` in a TOML section called `metadata`. It should look like the following:

```
[metadata]                    ❶
name = first-python-package   ❷
version = 0.0.1
```

❶ This is the "metadata" section.

❷ Sections contain one or more key-value pairs.

After you save the file, remove the `dist/` directory and run the build process again. List the contents of the newly-generated `dist/` directory, where you should see the following:

```
$ ls -a1 dist/
.
..
first-python-package-0.0.1.tar.gz
first_python_package-0.0.1-py3-none-any.whl
```

This confirms that you've supplied the name and version correctly. The build process recognized the values you supplied and used them to populate the filenames of the package artifacts. `UNKNOWN` has been replaced by a normalized version of `first-python-package`, and `0.0.0` has been replaced by `0.0.1` (table 3.1).

Before	After
`UNKNOWN-0.0.0.tar.gz`	`first-python-package-0.0.1.tar.gz`
`UNKNOWN-0.0.0-py3-none-any.whl`	`first-python-package-0.0.1-py3-none-any.whl`

To confirm that the package contains the intended files, you can manually inspect its contents. Change into the `dist/` directory and unpack the source distribution package using the following commands:

```
$ cd $HOME/code/first-python-package/dist/
$ tar -xzf first-python-package-0.0.1.tar.gz
```

This creates a `first-python-package-0.0.1/` directory next to the package file, containing the files packaged from your project along with a few generated files. You should see the following:

```
$ ls -1R first-python-package-0.0.1/
PKG-INFO                                                    ❶
first_python_package.egg-info
pyproject.toml                                              ❷
setup.cfg

first-python-package-0.0.1/first_python_package.egg-info:
PKG-INFO
SOURCES.txt
dependency_links.txt
top_level.txt
```

❶ The source distribution contains several generated files.

❷ The source distribution also contains files you created in your project.

You can also confirm that the metadata you specified has been faithfully reproduced in the package. Open either of the `PKG-INFO` files and take a look at the contents. The `PKG-INFO` file contains a normalized version of the metadata. You should see the following:

```
Metadata-Version: 2.1
Name: first-python-package   ❶
Version: 0.0.1               ❷
Summary: UNKNOWN             ❸
Home-page: UNKNOWN
License: UNKNOWN
Platform: UNKNOWN

UNKNOWN
```

❶ The package name you specified maps to the `Name` field.

❷ The package version you specified maps to the `Version` field.

❸ Fields you haven't specified yet show as `UNKNOWN`.

The package name and version you specified show up here, but there are several other fields that are still `UNKNOWN`. The build process is still alerting you to a missing URL, README, and author

information as well. Next, you'll fix these issues and flesh out the metadata a bit further to tell people about the package.

3.2.2 Optional core metadata

The name and version are the only two strictly required fields, per the core metadata specification (packaging.python.org/specifications/core-metadata/), but several other fields are indexed by search engines or surfaced in highly visible ways on sites like PyPI. If you want others to find and use your package, it's a good idea to supply information for as many of the fields as possible.

SIDEBAR **A rundown on package metadata**

If you want to learn all the different fields available and how they've evolved over time, the following PEPs (www.python.org/dev/peps/) deal with the package metadata specification:

- PEP 241: Metadata for Python Software Packages introduces the `PKG-INFO` file.
- PEP 301: Package Index and Metadata for Distutils introduces the idea of a centralized Python package index as well as classifiers to better distinguish Python packages.
- PEP 314: Metadata for Python Software Packages v1.1 augments PEP 241 with additional fields
- PEP 345: Metadata for Python Software Packages 1.2 augments PEP 314 with additional fields, changed fields, and deprecated fields
- PEP 566: Metadata for Python Software Packages 2.1 augments PEP 345 with the core metadata specification, stricter allowable values for package names, additional fields, and a canonical transform of package metadata to JSON.
- PEP 621: Storing project metadata in pyproject.toml defines a standard for providing package metadata in the `pyproject.toml` file as opposed to files like `setup.py` or `setup.cfg`. This has been accepted but doesn't yet have wide adoption by packaging tools.
- PEP 639: Metadata for Python Software Packages 2.2 proposes an approach to standardizing how licenses are specified for packages.

The core metadata specification provides the most current list of available fields and their format.

The build process is still alerting you to a missing URL and author information. Add the following fields to your metadata section, filling in your personal information where appropriate:

```
...
url = https://github.com/<username>/<package repo name>
author = Given Family
```

```
author_email = "Given Family" <given.family@example.com>
```

Run the build again, and you should no longer see the alerts about a missing URL and author. Unpack the source distribution file and view the `PKG-INFO` file again. You should see the following, with the new values you've added:

```
Metadata-Version: 2.1
Name: first-python-package
Version: 0.0.1
Summary: UNKNOWN
Home-page: https://github.com/<username>/<package repo name>       ❶
Author: Given Family                                               ❷
Author-email: "Given Family" <given.family@example.com>
License: UNKNOWN
Platform: UNKNOWN

UNKNOWN
```

❶ The `url` field maps to `Home-page`.

❷ The `author` and `author_email` fields map to `Author` and `Author-email`.

The summary is still showing as `UNKNOWN`. The summary is a short description of the package's purpose. You can think of this as an elevator pitch for your package: it's what people will see most often when they're searching for packages to use. If you're reading this book, chances are that you want to learn how to share your code. If you skimp on the metadata, chances are that no one will find it. Metadata ensures that your package will be as discoverable as possible later down the line. In Setuptools, the summary is called `description`. Add the `description` field to your metadata now:

```
...
description = This package does amazing things.
```

There's also that unlabeled `UNKNOWN` lurking at the end of the file. That space is for the package's long description, which can provide more details about how to install and use the package or what problems it solves. Recall that the build process is still complaining about a missing README file. You can fix both these issues in one pass by creating a README file and referencing it in the metadata.

Create a `README.md` file now, with content something like the following:

```
# first-python-package

This package does amazing things.

## Installation

```shell
$ python -m pip install first-python-package
```
```

In `setup.cfg`, you can now use the `long_description` field to reference your README file

using the special `file:` directive. The `file:` directive accepts the path to a file, relative to `setup.cfg`, whose contents should be taken as the value for the field. In addition, you also need to specify the `long_description_content_type` field to indicate that your README is something other than plain text. Because your file is a Markdown file, you should specify the `text/markdown` content type. Add both of these fields to your metadata now:

```
...
long_description = file: README.md
long_description_content_type = text/markdown
```

Run the build, extract the source distribution, and inspect `PKG-INFO` again. You should see the following:

- The `Summary` field is populated with the short description.
- The file now contains a `Description-Content-Type` with a value of `text/markdown`.
- The `UNKNOWN` at the end of the file is now replaced with the contents of your `README.md` file.

What's more, when you update your README file those changes will be pulled into the next version of the package you build. This automation reduces issues remembering to update your documentation in multiple places.

The license is the last `UNKNOWN` field you'll address for now, and it requires some special attention.

