O'REILLY®

Hypermodern Python Tooling

Building Reliable Workflows for an Evolving Python Ecosystem

Early Release
RAW & UNEDITED

Claudio Jolowicz

Hypermodern Python Tooling

Building Reliable Workflows for an Evolving Python Ecosystem

Claudio Jolowicz

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Hypermodern Python Tooling

by Claudio Jolowicz

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Brian Guerin
- Development Editor: Sarah Grey
- Production Editor: Gregory Hyman
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- April 2024: First Edition

Revision History for the Early Release

- 2022-11-01: First Release
- 2023-01-31: Second Release
- 2023-03-20: Third Release
- 2023-06-26: Fourth Release

See *https://oreilly.com/catalog/errata.csp?isbn=9781098139582* for release details.

978-1-098-13958-2

Chapter 1. Installing Python

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mail@claudiojolowicz.com.

If you've picked up this book, you likely have Python installed on your machine already. Most common operating systems ship with a `python` or `python3` command. This can be the interpreter used by the system itself or a shim that installs Python for you when you invoke it for the first time.

Why dedicate an entire chapter to the topic if it's so easy to get Python onto a new machine? The answer is that installing Python for long-term development can be a complex matter, and there are several reasons for this:

- You generally need multiple versions of Python installed side-by-side. (If you're wondering why, we'll get to that shortly.)
- There are a few different ways to install Python across the common platforms, each with unique advantages, tradeoffs, and sometimes pitfalls.
- Python is a moving target: You need to keep existing installations up-to-date with the latest maintenance release, add installations when a new feature version is published, and remove versions that are no longer supported. You may even need to test a prerelease of the next Python.
- You may want your code to run on multiple platforms. While Python makes it easy to write portable programs, setting up a developer environment requires some familiarity with the idiosyncrasies of each platform.
- You may want to run your code with an alternative implementation of Python.[1]

In this first chapter, I'll show you how to install multiple Python versions on some of the major operating systems in a sustainable way, and how to keep your little snake farm in good shape.

TIP

If you only develop for a single platform, feel free to skip ahead to your preferred platform. I'd encourage you to learn about working with Python on other operating systems though. It's fun—and familiarity with other platforms enables you to provide a better experience to the contributors and users of your software.

Supporting Multiple Versions of Python

Python programs often target several versions of the language and standard library at once. This may come as a surprise. Why would you run your code with anything but the latest Python? After all, this lets your programs benefit from new language features and library improvements immediately.

As it turns out, runtime environments often come with a variety of older versions of Python.[2] Even if you have tight control over your deployment environments, you may want to get into the habit of testing against multiple versions. The day the trusty Python in your production environment features in a security advisory better not be the day you start porting your code to newer releases.

For these reasons, it is common to support both current and past versions of Python until their official end-of-life date, and to set up installations for them side-by-side on a developer machine. With new feature versions coming out every year and support extending over five years, this gives you a testing matrix of five actively supported versions (see Figure 1-1). If that sounds like a lot of work, don't worry: the Python ecosystem comes with tooling that makes this a breeze.

THE PYTHON RELEASE CYCLE

Python has an annual release cycle: feature releases happen every October. Each feature release gets a new minor version in Python's `major.minor.micro` scheme. By contrast, new major versions are rare and reserved for strongly incompatible changes—at the time of writing this book, a Python 4 is not in sight. Python's backward compatibility policy allows incompatible changes in minor releases when preceded by a two-year deprecation period.

Feature versions are maintained for five years, after which they reach end-of-life. Bugfix releases for a feature version occur roughly every other month during the first 18 months after its initial release. This is followed by security updates whenever necessary during the remainder of the five-year support period. Each maintenance release bumps the micro version.

Prereleases for upcoming Python feature releases happen throughout the year before their publication. These prereleases fall into three consecutive phases: alphas, betas, and release candidates. You can recognize them by the suffix that gets appended to the upcoming Python version, indicating the release status and sequence number, such as `a1`, `b3`, `rc2`.

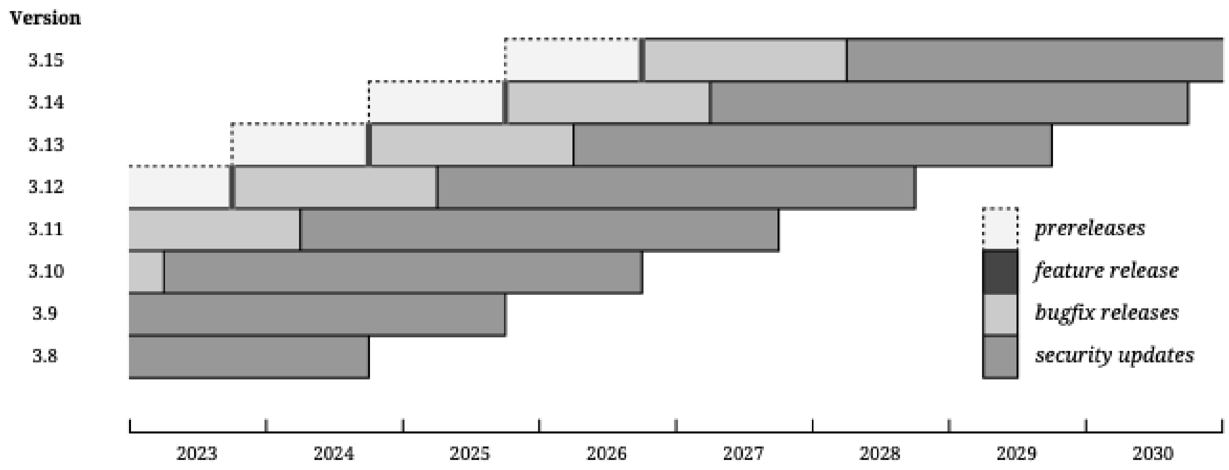

Figure 1-1. Timeline of Python Releases

Locating Python Interpreters

How do you select the correct Python interpreter if you have multiple ones on your system? Let's look at a concrete example. When you type `python` at the command line, the shell searches the directories in the `PATH` environment variable from left to right and invokes the first executable file named `python`. Most Python installations also provide versioned commands named `python3.x`, letting you disambiguate between different feature versions.