3.2.3 Specifying a license

Licenses are important because they help your users understand the conditions under which they're allowed to use your software. The detailed process of choosing a license is outside the scope of this book, but sites like Choose A License (choosealicense.com) guide you through the process by asking you what freedoms and restrictions you want to provide with your software.

Once you choose a license you need to declare that license alongside your code so that users can identify whether they can work with your software. Sites like GitHub automatically discover license information from a few files like `LICENSE` or `LICENSE.txt`. At the same time, you need to provide your license in your source and binary package distributions so people who install your package can view the license as well.

To properly identify your license of choice, and to include the license information in your built package distributions, use a combination of the following three fields:

- *`license`*—Specifies the identifier from the SPDX license list (spdx.org/licenses/) that corresponds to your chosen license.
- *`license_files`*—Specifies the path to one or more license files, relative to `setup.cfg`.
- *`classifiers`*—Specifies any relevant trove classifiers (pypi.org/classifiers/) your

package falls under for discovery purposes.

As an example, if you were to choose the MIT License (mit-license.org/), you'd place a copy of the license text in a `LICENSE` file in the root directory of your package, then add the following fields to your metadata:

```
...
license = MIT
license_files = LICENSE
classifiers =
    License :: OSI Approved :: MIT License
```

Now you've learned how to specify a variety of metadata about your package for the Setuptools build backend, and you've seen how the build system normalizes and uses that metadata when it builds distribution packages. The flow of metadata between input files and output files is summarized in figure 3.2.

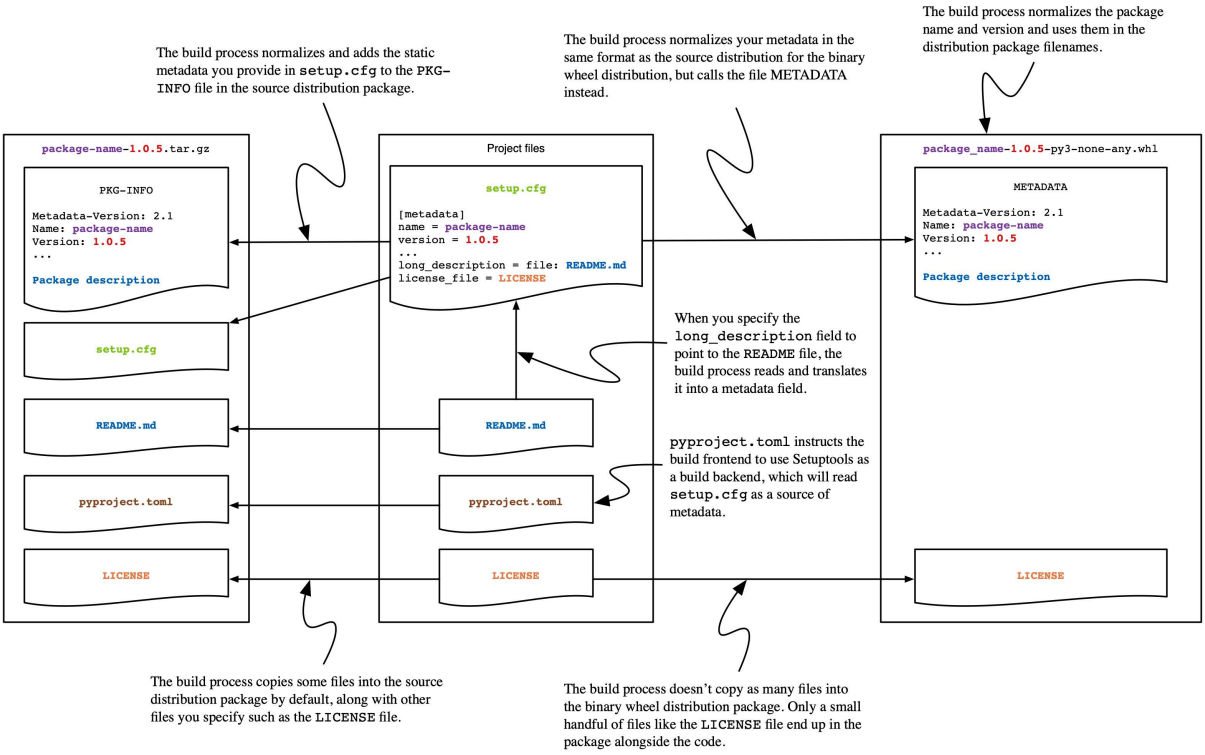

Figure 3.2 The flow of metadata between input project files and output distribution package files.

Now that you understand how the metadata flows from your project into the distribution package files, it's time to learn how your source code does the same.

3.3 Controlling source code and file discovery

Imagine you've finally finished creating a package, complete with 100% unit test coverage. You publish it, only to start getting reports of a bug. It turns out that you ran your tests against the raw source code instead of the packaged code CarCorp actually received when they installed the package, and you packaged the code incorrectly.

Python doesn't impose a specific directory structure for your code and your tests. This flexibility can be helpful, but it also leads naturally to multiple conventions. Some conventions are open to practices that lead you to create broken packages due to missing files or incorrect imports. Use a convention that discourages these practices by forcing you to test the packaged code. Running the packaging often manually can become tedious as a result, but tools to remove that burden are covered in chapter 5.

Keeping your tests separate from your implementation code altogether limits the possibility of running the tests against the raw source accidentally (See Ionel Cristian Mărieș, "Packaging a python library", blog.ionelmc.ro/2014/05/25/python-packaging/). In this model, the implementation modules and the test modules are both nested in their own directory:

```
some-package/                    ❶
    ...
    src/                         ❷
        some_package/            ❸
            __init__.py
            module_one.py
            module_two.py
            module_three.py
    test/                        ❹
        test_module_one.py
        test_module_two.py
        test_module_three.py
```

❶ This is the root directory of the distribution package.

❷ This directory contains the implementation code.

❸ This is the import package.

❹ This directory contains the tests.

NOTE **You'll learn more about testing the packaged code in chapter 5**

This approach also makes the purpose of each top-level directory clearer to someone who happens upon your project: the `src/` directory likely contains the implementation, and the `test/` directory likely contains code that tests the implementation. By separating the tests from the implementation you've also decoupled the structure of the two areas. Although it can make sense for the tests and the implementation to share a similar hierarchy, you're not *bound* to that.

SIDEBAR **Exercise**

Create the layout for your package. You should create the following structural pieces:

- **A `src/` directory**
- **A `test/` directory**
- **An import package called `imppkg` containing an empty module called `hello.py`**

After you're done, you should have the following directories and files in addition to the files you created earlier in this chapter:

```
first-python-package/
    ...
    src/
        imppkg/
            __init__.py
            hello.py
    test/
```

Run the build process and unpack the distribution file. Notice anything missing? The `imppkg` code files aren't there. Due to the flexibility of project layouts, and because you can distribute multiple import packages in a single distribution package, some build systems will require more specificity than you might think to discover your code. Setuptools needs to know the following:

- In which directories to look for packages.
- The names of specific (sub)packages to look for, or a directive to recursively find them all automatically.
- How to map any found package directories to different import names, if desired.

For the layout you've created, you can accomplish this with the following additional sections and fields in `setup.cfg`:

- *`[options]`*—This section provides additional options for Setuptools package builds.
- *`[options].package_dirs`*—This is a list of key, value pairs to map discovered directories to import paths. An empty key means the "root," such that any directory mapped to the root will be removed from the import path and only its child directories will be included.
- *`[options].packages`*—This is either an explicit list of packages, or the special `find:` directive that tells Setuptools to recursively search for any packages. `find:` is often the best choice, as you won't need to update it if you add new packages later.
- *`[options.packages.find]`*—This section provides options to the Setuptools package discovery process triggered by the `find:` directive.
- *`[options.packages.find].where`*—This tells Setuptools which directory to look in for packages.

Add those options to your `setup.cfg` now. The configuration should look something like listing 3.3.

Listing 3.3 A configuration for discovering packages with Setuptools

```
...

[options]
package_dirs =            ❶
    =src                  ❷
packages: find:           ❸

[options.packages.find]   ❹
where = src               ❺
```

❶ Configures which directories should map to which imports

❷ Maps the root to `src` so only its child directories will be included in import paths

❸ Tells Setuptools to find packages automatically instead of listing them

❹ Provides additional options to the `find:` directive

❺ Tells Setuptools to find packages in the `src/` directory

This configuration will cause Setuptools to search in the `src/` directory, find the `imppkg` package there, map the `src/imppkg/` directory to the `imppkg` import package, and include any modules within the `imppkg/` directory in the distribution package.

Notably, this configuration does not include anything from the `test/` directory. It's common to exclude tests from distribution packages to reduce the package size and because users rarely run the tests for third-party packages.

TIP **You may wish to add a field in `options.packages.find` to explicitly exclude any test modules from the package in case any accidentally make their way outside the `test/` directory in the future:**

```
...
exclude =
    test*
```

This will exclude any (sub)packages that begin with `test` from the distribution package.

Run the build process and unpack the distribution again. This time, it contains the `imppkg` package with its `hello.py` module faithfully reproduced there. You've got a working build! Although you've successfully packaged your Python files, there's still one configuration needed to ensure that non-Python files are included in your package.

3.4 Including non-Python files in a package

CarCorp has received your latest package and the bug they'd been dealing with is fixed. Unfortunately, a new bug has reared its head. The JSON file containing input data seems to be missing!

You've successfully packaged up your Python code and your metadata, but you haven't accounted for non-Python files yet. Create a `data.json` file in the same directory as your `hello.py` module now. Run the build process and observe that the `data.json` file is not present in the distribution.

With Setuptools, one of the most straightforward approaches to including non-Python files is using the `MANIFEST.in` file. This file contains directives that specify how to treat a matching set of files. The directives deal with including or excluding, and have varying levels of granularity (figure 3.3).

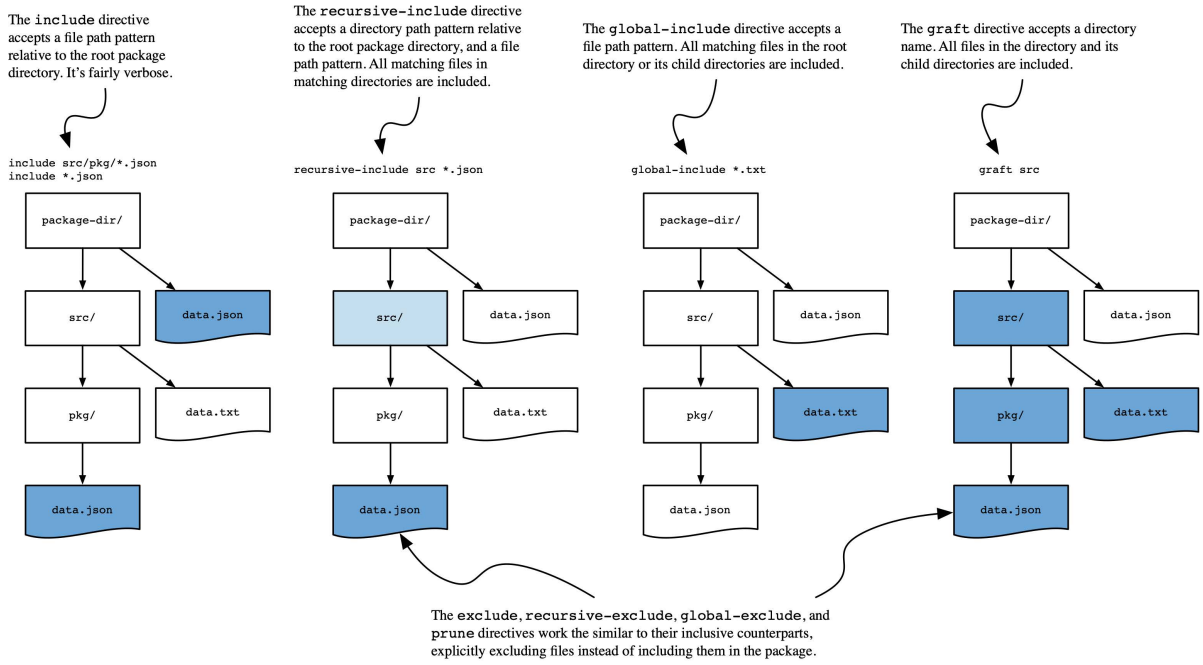

Figure 3.3 MANIFEST.in file directives to include non-Python files in packages

One of the quickest places to start is including all files in the `src/` directory, and recursively excluding some files generated by Python. You can achieve this by creating the `MANIFEST.in` file and adding the following content:

```
graft src                                    ❶
recursive-exclude __pycache__ *.py[cod]      ❷
```

❶ Include all files from the `src/` directory

❷ Except `__pycache__` directories or files that end in `.pyc`, `.pyo`, or `.pyd`

Run the build process and check the source distribution for the `data.json` file. Now, check the binary wheel distribution. The `data.json` file isn't present. You can tell Setuptools to include any non-Python files from the source distribution into the binary distribution by adding the following field to the `[options]` section of your `setup.cfg` file:

```
...
include_package_data = True
```

The binary wheel distribution file is now configured such that the `data.json` file is included.

SIDEBAR **Exercise**

Run the build process one more time and unpack both distributions. List the contents and confirm the presence of the `data.json` file. For reference, the file is located in the following places for each distribution:

```
$ ls <unpacked sdist>/src/imppkg/
__init__.py
data.json
hello.py

$ ls <unpacked bdist_wheel>/imppkg/
__init__.py
data.json
hello.py
```

You've learned how your source code, metadata, and supporting files can all be packaged together into single-file distributions. You also learned the different pieces of the Python build system and how they interoperate to produce the package file. You're ready for chapter 4, where you'll get into the specifics of a project, installing third-party dependencies, and building for multiple target systems.