NOTE

On Windows, `PATH`-based interpreter discovery is far less relevant because Python installations can be located via the Windows Registry (see "The Python Launcher for Windows").

A common default for the `PATH` variable is `/usr/local/bin:/usr/bin:/bin` on Unix-like systems. You can modify the variable using the `export` builtin of many shells. Here's how you would add a Python installation in */usr/local/opt/python* using the Bash shell:

```
export PATH="/usr/local/opt/python/bin:$PATH"
```

Note that you're adding the *bin* subdirectory instead of the installation root, because that's where the interpreter is normally located on these systems. We'll take a closer look at the layout of Python installations in Chapter 2.

The line above also works with the Zsh shell, which is the default on macOS. That said, there's a more idiomatic way to manipulate the search path on Zsh:

```
typeset -U path ❶
path=(/usr/local/opt/python/bin $path) ❷
```

❶ This instructs the shell to remove duplicate entries from the search path.

❷ The shell keeps the `path` array synchronized with the `PATH` variable.

The Fish shell offers a function to uniquely and persistently prepend an entry to the search path:

```
fish_add_path /usr/local/opt/python/bin
```

It would be tedious to set up the search path manually at the start of every shell session. Instead, you can place the commands above in your *shell profile*—this is a file in your home directory that is read by the shell on startup. Table 1-1 shows the most common ones:

Table 1-1. The startup files of some common shells

Shell	**Startup file**
Bash	*.bash_profile* or *.profile* (Debian and Ubuntu)
Zsh	*.zshrc*
Fish	*.config/fish/fish.config*

The order of directories on the search path matters because earlier entries take precedence over, or “shadow”, later ones. You’ll often add Python versions against a backdrop of existing installations, such as the interpreter used by the operating system, and you want the shell to choose your installations over those present elsewhere.

TIP

Unless your system already comes with a well-curated and up-to-date selection of interpreters, prepend Python installations to the `PATH` environment variable, with the latest stable version at the very front.

Figure 1-2 shows a macOS machine with several Python installations. Starting from the bottom, the first interpreter is located in */usr/bin* and part of Apple’s Command Line Tools (Python 3.8). Next up, in */usr/local/bin*, are several interpreters from the Homebrew distribution; the `python3` command here is its main interpreter (Python 3.10). The Homebrew interpreters are followed by a prerelease from *python.org* (Python 3.12). The top entries contain the current release (Python 3.11), also from Homebrew.

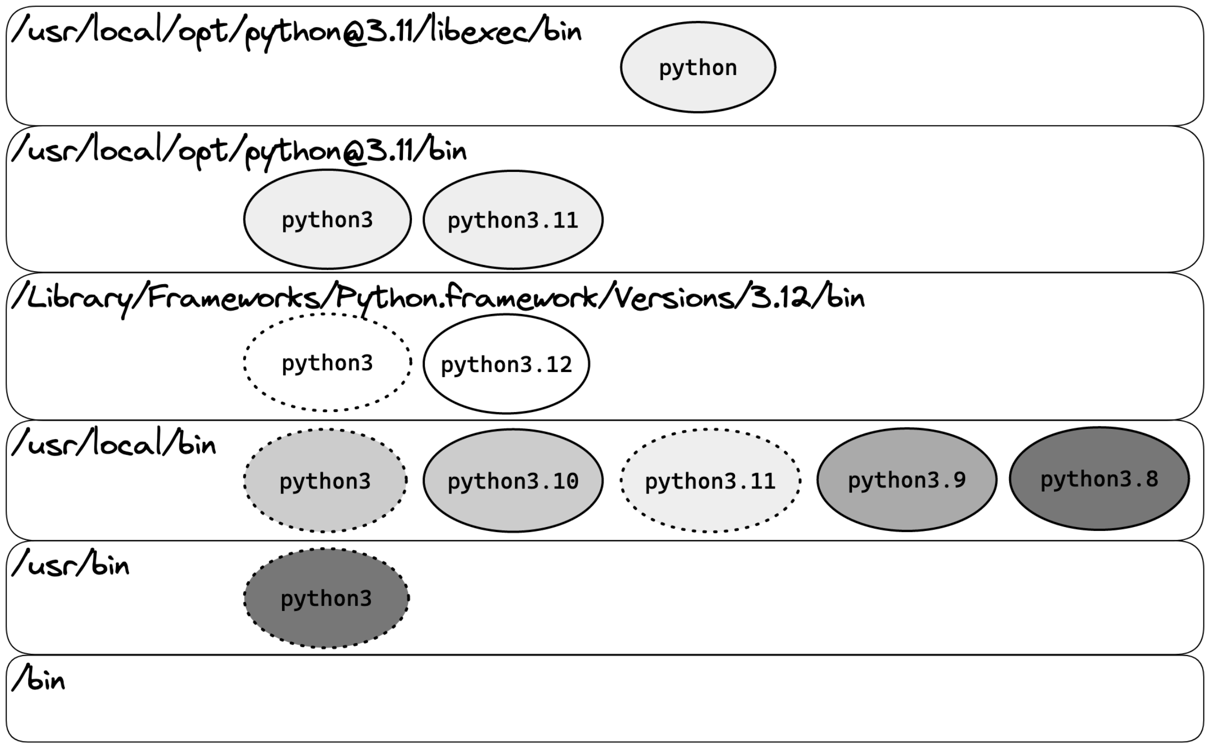
/usr/local/opt/python@3.11/libexec/bin
python
/usr/local/opt/python@3.11/bin
python3
python3.11
/Library/Frameworks/Python.framework/Versions/3.12/bin
python3
python3.12
/usr/local/bin
python3
python3.10
python3.11
python3.9
python3.8
/usr/bin
python3
/bin

Figure 1-2. A developer system with multiple Python installations. The search path is displayed as a stack of directories; commands at the top shadow those further down.

A SHORT HISTORY OF PATH

Curiously, the `PATH` mechanism has remained essentially the same since the 1970s. In the original design of the Unix operating system, the shell still looked up commands entered by the user in a directory named */bin*. With the 3rd edition of Unix (1973), this directory—or rather, the 256K drive that backed it—became too small to hold all available programs. Researchers at Bell Labs introduced an additional filesystem hierarchy rooted at */usr*, allowing a second disk to be mounted. But now the shell had to search for programs across multiple directories—*/bin* and */usr/bin*. Eventually, the Unix designers settled on storing the list of directories in an environment variable named `PATH`. Since every process inherits its own copy of the environment, users can customize their search path without affecting system processes.

Installing Python on Windows

The core Python team provides official binary installers in the Downloads for Windows section of the Python website. Locate the latest release of each Python version you wish to support, and download the 64-bit Windows installer for each.

NOTE

Depending on your domain and target environment, you may prefer to use the Windows Subsystem for Linux (WSL) for Python development. In this case, please refer to the section "Installing Python on Linux" instead.

In general, there should be little need to customize the installation—with one exception: When installing the latest stable release (and only then), enable the option to add Python to your `PATH` environment variable on the first page of the installer dialog. This ensures that your default `python` command uses a well-known and up-to-date Python.

The *python.org* installers are an efficient way to set up multi-version Python environments on Windows, for several reasons:

- They register each Python installation with the Windows Registry, making it easy for developer tools to discover interpreters on the system (see "The Python Launcher for Windows".)
- They don't have some disadvantages of redistributed versions of Python, such as lagging behind the official release, or being subject to downstream modifications.
- They don't require you to build the Python interpreter, which—apart from taking precious time—involves setting up Python's build dependencies on your system.

Binary installers are only provided up to the last bugfix release of each Python version, which occurs around 18 months after the initial release. Security updates for older versions, on the other hand, are provided as source distributions only. For these, you'll need to build Python from source[3] or use an alternative installer such as Conda (see "Installing Python from Anaconda").

Keeping Python installations up-to-date falls on your shoulders when you're using the binary installers from *python.org*. New releases are announced in many places, including the Python blog and the Python Discourse. When you install a bugfix release for a Python version that is already present on the system, it will replace the existing installation. This preserves virtual environments and developer tools on the upgraded Python version and should be a seamless experience. When you install a new feature release of Python, there are some additional steps to be mindful of:

- Enable the option to add the new Python to the `PATH` environment variable.
- Remove the previous Python release from `PATH`. Instead of modifying the environment variable manually, you can re-run its installer in maintenance mode, using the *Apps and Features* tool that is part of Windows. Locate the entry for the previous Python in the list of installed software, and choose the *Modify* action from the context menu.

- You may also wish to reinstall some of your developer tooling, to ensure that it runs on the latest Python version.

Eventually, a Python version will reach its end of life, and you may wish to uninstall it to free up resources. You can remove an existing installation using the *Apps and Features* tool. Choose the *Uninstall* action for its entry in the list of installed software. Beware that removing a Python version will break virtual environments and developer tools that are still using it, so you should upgrade those beforehand.

MICROSOFT STORE PYTHON

Windows systems ship with a `python` stub that redirects the user to the latest Python package on the Microsoft Store. The Microsoft Store package is intended mainly for educational purposes, and does not have full write access to some shared locations on the filesystem and the registry. While it's useful for teaching Python to beginners, I would not recommend it for most intermediate and advanced Python development.

The Python Launcher for Windows

Python development on Windows is special in that tooling can locate Python installations via the Windows Registry. The Python Launcher for Windows leverages this to provide a single entry point to interpreters on the system. It is a utility included with every *python.org* release and associated with Python file extensions, allowing you to launch scripts from the Windows File Explorer.

Running applications with a double-click is handy, but the Python Launcher is at its most powerful when you invoke it from a command-line prompt. Open a Powershell window and run the `py` command to start an interactive session:

```
> py
Python 3.10.5 (tags/v3.10.6:f377153, Jun  6 2022,
Type "help", "copyright", "credits" or "license"
>>>
```

By default, the Python Launcher selects the most recent version of Python installed on the system. It's worth noting that this may not be the same as the *most recently installed* version on the system. This is good—you don't want your default Python to change when you install a bugfix release for an older version.

If you want to launch a specific version of the interpreter, you can pass the feature version as a command-line option:

```
> py -3.9
Python 3.9.13 (tags/v3.9.13:6de2ca5, May 17 2022,
Type "help", "copyright", "credits" or "license"
>>>
```

Any remaining arguments to `py` are forwarded to the selected interpreter. Let's see how you would display the versions of two interpreters on the system:

```
> py -V
Python 3.10.5

> py -3.9 -V
Python 3.9.13
```

Using the same mechanism, you can run a script on a specific interpreter:

```
> py -3.9 path\to\script.py
```

If you don't specify a version, `py` inspects the first line of the script to see if a version is specified there, in the form `#!python3.x`. Both the line itself and the version inside are optional; when omitted, `py` defaults to the latest interpreter. You can also include a leading path as in `#!/usr/bin/python` (a standard location on Unix systems) and `py` will conveniently ignore this part; this lets you install the same script on Windows and Unix-like systems.

NOTE

You may have recognized the `#!` line as what's known as a *shebang* on Unix-like operating systems. On these systems, the program loader uses it to locate and launch the interpreter for the script.