3.5 Summary

- A Python package build needs a build frontend, and build backend, your source code, and your metadata.
- Build frontends and backends can be swapped for alternatives, but use the same core workflow.
- Packages rely on core required metadata to build properly, and systems rely on additional metadata to provide a rich discovery and browsing experience.
- Structure may differ from project to project, and build backends must be configured accordingly to package the right code.
- Build backends may need additional configuration to package non-Python files.

4 Handling package dependencies, entry points, and extensions

This chapter covers

- Defining dependencies for your package
- Making functionality available as command-line tools
- Packaging extensions written in C

You're just getting ready to start adding your groundbreaking functionality into your Python package for CarCorp when they call with a few last-minute requests. They want you to make sure it's *really fast* and that it can be run as a standalone command because their developers aren't as well-versed in Python as you are. You haven't even delivered the first version of your package yet, and already the requirements are growing! Before you panic, take a breath and read this chapter to learn more.

4.1 A package for calculating vehicle drift

Imagine the software you've been working on for CarCorp will help them guide their vehicles on the road. During their testing they've observed that the vehicles tend to drift around the road more than they'd like, and they've started measuring it. Although they have the raw data, they don't have a great way of measuring the impact of any potential improvements they make.

The package you're building will provide utilities for CarCorp to gain visibility into this issue. The first thing you'll do is provide a way to calculate the average drift in millimeters per second over a given distance. The vehicles measure their drift rate about one million times during each run through the five-kilometer testing course. Your package will consume these measurements as a list of floating point numbers and calculate their *harmonic mean*.

SIDEBAR **Harmonic mean**

The harmonic mean is different than the more common arithmetic mean and is the correct calculation to use when you want the average rate instead of the average of rates. Peter A. Lindstrom shows some examples in The Average of Rates and the Average Rate (www.comap.com/FloydVest/Course/PDF/Cons25PO.pdf).

You can calculate the harmonic mean of the drift by dividing the total number of measurements by the sum of reciprocals of the measurements:

$$H = \frac{N_{measurements}}{\frac{1}{m_1} + \frac{1}{m_2} + \frac{1}{m_3} + \ldots} H = \frac{N_{measurements}}{\frac{1}{m_1} + \frac{1}{m_2} + \frac{1}{m_3} + \ldots}$$

With one million inputs, this calculation might take some time. You can see why CarCorp emphasized that they want speed. When examining code performance, it's best to profile the code instead of speculating about the impact of improvements (Hillard, Dane. "Designing for High Performance." *Practices of the Python Pro*, Manning Publications, 2020, pp. 72–76.). Before you get too much further, you must first observe how the Python version fares.

SIDEBAR **Exercise**

In the root directory of your package, create a `harmonic_mean.py` module. In this module, write a `harmonic_mean` function that accepts an arbitrarily long list of floating point numbers and returns their harmonic mean.

Now that you've written a Python implementation of the harmonic mean calculation, you can use the built-in `timeit` module (docs.python.org/3/library/timeit.html) to measure its performance. When you profile code, you should reduce it to the minimal portion for which you want to measure the performance to ensure an accurate picture when comparing solutions. The `timeit` module enables you to separate setup code from the code you want to measure by passing the setup code as a string to the `--setup` option. The setup code will run only once and will not be counted toward the measurement of your code. You can invoke the module directly with `py -m timeit` and any arguments you want to pass. You can use the `--setup` option multiple times to separate multiple expressions, or use it a single time by separating expressions with a semicolon, as showing in the following snippet:

```
$ py -m timeit \   ❶
    --setup '<SETUP EXPRESSION 1>' \
    --setup '<SETUP EXPRESSION 2>' \
    '<MEASURED CODE>`

$ py -m timeit \   ❷
    --setup '<SETUP EXPRESSION 1>; <SETUP EXPRESSION 2>' \
    '<MEASURED CODE>'
```

❶ Multiple setup expressions can be separated into multiple arguments.

❷ Setup expressions can also be separated by a semicolon in a single argument.

To avoid setup overhead in your profile measurement, you should perform any imports and create data inputs in the setup step. Because you need the `harmonic_mean` function and the `random.randint` function, those should be imported as setup steps. You also want to measure the performance of `harmonic_mean` against a true-to-life set of data. You can create a list of random integers as a setup step as well, and pass that list to the `harmonic_mean` function in the execution step. Your command should look something like the following snippet:

```
$ py -m timeit \
    --setup 'from harmonic_mean import harmonic_mean' \                  ❶
    --setup 'from random import randint' \                               ❷
    --setup 'nums = [randint(1, 1_000_000) for _ in range(1_000_000)]' \ ❸
    'harmonic_mean(nums)'                                                ❹
```

❶ Import the function you want to measure

❷ Import helper functions needed for setup

❸ Create data inputs ahead of time

❹ Use only the function you want to measure in the execution

Run the command now. The `timeit` module will print out the statistics of the profiling, including:

- How many times it ran the code to get an average execution time (the measurement loops)
- How many sets of measurement loops it ran
- The best execution time of all the sets of measurements loops

The following snippet shows the statistics of how the `harmonic_mean` function performed on my MacBook Pro with 16 GB of memory and a 2.2 GHz 6-core processor:

```
5 loops, best of 5: 52.8 msec per loop
```

The `timeit` module ran five sets of five-measurement loops, and ultimately found that the call to `harmonic_mean` could run in as short a time as 52.8 milliseconds on average. You may see similar results, but they can vary based on what hardware you have and what else your computer is using it for at the time of measurement. The `timeit` module tries to account for some of these factors using the measurement loops. At the end of the day, it's important to remember that profiling should be used to compare one solution to another in a relative manner.

Save the results of your profiling somewhere for later reference, because now you're going to see how you can speed up this calculation to the level CarCorp is hoping for.

4.2 Creating a C extension for Python

When you write code or install third-party packages, you're extending the functionality of your software beyond what Python alone can provide. Typically, though, you're still using Python to achieve that extension. Just as you can use packages to extend functionality, you can also create and use extensions written in other languages to improve performance. Because the reference Python interpreter is written in the C programming language, C is a common choice for these extensions. But people also write extensions in C++, Rust, and even Fortran.

You learned about Python build backends in chapter 3, and you used Setuptools to start building the skeleton of your package. Setuptools has strong capabilities for building extensions from other languages. Other build backends may offer differing levels of support for extensions. Any time you're considering switching your build backend, consider any candidate backend's ability to meet your needs in this area. For now, you'll continue using Setuptools to integrate a C extension into your package.

4.2.1 Creating C extension source code

Covering in depth the writing of C-level code for use in Python is beyond the scope of this book, but because extensions are a common need for numerical programming it's important to learn how you can integrate them into a Python package. As with Python build backends and frontends, extensions and the tools to build them can be swapped in and out of your project as needed, and the details are left up to whichever tools you decide to invest in.

To expose you to one available option that also has a low barrier to entry, you'll convert your `harmonic_mean` function into a C extension using *Cython* (cython.org/). Not to be confused with CPython, the reference implementation of Python, Cython is a compiler and language for creating Python C extensions. The Cython language is a superset of Python, and at its most basic can be used to speed up some Python code without requiring sweeping changes. The Cython compiler converts Cython source code to optimized C code, which will then be compiled during a package's build process (figure 4.1).