If the script is installed as a module, you can also pass its import name to the `-m` interpreter option:

```
> py -m module
```

Perhaps the most important example of this technique is installing a third-party package with pip, the Python package installer:

```
> py -m pip install package
```

Pip installs packages into its own environment, so invoking it via the Python Launcher lets you control where a package is installed. The explicit form is almost always what you want, so you should prefer it over the shorter `pip install` as a matter of routine.[4]

As you have seen, the Python Launcher defaults to the newest version on the system. There is an exception to this rule if a virtual environment is active. In this case, `py` defaults to the interpreter in the virtual environment. (You can think of virtual environments as lightweight satellites of a full Python installation; I'll talk more about them in Chapter 2.) In the following example session using Powershell, you create and activate a virtual environment using an older interpreter version, then switch back to the global environment:

```
> py -V
Python 3.10.5
> py -3.9 -m venv venv-3.9
> venv-3.9\Scripts\activate
(venv-3.9) > py -V
Python 3.9.13
(venv-3.9) > deactivate
> py -V
Python 3.10.5
```

WARNING

Do not pass the version to `py` when you have a virtual environment activated. This would cause `py` to select the global Python installation, even if the version matches the interpreter inside the active environment.

The Python Launcher defaults to the latest Python version on the system even if that happens to be a prerelease. You can override this default persistently by setting the `PY_PYTHON` and `PY_PYTHON3` environment variables to the current stable release:

```
> setx PY_PYTHON 3.x
> setx PY_PYTHON3 3.x
```

Restart the console for the setting to take effect. Don't forget to remove these variables once you upgrade to the final release.

To conclude our short tour of the Python Launcher, use the command `py --list` to enumerate the interpreters on your system:

```
> py --list
 -V:3.11          Python 3.11 (64-bit)
 -V:3.10 *        Python 3.10 (64-bit)
 -V:3.9           Python 3.9 (64-bit)
```

In this listing, the asterisk marks the default version of Python.

Installing Python on macOS

You can install Python on macOS in several ways. In this section, I'll take a look at the Homebrew package manager and the official *python.org* installers. Both provide multi-version binary distributions of Python. Some installation methods that are common on Linux—such as Pyenv—also work on macOS. The Conda package manager even supports Windows, macOS, and Linux. I'll talk about them in later sections.

PYTHON FROM APPLE'S COMMAND LINE TOOLS

macOS ships with a `python3` stub that installs Apple's Command Line Tools when you run it for the first time, including an older version of Python. While it's good to know about Python commands on your system, other distributions will serve you better; they allow you to develop with Python versions of your choice.

Homebrew Python

Homebrew is a third-party package manager for macOS and Linux. It provides an *overlay distribution*, an open-source software collection that you install on top of the existing operating system. Installing the package manager is straightforward; refer to the official website for instructions.

Homebrew distributes packages for every maintained feature version of Python. Use the `brew` command-line interface to manage them:

`brew install python@3.X`

Install a new Python version.

`brew upgrade python@3.X`

Upgrade a Python version to a maintenance release.

`brew uninstall python@3.X`

Uninstall a Python version.

You may find that you already have some Python versions installed for other Homebrew packages that depend on them. Nonetheless, it's important that you install every version explicitly. Automatically installed packages may get deleted when you run `brew autoremove` to clean up resources.

Homebrew places a `python3.`*`x`* command for each version on your `PATH`, as well as a `python3` command for its main Python package—which may be either the current or the previous stable release. You should override this to ensure both `python` and `python3` point to the latest version. First, query the package manager for the installation root (which is platform-dependent):

```
$ brew --prefix python@3.10
/opt/homebrew/opt/python@3.10
```

Next, prepend the *bin* and *libexec/bin* directories from this installation to your `PATH`. Here's an example that works on the Bash shell:

```
export PATH="/opt/homebrew/opt/python@3.10/bin:$I
export PATH="/opt/homebrew/opt/python@3.10/libexe
```

Homebrew has some advantages over the official *python.org* installers:

- You can use the command line to install, upgrade, and uninstall Python versions.
- Homebrew includes security releases for older versions—by contrast, *python.org* installers are provided up to the last bugfix release only.
- Homebrew Python is tightly integrated with the rest of the distribution. In particular, the package manager can satisfy Python dependencies like OpenSSL. This gives you the option to upgrade them independently when needed.

On the other hand, Homebrew Python also comes with some limitations and caveats:

- Homebrew Python is used by some other packages in the distribution. Beware that using pip to install or uninstall Python packages system-wide can affect—and, in the worst case, break—those packages.
- Packages generally lag a few days or weeks behind official releases. They also contain some downstream modifications, although these are quite reasonable. For example, Homebrew separates modules related to graphical user interfaces (GUI) from the main Python package.
- Homebrew does not package prereleases of upcoming Python versions.

By default, Homebrew upgrades Python to maintenance releases automatically. This won't break virtual environments and developer tools, though: They reference the interpreter using a *symbolic link*—a special kind of file that points to another file, much like a shortcut in Windows—that remains stable for all releases of the same feature version.

TIP

Personally, I recommend Homebrew for managing Python on macOS—it's well-integrated with the rest of the system and easy to keep up-to-date. Use the *python.org* installers to test your code against prereleases, which are not available from Homebrew.

The python.org Installers

The core Python team provides official binary installers in the Downloads for macOS section of the Python website. Download the 64-bit universal2 installer for the release you wish to install. The universal2 binaries of the interpreter run natively on both Apple Silicon and Intel chips.

For multi-version development, I recommend a custom install—watch out for the *Customize* button in the installer dialog. In the ensuing list of installable components, disable the *UNIX command-line tools* and the *Shell profile updater*. Both options are designed to put the interpreter and some other commands on your `PATH`.[5] Instead, you should edit your shell profile manually. Prepend the directory */Library/Frameworks/Python.framework/Versions/3.x/bin* to `PATH`, replacing `3.x` with the actual feature version. Make sure that the current stable release stays at the front of `PATH`.

NOTE

After installing a Python version, run the *Install Certificates* command located in the */Applications/Python 3.x/* folder. This command installs Mozilla's curated collection of root certificates.