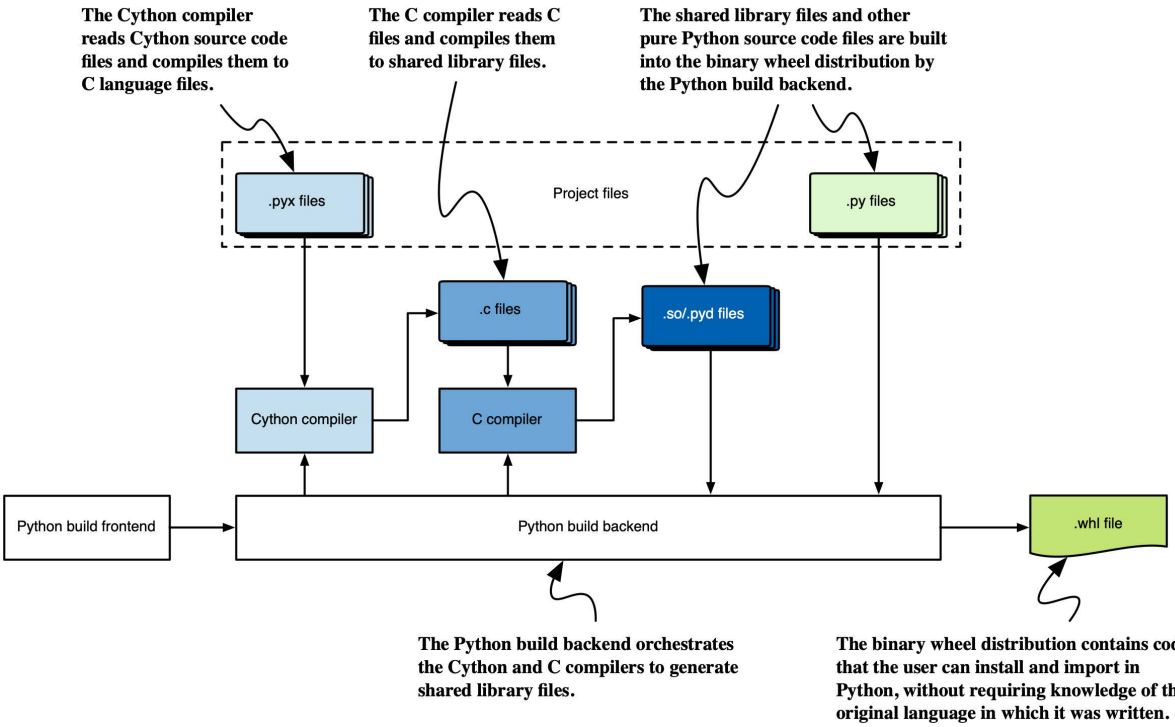

Figure 4.1 Extensions are compiled into shared libraries that are included in binary wheel distributions.

Cython source files end with `.pyx` and can contain Python or Cython code. Because the Cython language is a superset of Python, a valid Python program is also a valid Cython program. Rename your `harmonic_mean.py` module to `harmonic_mean.pyx` and move it into the `src/imppkg/` directory. Now that you have a Cython source code file, you need to integrate Cython into your package's build process.

4.2.2 Integrating Cython into a Python package build

You learned in chapter 3 that you can specify dependencies for a Python package's build process in the `pyproject.toml` file. Cython is a Python package itself, so you can add it to the list of build dependencies. This will ensure that Cython is installed before the build starts and is available to convert your Cython file to C code for compilation. Update the `requires` value in the `build-system` section of the `pyproject.toml` file to include `"cython"` now.

Next, you'll need to ensure that your Cython source files are included in your package. You also learned in chapter 3 that, when using Setuptools, you can include non-Python files in your package using the `MANIFEST.in` file. You used the `graft` directive to include all non-Python source files from the `src/` directory. That expression also includes all `.pyx` files. Run the build process using the `pyproject-build` command and confirm your Cython file ends up in the package as expected.

Now that you've created and included your Cython code in the package, you need to tell Cython to convert it to C code so that it will be compiled. Without this step, your friends at CarCorp

would only receive the raw `.pyx` file, which might leave them scratching their heads. To run Cython, you'll need to create one more file called `setup.py`.

SIDEBAR **Why do I see the `setup.py` file used so widely?**

The `setup.py` file has been a part of the Python packaging ecosystem for ages—it even predates Setuptools. Introduced in PEP 229 (www.python.org/dev/peps/pep-0229/) in 2000, its goal was to centralize where packaging configuration happens. You'll likely see that a wide variety of packages still use it, and although it's still necessary for some use cases the new build workflow and tools you learned in chapters 2 and 3 intend to replace `setup.py` in the long term. For pure-Python packages that don't need to determine any dynamic information at build time, you can use `pyproject.toml` to define your build and `setup.cfg` for configuration when using Setuptools as your build backend.

SIDEBAR **Exercise**

Create the the `setup.py` module in the root directory of your package. The module must:

1. Import the `setuptools.setup` function for hooking into the build process
2. Import the `Cython.Build.cythonize` function for identifying which Cython files need to be converted
3. Call `cythonize` with the path to your Cython file, relative to the root directory of the package
4. Call the `setup` function with the `ext_modules` keyword argument set to the result of the `cythonize` call

Now that you've created the `setup.py` module, you have a configuration that will convert your Cython files to C code and then compile it during the package build process. Run the build now. You should see a few new lines in the output that can help verify things are working as you expect:

- Cython gets installed as a build dependency
- Your Cython file gets pulled into the source distribution
- The `build_ext` process is triggered by your call to `setuptools.setup`
- The extension is compiled to a binary file (`.so` for macOS and Linux, or `.pyd` for Windows) and added to the binary wheel distribution

```
...
Collecting cython                                                    ❶
...
copying src/imppkg/harmonic_mean.pyx
<linearrow /> -> first-python-package-0.0.1/src/imppkg               ❷
...
running build_ext                                                    ❸
building 'harmonic_mean' extension
...
adding 'harmonic_mean.cpython-39-darwin.so'                          ❹
```

❶ Cython gets installed as a build dependency

❷ Your Cython file is copied into the source distribution

❸ Setuptools builds your extension

❹ The created binary file is copied into the binary wheel distribution

Additionally, you should see in the `dist/` directory that your binary wheel distribution file has changed names. Before, it was `first_python_package-0.0.1-py3-none-any.whl`. Now, its name will depend on the system you're using and the Python version you used. On my MacBook Pro with Python 3.10, the file is named `first_python_package-0.0.1-cp310-cp310-macosx_11_0_x86_64.whl`. You'll learn more about why this is later in this chapter. For now, continue to the next section where you'll install and profile the C extension version of your `harmonic_mean` function.