When you install a bugfix release for a Python version that is already present on the system, it will replace the existing installation. Uninstalling a Python version is done by removing these two directories:

- */Library/Frameworks/Python.framework/Versions/3.x/*
- */Applications/Python 3.x/*

FRAMEWORK BUILDS ON MACOS

Most Python installations on a Mac are so-called *framework builds*. Frameworks are a macOS concept for a “versioned bundle of shared resources.” You may have come across bundles before, in the form of apps in the *Applications* folder. A bundle is just a directory with a standard layout, keeping all the files in one place.

Frameworks contain multiple versions side-by-side in directories named *Versions/3.x*. One of these is designated as the current version using a *Versions/Current* symlink. Under each version in a Python Framework, you can find a conventional Python installation layout with `bin` and `lib` directories.

Installing Python on Linux

The core Python team does not provide binary installers for Linux. Generally, the preferred way to install software on Linux distributions is using the official package manager. However, this is not unequivocally true when installing Python for development—here are some important caveats:

- The system Python in a Linux distribution may be quite old, and not every distribution includes alternate Python versions in their main package repositories.
- Linux distributions have mandatory rules about how applications and libraries may be packaged. For example, Debian's Python Policy mandates that the standard `ensurepip` module must be shipped in a separate package; as a result, you can't create virtual environments on a default Debian system (a situation commonly fixed by installing the `python3-full` package.)
- The main Python package in a Linux distribution serves as the foundation for other packages that require a Python interpreter. These packages may include critical parts of the system, such as Fedora's package manager `dnf`. Distributions therefore apply safeguards to protect the integrity of the system; for example, they often limit your ability to use pip outside of a virtual environment.

In the next sections, I'll take a look at installing Python on two major Linux distributions, Fedora and Ubuntu. Afterwards, I'll cover some generic installation methods that don’t use the official package manager. I’ll also introduce you to the Python Launcher for Unix, a third-party package that aims to bring the `py` utility to Linux, macOS, and similar systems.

Fedora Linux

Fedora is an open-source Linux distribution, sponsored primarily by Red Hat, and the upstream source for Red Hat Enterprise Linux (RHEL). It aims to stay close to upstream projects and uses a rapid release cycle to foster innovation. Fedora is renowned for its excellent Python support, with Red Hat employing several Python core developers.

Python comes pre-installed on Fedora, and you can install additional Python versions using `dnf`, its package manager:

`sudo dnf install python3.X`

    Install a new Python version.

`sudo dnf upgrade python3.X`

    Upgrade a Python version to a maintenance release.

`sudo dnf remove python3.X`

    Uninstall a Python version.

Fedora has packages for all active feature versions and prereleases of CPython, the reference implementation of Python, as well as packages with alternative implementations like PyPy. A convenient shorthand to install all of these at once is to install *tox*:

```
$ sudo dnf install tox
```

In case you're wondering, *tox* is a test automation tool that makes it easy to run a test suite against multiple versions of Python (see Chapter 6); its Fedora package pulls in most available interpreters as recommended dependencies.

Ubuntu Linux

Ubuntu is a popular Linux distribution based on Debian and funded by Canonical Ltd. Ubuntu only ships a single version of Python in its main repositories; other versions of Python, including prereleases, are provided by a Personal Package Archive (PPA). A *PPA* is a community-maintained software repository on Launchpad, the software collaboration platform run by Canonical.

Your first step on an Ubuntu system should be to add the *deadsnakes* PPA:

```
$ sudo apt update && sudo apt install software-p
$ sudo add-apt-repository ppa:deadsnakes/ppa && 
```

You can now install Python versions using the `apt` package manager:

`sudo apt install python3.X-full`

Install a new Python version.

`sudo apt upgrade python3.X-full`

Upgrade a Python version to a maintenance release.

`sudo apt remove python3.X-full`

Uninstall a Python version.

TIP

Always remember to include the `-full` suffix when installing Python on Debian and Ubuntu. The `python3.X-full` packages pull in the entire standard library and up-to-date root certificates. In particular, they ensure you're able to create virtual environments.

Other Linux Distributions

What do you do if your Linux distribution does not package multiple versions of Python? The traditional answer is "roll your own Python".

This may seem scary, but we'll see how straightforward building Python has become these days in "Installing Python with Pyenv". However, it turns out that building from source is not your only option. Several cross-platform package managers provide binary packages of Python; in fact, we've already seen one of them.

The Homebrew distribution (see "Homebrew Python") is available on macOS and Linux, and most of what we said above applies to Linux as well. The main difference between both platforms is the installation root: Homebrew on Linux installs packages under */home/linuxbrew/.linuxbrew* by default instead of */opt/homebrew*. Keep this in mind when adding Homebrew's Python installations to your `PATH`.

A popular cross-platform way to install Python is the Anaconda distribution, which is targeted at scientific computing and supports Windows, macOS, and Linux. We'll cover Anaconda in a separate section at the end of this chapter (see "Installing Python from Anaconda").

THE NIX PACKAGE MANAGER

Another fascinating option for both macOS and Linux is Nix, a purely functional package manager with reproducible builds of thousands of software packages. Nix makes it easy and fast to set up isolated environments with arbitrary versions of software packages. Here's how you would set up a development environment with two Python versions:

```
$ nix-shell --packages python310 python39
[nix-shell]$ python -V
3.10.6
[nix-shell]$ python3.9 -V
3.9.13
[nix-shell]$ exit
```

Before dropping you into the environment, Nix transparently downloads pre-built Python binaries from the Nix Packages Collection and adds them to your `PATH`. Each package gets a unique subdirectory on the local filesystem, using a cryptographic hash that captures all its dependencies.