4.2.3 Installing and profiling your C extension

You've gone to all the work of building your package several times now, but you have yet to install it. In chapter 2, you created a virtual environment in the `.venv/` directory, located in the root of your package directory. You can use this environment to test the installation of your package. Use the `pip` module to install it using the following command from the root directory of your package. The `.` indicates that pip should install the current directory as a package.

```
$ py -m pip install .   ❶
```

❶ Install the current directory as a package

After the command completes, your `first-python-package` package will be installed just as if it had been installed from PyPI! You should still be able to import your `harmonic_mean` module and use the `harmonic_mean` function, but this time it will resolve to the installed package instead of directly from the source code. Try it in the Python interpreter, as shown in the following snippet:

```
$ py
...
>>> from harmonic_mean import harmonic_mean
>>> harmonic_mean([0.65, 0.7])
0.674074074074074
```

Because your pure-Python version was importable as `harmonic_mean.harmonic_mean` but the C extension is imported from `imppkg.harmonic_mean.harmonic_mean`, you need to update the setup step to profile this new implementation.

SIDEBAR **Exercise**

Run the command to measure the performance of the C extension version of your harmonic mean function. As before, your setup step should do the following:

- **Import the `harmonic_mean` function**
- **Import `random.randint`**
- **Create a list of one million random numbers with values between 1 and 1,000 using the built-in `random.randint` function**

The code you measure should only be the call to your `harmonic_mean` function with the list of random numbers as input.

What do you see? Compare the results to your earlier measurement of the pure-Python implementation. Remember that you haven't changed the code you had to write—you only changed the file name and told Cython how to handle it. On my system, this change alone resulted in the following statistics:

```
20 loops, best of 5: 18.5 msec per loop
```

You read that correctly—the C extension version of the code can run as quickly as 18.5 milliseconds, nearly *three times* faster than the pure-Python implementation. And Cython did all the heavy lifting! Now, what about that binary wheel distribution file?

4.2.4 Build targets for binary wheel distributions

Although the performance you gained using Cython could be considered an easy win, it isn't without its tradeoffs. When you write packages using Python alone, they're extremely portable—any system running a compatible Python version can run your code. As soon as you introduce code that must be compiled, everything changes.

Some of the most performant programming languages are able to achieve their speed through static typing, pre-defined memory allocation, and a compilation step prior to runtime. These features are valuable in computation-heavy contexts. Unfortunately, many of these same features also rely on knowledge of the computer architecture and operating system on which they're running. Performance is often gained by exploiting features and behavior of these systems, so what works in one place won't necessarily work in another. In the worst cases, it can actually cause memory corruption and failed execution if run in the wrong context.

Because of the nuances of execution, source code for these languages must typically be compiled separately for each of the architectures and operating systems where it will be used. Take another look at the binary wheel distribution file in the `dist/` directory. Its file name is divided into several important sections (figure 4.2). The first two are the normalized package name and version. These may be followed by an optional build number. The last three parts are tags that identify the compatibility of the binary wheel:

- Python version—which implementation of Python the code must execute on
- Application binary interface (ABI)—how the binary of the compiled code is organized
- Platform—which operating system and CPU architecture the code must execute on

When you install packages, your package manager will determine which binary wheel distributions are available and use these tags to identify which of those it should download for your system. As an example, the binary wheel distribution file `first_python_package-0.0.1-cp310-cp310-macosx_11_0_x86_64.whl` is compatible with the CPython 3.10 implementation, the CPython 3.10 API, and the macOS 11 operating system running on an x86 64-bit CPU architecture.

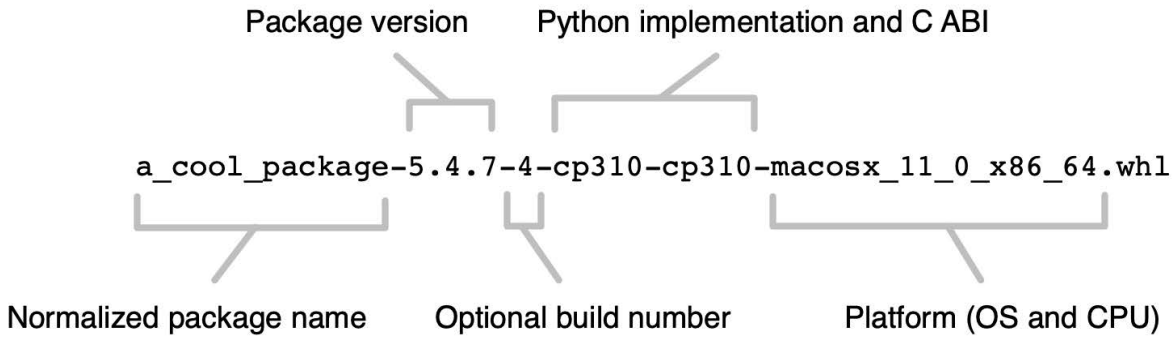

Figure 4.2 The anatomy of a binary wheel distribution file name

You might notice that three of these file name segments relate to the number of binary wheel distributions you would need to build to cover all possible targets. Fortunately, the first two segments—the Python implementation and ABI version—are usually identical. On the other hand, some operating systems can run on different CPU architectures, so that single tag is really two pieces of information. This all boils down to the Python implementations, operating systems, and CPU architectures you want to support. That means the number of binary wheel distributions you need to build is roughly the following:

$$N_{\text{Python implementations}} \cdot N_{\text{Operating systems}} \cdot N_{\text{CPU architectures}} N_{\text{Python implementations}} \cdot N_{\text{Operating systems}} \cdot N_{\text{CPU architectures}}$$

As an example, at the time of this writing the numpy project (numpy.org/) supports CPython 3.7, 3.8, and 3.9 as well as PyPy 3.7. It supports these across different architectures for each of macOS, Linux, and Windows. In all, each release of numpy makes *twenty seven* wheels

available. This sounds like a lot of work, and as a lone maintainer that's probably true. But because numpy is so central to the scientific community for building performant numerical software, the project maintainers are willing to put in this effort on an infrequent basis to deliver the performance their users need all the time.

So far, you've built one pure-Python wheel and one wheel specific to your Python implementation and platform. This might seem a far cry from the scale that projects like numpy are operating at. Fortunately, some tools have emerged that can ease the burden of building these wheels, and you'll learn more about them in chapter 7. For the moment, rejoice that you've built a working Python package and get ready to handle that second request from CarCorp.

4.3 Offering command-line tools from a Python package

CarCorp wants to be able to run a standalone command to quickly calculate harmonic means. They're familiar with using a shell to run commands, but not as familiar with using Python to write and run scripts. Fortunately, most Python build systems also support this. You can tell these systems that, as part of the installation process, some part of the code should be exposed as a runnable command. In the following sections, you'll learn how Setuptools handles this use case.

4.3.1 Creating commands with Setuptools entry points

Setuptools enables you to provide commands to users through what it calls *entry points*. An entry point is like a door—a way in and out of a place. Setuptools entry points provide a way into a package's functionality in a discoverable way. Named commands are one such way of exposing those entry points.