I won't go into the details of installing and using Nix in this book. If you have Docker installed, you can get a taste of what's possible using the Docker image for NixOS, a Linux distribution built entirely using Nix:

```
$ docker run -it nixos/nix
```

The Python Launcher for Unix

The Python Launcher for Unix is a port of the official `py` utility to Linux and macOS, as well as any other operating system supporting the Rust programming language. You can install the `python-launcher` package with a number of package managers, including `brew`, `dnf`, and `cargo`. Generally, it works much like its Windows counterpart (see "The Python Launcher for Windows"):

```
$ py -V
3.10.6

$ py -3.9 -V
3.9.13
$ py --list
 3.10 │ /usr/local/opt/python@3.10/bin/python3.1
 3.9  │ /usr/local/opt/python@3.9/bin/python3.9
 3.8  │ /usr/local/opt/python@3.8/bin/python3.8
 3.7  │ /usr/local/opt/python@3.7/bin/python3.7
```

The Python Launcher for Unix discovers interpreters by scanning the `PATH` environment variable for `pythonX.Y` commands; in other words, invoking `py -X.Y` is equivalent to running `pythonX.Y`. The main benefit of `py` is to provide a cross-platform way to launch Python, with a well-defined default when no version is specified: the newest interpreter on the system or the interpreter in the active virtual environment.

The Python Launcher for Unix also defaults to the interpreter in a virtual environment if the environment is named `.venv` and located in the current directory or one of its parents. Unlike with the Windows Launcher, you don't need to activate the environment for this to work. For example, here's a quick way to get an interactive session with the `rich` console library installed:

```
$ py -m venv .venv
$ py -m pip install rich
$ py
>>> from rich import print
>>> print("[u]Hey, universe![/]")
Hey, universe!
```

If you invoke `py` with a Python script, it inspects the shebang to determine the appropriate Python version. Note that this mechanism bypasses the program loader. For example, the following script may produce different results when run stand-alone versus when invoked with `py`:

```
#!/usr/bin/python3
import sys
print(sys.executable)
```

Entry points are a more sustainable way to create scripts that does not rely on handcrafting shebang lines. We'll cover them in Chapter 3.

Installing Python with Pyenv

Pyenv is a Python version manager for macOS and Linux. It includes a build tool—also available as a stand-alone program named `python-build`—that downloads, builds, and installs Python versions in your home directory. Pyenv allows you to activate and deactivate these installations globally, per project directory, or per shell session.

NOTE

In this section, we'll use Pyenv as a build tool. If you're interested in using Pyenv as a version manager, please refer to the official documentation for additional setup steps.

The best way to install Pyenv on macOS and Linux is using Homebrew:

```
$ brew install pyenv
```

One great benefit of installing Pyenv from Homebrew is that you'll also get the build dependencies of Python. If you use a different installation method, check the Pyenv wiki for platform-specific instructions on how to set up your build environment.

Display the available Python versions using the following command:

```
$ pyenv install --list
```

As you can see, the list is quite impressive: Not only does it cover all active feature versions of Python, it also includes prereleases, unreleased development versions, almost every point release published over the past two decades, and a wealth of alternative implementations, such as GraalPython, IronPython, Jython, MicroPython, PyPy, and Stackless Python.

You can build and install any of these versions by passing them to `pyenv install`:

```
$ pyenv install 3.x.y
```

When using Pyenv as a mere build tool, as we're doing here, you need to add each installation to `PATH` manually. You can find its location using the command `pyenv prefix 3.x.y` and append */bin* to that. Here's an example for the Bash shell:

```
export PATH="$HOME/.pyenv/versions/3.x.y/bin:$PAT
```

Installing a maintenance release with Pyenv does *not* implicitly upgrade existing virtual environments and developer tools on the same feature version, so you'll have to recreate these environments using the new release. When you no longer use an installation, you can remove it like this:

```
$ pyenv uninstall 3.x.y
```

By default, Pyenv does not enable profile-guided optimization (PGO) and link-time optimization (LTO) when building the interpreter. According to the Python Performance Benchmark Suite, these optimizations can lead to a significant speedup for CPU-bound Python programs—between 10% and 20%. You can enable them using the `PYTHON_CONFIGURE_OPTS` environment variable:

```
$ export PYTHON_CONFIGURE_OPTS='--enable-optimiz
```

Unlike most macOS installers, Pyenv defaults to POSIX installation layout instead of the framework builds typical for this platform. If you are on macOS, I advise you to enable framework builds for consistency.[6] You can do so by adding the configuration option `--enable-framework` to the list above.

MANAGING PYTHON VERSIONS WITH PYENV

Version management in Pyenv works by placing small wrapper scripts called *shims* on your `PATH`. These shims intercept invocations of the Python interpreter and other Python-related tools and delegate to the actual commands in the appropriate Python installation. This can be seen as a more powerful method of interpreter discovery than the `PATH` mechanism of the operating system, and it avoids polluting the search path with a plethora of Python installations.

Pyenv's shim-based approach to version management is convenient, but it also comes with a tradeoff in runtime and complexity: Shims add to the startup time of the Python interpreter. They also put deactivated commands on `PATH`, which interferes with other tools that perform interpreter discovery, such as `py`, `virtualenv`, `tox`, and Nox. When installing packages with entry point scripts, you need to run `pyenv rehash` for the scripts to become visible.

If the practical advantages of the shim mechanism convince you, you may also like asdf, a generic version manager for multiple language runtimes; its Python plugin uses `python-build` internally. If you like per-directory version management, but don't like shims, take a look at direnv, which is able to update your `PATH` whenever you enter a directory. (It can even create and activate virtual environments for you.)

Installing Python from Anaconda

Anaconda is an open-source software distribution for scientific computing, maintained by Anaconda Inc. Its centerpiece is Conda, a cross-platform package manager for Windows, macOS, and Linux. Conda packages can contain software written in any language, such as C, C++, Python, R, or Fortran.

In this section, you'll use Conda to install Python. Conda does not install software packages globally on your system. Each Python installation is contained in a Conda environment and isolated from the rest of your system. A typical Conda environment is centered around the dependencies of a particular project—say, a set of libraries for machine learning or data science—of which Python is only one among many.

Before you can create Conda environments, you'll need to bootstrap a base environment containing Conda itself. There are a few ways to go about this: You can install the full Anaconda distribution, or you can use the Miniconda installer with just Conda and a few core packages. Both Anaconda and Miniconda download packages from the *defaults* channel, which may require a commercial license for enterprise use.

Miniforge is a third alternative—it is similar to Miniconda but installs packages from the community-maintained *conda-forge* channel. You can get Miniforge using its official installers from GitHub, or you can install it from Homebrew on macOS and Linux:

```
$ brew install miniforge
```

Conda requires shell integration to update the search path and shell prompt when you activate or deactivate an environment. If you've installed Miniforge from Homebrew, update your shell profile using the `conda init` command with the name of your shell. For example:

```
$ conda init bash
```

By default, the shell initialization code activates the base environment automatically in every session. You may want to disable this behavior if you also use Python installations that are not managed by Conda:

```
$ conda config --set auto_activate_base false
```

The Windows installer does not activate the base environment globally. Interact with Conda using the Miniforge Prompt from the Windows Start Menu.

Congratulations, you now have a working Conda installation on your system! Let's use Conda to create an environment with a specific version of Python:

```
$ conda create --name=name python=3.x
```

Before you can use this Python installation, you need to activate the environment:

```
$ conda activate name
```

Upgrading Python to a newer release is simple:

```
$ conda update python
```

This command will run in the active Conda environment. What's great about Conda is that it won't upgrade Python to a release that's not yet supported by the Python libraries in the environment.

When you're done working in the environment, deactivate it like this:

```
$ conda deactivate
```

In this chapter, we've only looked at Conda as a way to install Python. In Chapter 2, you'll see how to use Conda to manage the dependencies of a specific project.

Summary

In this chapter, you've learned how to install Python on Windows, macOS, and Linux. The flow chart in Figure 1-3 should provide some guidance on selecting the installation method that works best for you. You've also learned how to use the Python Launcher to select interpreters installed on your system. While the Python Launcher helps remove ambiguity when selecting interpreters, you should still audit your search path to ensure you have well-defined `python` and `python3` commands.

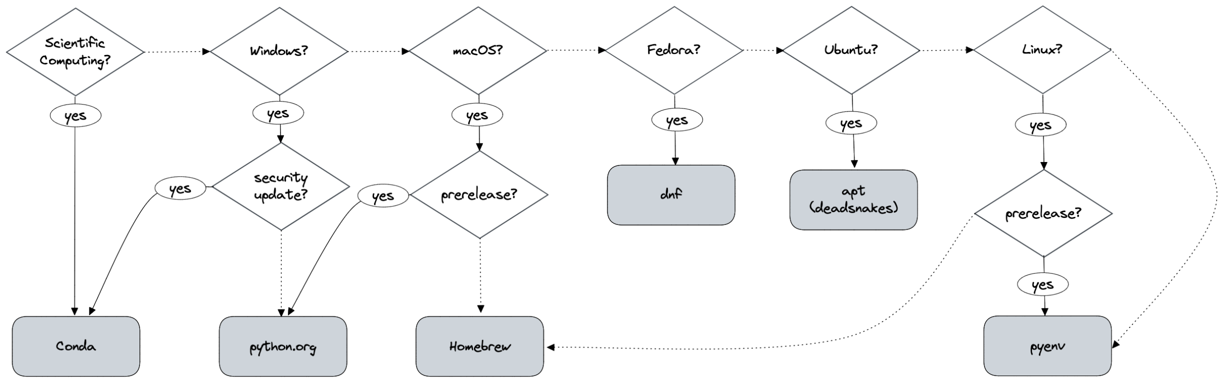

Figure 1-3. Choosing an installation method for Python

While CPython is the reference implementation of Python, there are quite a few more to choose from, from ports to other platforms (WebAssembly, Java, .NET, MicroPython) to performance-oriented forks and reimplementations such as PyPy, Pyjion, Pyston, and Cinder.

Let's take an example: At this time of writing, the long-term support (LTS) release of Debian Linux ships patched versions of Python 2.7.13 and 3.5.3—both released half a decade ago. (To be clear, this is an observation about real-world production environments, not about Debian. Debian's "testing" distribution, which is widely used for development, comes with a current version of Python.)

Building Windows installers from source is beyond the scope of this book, but you can find a good step-by-step guide on Stack Overflow.

As a side benefit, `py -m pip install --upgrade pip` is the only way to upgrade pip without an *Access denied* error. Windows refuses to replace an executable while it's still running.

The *UNIX command-line tools* option places symbolic links in the */usr/local/bin* directory, which can conflict with Homebrew packages and other versions from *python.org*.

For historical reasons, framework builds use a different path for the *per-user site directory*, the location where packages are installed if you invoke pip outside of a virtual environment and without administrative privileges. This different installation layout can prevent you from importing a previously installed package.

Chapter 2. Python Environments

At their core, Python installations consist of an interpreter and the modules from the standard library (and from third-party packages, if you've installed any). Together, these provide the essential components you need to execute a Python program: a *Python environment*. Figure 2-1 shows a first approximation of a Python environment; we'll keep refining this picture throughout the present chapter.

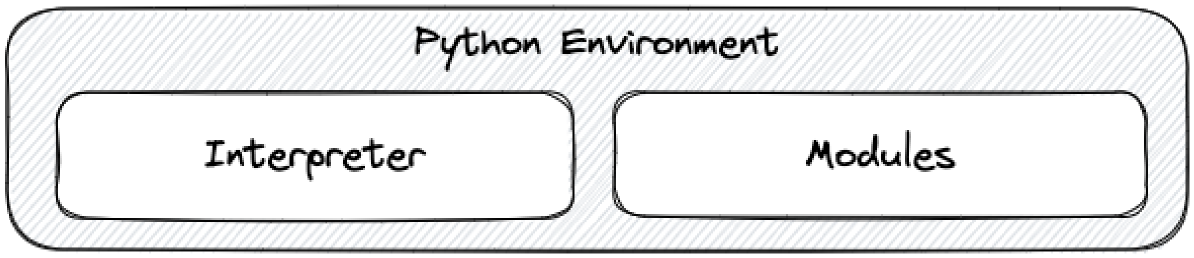

Figure 2-1. The essential components of a Python environment

Python installations aren't the only kind of Python environment. *Virtual environments* are stripped-down environments that share the interpreter and the standard library with a full installation. You use them to install project-specific modules while keeping the system-wide environment pristine. The *per-user environment* allows you to install modules for a single user. As you will see, neither of these can stand alone as a Python environment; both require a Python installation to provide an interpreter and the standard library.