You're likely familiar with the `if __name__ == "__main__":` syntax used in many a Python script intended for command line use. When you run `python some.py`, the `__name__` for `some.py` will be `"__main__"` and the code in the conditional will run. Commands are a more generalized and flexible version of this concept. At a high level, you create a command in Setuptools by mapping a name for the command to the dotted module path of a function. As an example, imagine you want a command named `harmony` that supplied the calculation behavior from the `imppkg.harmonic_mean.harmonic_mean` function. Instead of requiring you to run `python harmonic_mean.py` and respond with `if __name__ == "__main__"` in your code, an entry point enables you to run the `harmony` command and point to a function that calls `imppkg.harmonic_mean.harmonic_mean` with the arguments from the command (table 4.1).

Table 4.1 The different ways to execute code from a Python module

Approach	Command	Requires installation	Pros	Cons
Execute module directly	`$ py /path/to/package/src/imppkg/harmony.py [args]`	No		Imports within code may not work
Execute as importable module	`$ py -m imppkg.harmony [args]`	No (works for any importable code)	Imports within code work	Command is still long, requires knowledge of package structure
Execute as entry point	`$ harmony [args]`	Yes	Short command, no knowledge of package structure required	

To create a command entry point, you first need to create the handler function.

SIDEBAR **Exercise**

Add a new Python module in your `src/imppkg/` directory called `harmony.py`. Inside that module, create a `main` function that:

- **Uses `sys.argv` to get the arguments from the command line**
- **Converts the arguments to a list of floating point numbers**
- **Calls `imppkg.harmonic_mean.harmonic_mean`, passing in the list of numbers**
- **Prints the calculated mean of the numbers**

Remember to import `sys` and `imppkg.harmonic_mean.harmonic_mean`.

With your handling function in place, you can now configure Setuptools to make it available as a command. You tell Setuptools where to look for the command using the `[options.entry_points]` section in the `setup.cfg` file. This section is a table mapping entry point group to (command name, handler function) pairs. For commands, the entry point group is `console_scripts`.

SIDEBAR **What other entry point types are there?**

The entry points system in Setuptools is quite flexible. The `console_scripts` group is the convention used for creating command-line tools, but a group can be any other valid string. This can be used to coordinate functionality across packages if they agree on a convention for an entry point, making plugin-based architectures (en.wikipedia.org/wiki/Plug-in_%28computing%29) possible. Different packages can find each other's software without knowing specifics about it ahead of time, which is a powerful offering for extensibility (Hillard, Dane. "Extensibility and Flexibility." Practices of the Python Pro, Manning Publications, 2020, pp. 147–142.). If you want to build a package that others can extend without you needing to be involved, this is an area to look further into on your own.

Write the entry points section now. It should look like the following snippet:

```
...

[options.entry_points]                  ❶
console_scripts =                       ❷
    harmony = imppkg.harmony:main       ❸
```

❶ Where Setuptools looks for entry points

❷ The group for creating runnable commands

❸ A map of command names to handler functions

Now that you have a handler function and Setuptools knows to make it available through the `harmony` command, it's time to verify that it works. Re-install your package into your virtual environment now. Once that completes, run the following command from the root directory of your package:

```
$ ./venv/bin/harmony 0.65 0.7
```

You should see the following output:

```
0.674074074074074
```

TIP **Notice that you had to include the `.venv/bin/` prefix for the command. When a use installs your package into their active virtual environment or a base Python version, the installed command will automatically be added to their path, and the prefix won't be necessary.**

You now have a buildable, installable Python package that delivers a fast calculation for

harmonic means. Confident you've satisfied the functionality CarCorp asked for, you decide you want to wow them with something they didn't know they needed. Because working in a console often means finding lines of interest in a wall of text, you want the `harmony` output to really pop. You decide you want to add colorized text, but you don't have the time to learn ANSI escape sequences (en.wikipedia.org/wiki/ANSI_escape_code). You decide to install another package to handle this for you.

4.4 Specifying third-party dependencies for Python packages

Up to now, your package has not depended on any third-party Python packages. Now that you want to add one, it might be tempting to install it directly into your virtual environment using pip. Unfortunately, this won't work for your users because they would also need to install the package themselves. Remember from chapter 1 that a major part of the value of package management systems is dependency resolution and installation. What you really want to do is specify to the Python package management tools that your package has a dependency, and let them manage installation for you. This will help you get the dependency, but it will *also* help your users get it. A win all around!

Specifying dependencies for packages is similar to the familiar route of using something like a `requirements.txt` file to list your dependencies, but with two key differences:

- Dependencies need to be specified where your build system will see them, so that the dependencies can be incorporated into the metadata package managers use to resolve and install dependencies.
- Dependencies should be specified as loosely as they can be, to maximize compatibility for users.

IMPORTANT **To emphasize that last point: packages should avoid being pinned to an exact package version when they don't need to be. Imagine that you and I each create a package and they both depend on the `requests` package. Now imagine someone wants to use them both in their project. They install yours, but when they try to install mine they get an error saying that my package depends on `requests==2.1.1` but your package depends on `requests==2.1.2`. There's no way forward through this issue, because solving the problem for one package creates a problem for the other.**

If instead we both make our packages depend on `requests>=2.1.1,<3`, any version of requests greater than 2.1.0 and less than 3.X will work for the user. As the user installs more packages with more dependency specifications, this ensures that we don't unnecessarily narrow the space of valid dependency combinations for them.

Another benefit to using looser dependency definitions for packages is that you can find issues caused by upstream packages sooner. If you pin to an exact version for six months and then try

to upgrade later, you may find a cascade of issues and have to spend a whole day getting back up to speed. If you loosely define dependencies, you'll uncover those issues any time you reinstall dependencies for the package during development and testing. It can feel daunting at first dealing with new issues so frequently, but you'll appreciate iterating on these comparatively smaller changes regularly over having to put out a proverbial fire every few months.

Setuptools looks for package dependencies in the `install_requires` key of the `[options]` section in the `setup.cfg` file. The `install_requires` value is a list of dependencies, specified using the same syntax you would use in a `requirements.txt` file. To add some color to the output of the `harmony` command, you'll use the `termcolor` package. As of this writing, the latest release of termcolor is `1.1.0`. Because you aren't going to test earlier versions, and you trust them to maintain existing features until at least the `2.0.0` release, you can specify `termcolor>=1.1.0,<2` as the version.

Add the `install_requires` key now. It should look like the following snippet:

```
[options]
...
install_requires =
    termcolor>=1.1.0,<2
```

Now, when your package is installed, pip will also download and install the latest 1.X version of `termcolor`. With this in place, you can make use of `termcolor` in your `harmony.py` module. Instead of using the built-in `print` function to print the result of the harmonic mean calculation, import and use the `termcolor.cprint` function. This function accepts additional arguments compared to `print`:

- An optional foreground color specifier, like `'red'` or `'green'`
- An optional background color specifier, like `'on_cyan'` or `'on_red'`
- An `attrs` list for styles, like `['bold', 'italic']`

SIDEBAR **Exercise**

Replace the `print` call in the `harmony.py` module with a call to `termcolor.cprint`. The text should be bold and red on a cyan background. Re-install your package and re-run the `harmony` command to confirm the output looks as you expect.

Looking spectacular? If not, play around with the values for `termcolor` and find a color scheme that you like.

You now have something fairly polished that you can think about shipping to CarCorp. But you have a nagging feeling they'll be asking for more functionality soon. When you're ready, continue on to chapter 5 to see how you can integrate your test suite to verify your changes as your package grows.

4.5 Summary

- Explore non-Python extensions by first using a high level translation layer like Cython.
- Providing a non-Python extension gains runtime performance but adds build time complexity, either for you or your consumer.
- Entry points into your package offer more ways of interacting with its behavior than just importing the code.
- Leverage the power of package management systems to handle dependency resolution for you.

Appendix A: Installing asdf and python-launcher

In this book you'll need to manage multiple Python versions and be able to switch between them. This appendix covers the installation of those tools, which you'll learn more about in the chapters.

A.1 Installing asdf

asdf (github.com/asdf-vm/asdf) is a tool for installing multiple versions of languages, frameworks, and tools and switching between them. You'll use asdf to install multiple base versions of Python, from which you'll create isolated environments for your projects. You can use asdf on macOS, Linux, or the Windows subsystem for Linux.

> **TIP** **If you use other languages, asdf can help there too. It works in a fairly homogeneous way across NodeJS, Ruby and more!**

To install asdf, first determin the most recent release by visiting the releases page (github.com/asdf-vm/asdf/releases). Then, clone the branch corresponding to that version into the `$HOME/.asdf/` directory. As an example, if the latest release is `v1.2.3`, you would run the following command:

```
$ git clone \
    https://github.com/asdf-vm/asdf.git \   ❶
    $HOME/.asdf \                           ❷
    --branch v1.2.3                         ❸
```

❶ The asdf repository on GitHub

❷ The destination for the cloned code

❸ The version of the code to use

After cloning the code, you need to source it during your shell's startup. For macOS, where the default shell is zsh, add the following lines to `$HOME/.zshrc`. For bash, add them to

`$HOME/.bash_profile`.

```
if [ -f $HOME/.asdf/asdf.sh ]; then
    source $HOME/.asdf/asdf.sh
fi
```

After saving your startup file, open a fresh shell session. Verify that asdf was installed properly using the following command:

```
$ asdf --version
```

You should see a version that matches the branch you checked out when you cloned the repository. Now that you have asdf installed, install the Python plugin using the following command:

```
$ asdf plugin add python
```

This will make the plugin available immediately. Verify the plugin is working and see which versions of Python are available using the following command:

```
$ asdf list all python
```

You should see several hundred versions listed, including PyPy, Anaconda, and others. Scroll up to the versions that are only numbered without any name—these are the standard CPython implementation versions.

You'll test your project on the three most recent minor versions of Python, so install these next. As an example, if the most recent version of Python is `3.11.X`, you should install the most recent versions that match the following:

- `3.11.X`
- `3.10.Y`
- `3.9.Z`

You can install these with asdf using the following command, replacing the version with the one you'd like to install:

```
$ asdf install python 3.11.X 3.10.Y 3.9.Z
```

SIDEBAR **WARNING**

macOS Big Sur may cause issues installing older versions of Python. If you see compilation errors trying to install a Python version, you can patch Python with the following approach:

```
ASDF_PYTHON_PATCH_URL=\
"https://github.com/python/cpython/commit/8ea6353.patch?full_index=1" \  (2)
asdf install python 3.11.X                                               (1)
```

(1) **Instruct asdf to apply a patch to the Python code before compiling**

(2) **This specific patch fixes a common compilation issue on macOS Big Sur**

You can always check the documentation for the Python plugin (github.com/danhper/asdf-python) if you run into additional issues.

Finally, make all the Python versions you've installed available on your `$PATH` using the following command, replacing the versions as appropriate:

```
$ asdf global 3.11.X 3.10.X 3.9.X
```

This will create a `$HOME/.tool-versions` file with content similar to the following:

```
python 3.11.X 3.10.X 3.9.X
```

To verify your configuration, start a fresh shell session and invoke the `python` command. This should start an interpreter of the first version of Python you passed to `asdf global python`. You should also be able to start an interpreter from any of the installed versions. As an example, if you have Python 3.9 installed you should be able to invoke the `python3.9` command to get a Python 3.9 interpreter.

A.2 Installing python-launcher

It's not too hard to use different versions of Python with asdf alone, but it will become more difficult as you create environments for different projects. python-launcher (github.com/brettcannon/python-launcher) is a convenience tool for launching the right Python at the right time.

To install python-launcher, you'll first need to install Rust (www.rust-lang.org). As of this writing, the recommended way to install Rust is with the following command (for the latest installation instructions, see "Install Rust", www.rust-lang.org/tools/install):

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Once Rust is installed, use Rust's `cargo` tool to install python-launcher with the following

command:

```
$ cargo install python-launcher
```

To verify your configuration, start a fresh shell session and invoke the `py` command. This should start an interpreter of the *highest* version of Python you installed. You should also be able to start an interpreter from any of the installed versions. As an example, if you have Python 3.9 installed you should be able to invoke the `py` command with the `-3.9` flag to get a Python 3.9 interpreter.

Appendix B: Installing pipx, build, and tox

In this book you'll be using a few tools frequently, and you'll eventually need them across several projects. This appendix covers the installation of these tools, which you'll learn more about in the chapters.

B.1 Installing pipx

Several tools are available as Python packages, but you don't want to have to install them in each project where you use them. You should install the tools in isolation, though, so that they don't mix with other installed software unnecessarily. pipx (github.com/pypa/pipx/) is a tool for running Python applications in isolated environments.

You'll install pipx using the pipx-in-pipx (github.com/mattsb42-meta/pipx-in-pipx) project so that even pipx itself is isolated and managed by itself. Install pipx-in-pipx using the following command:

```
$ py -m pip install pipx-in-pipx
```

B.2 Installing build

build (github.com/pypa/build) is a tool provided by the Python Packaging Authority (PyPA) for building Python packages. Because you might want to use it to build several different packages eventually, installing it with pipx will make it available wherever you might need it.

Install build using the following command:

```
$ pipx install build
```

You should see output similar to the following, indicating the `pyproject-build` application was installed:

```
  installed package build 0.4.0, Python 3.10.0
  These apps are now globally available
```

```
    - pyproject-build
done!
```

To verify your configuration, run the following command:

```
$ pyproject-build --version
```

The version should match the version in the output of the `pipx install` command.

B.3 Installing tox

tox is a testing and task management tool for Python projects. Install tox using the following command:

```
$ pipx install tox
```

You should see output similar to the following:

```
  installed package tox 3.23.1, Python 3.10.0
  These apps are now globally available
    - tox
    - tox-quickstart
done!
```

To verify your configuration, run the following command:

```
$ tox --version
```

The version should match the version in the output of the `pipx install` command.