Managing environments is a crucial aspect of Python development. You'll want to make sure your code works on your users' systems, particularly across the language versions you support, and possibly across major versions of an important dependency. Furthermore, a Python environment can only contain a single version of each module —if two programs require different versions of the same module, they can't be installed side-by-side. That's why it's considered good practice to install every Python application in a dedicated environment.

Python environments are primarily *runtime environments*, which is another way of saying that they provide the prerequisites for running a Python program. But environments also provide a *build environment*: They serve to build *packages*—the artifacts used to share modules with the world.

In this and the following chapter, you'll build a deeper understanding of what Python environments are and how they work, and you'll learn about tools that help you manage them efficiently. This chapter takes a look under the hood: the contents and structure of an environment, and how your code interacts with the environment. Specifically, I'll teach you how—and where—Python finds the modules you import. In the next chapter, I'll introduce you to third-party tools that help you manage environments and the modules installed in them.

Contents of a Python Environment

Figure 2-2 gives a more complete picture of the components that make a Python environment.

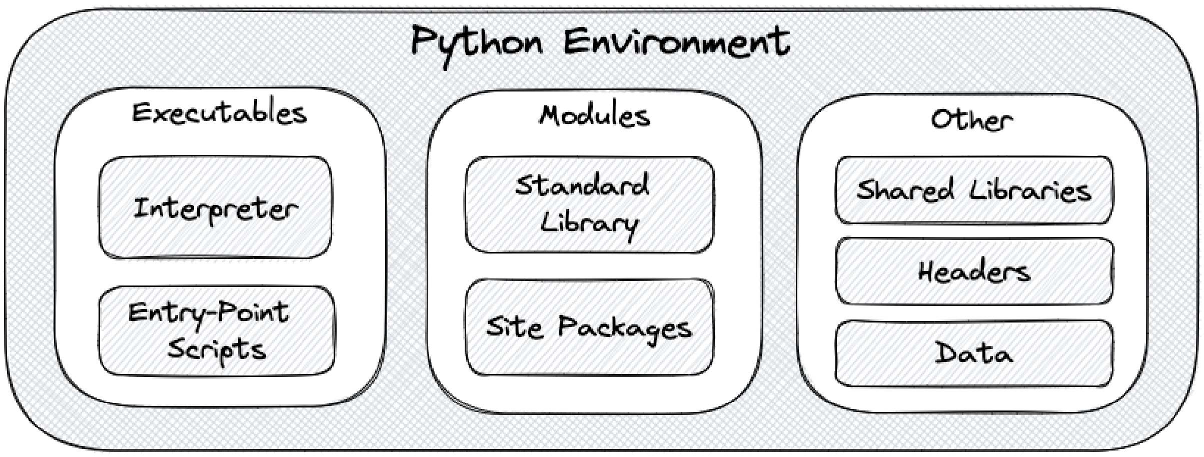

Figure 2-2. Python environments consist of an interpreter, the standard library, site packages, entry-point scripts, shared libraries, and more.

Let's take a quick inventory—feel free to follow along on your own system:

The Python Interpreter

The executable that runs Python programs is named *python.exe* on Windows and located at the root of the installation.[1] On Linux and macOS, the interpreter is named *python3.x* and stored in the *bin* directory with a *python3* symbolic link.

Python modules

Modules are containers of Python objects that you load via the `import` statement. They are organized under *Lib* (Windows) or *lib/python3.x* (Linux and macOS). While modules from the standard library are distributed with Python, *site packages* are modules you install from the Python Package Index (PyPI) or another source.

Entry-point scripts

These are executable files in *Scripts* (Windows) or *bin* (Linux and macOS). They launch Python applications by importing and invoking their entry-point function.

Shared libraries

Shared libraries contain native code compiled from low-level languages like C. Their filenames end in *.dll* or *.pyd* on Windows, *.dylib* on macOS, and *.so* on Linux. Some have a special entry point that lets you import them as modules from Python—they're known as *extension modules*. Extension modules, in turn, may use other shared libraries from the environment or the system.[2]

Headers

Python installations contain headers for the *Python/C API*, an application programming interface for writing extension modules or embedding Python as a component in a larger application. They are located under *Include* (Windows) or *include/python3.x* (Linux and macOS).

Static data

Python environments can also contain static data in various locations. This includes configuration files, documentation, and any resource files shipped with third-party packages.

The next sections take a closer look at the core parts of a Python environment: the interpreter, modules, and scripts.

NOTE

By default, Python installations also include *Tcl/Tk*, a toolkit for creating graphical user interfaces (GUIs) written in Tcl. The standard `tkinter` module allows you to use this toolkit from Python.

The Interpreter

The Python interpreter ties the environment to three things:

- a specific version of the Python language
- a specific implementation of Python
- a specific build of the interpreter

The implementation might be CPython, the reference implementation of Python, but it could also be any of a number of alternative implementations—such as PyPy, a fast interpreter with just-in-time compilation, written in Python itself. Builds differ in their instruction set architecture—for example, 32-bit versus 64-bit, or Intel versus Apple Silicon—and their build configuration, which determines things like compile-time optimizations or the installation layout.

QUERYING THE INTERPRETER ABOUT THE PYTHON ENVIRONMENT

In an interactive session, import the `sys` module and inspect the following variables:

`sys.version_info`

The version of the Python language, represented as a *named tuple* with the major, minor, and micro versions, as well as the release level and serial number for prereleases

`sys.implementation.name`

The implementation of Python, such as `"cpython"` or `"pypy"`

`sys.implementation.version`

The version of the implementation, same as `sys.version_info` for CPython

`sys.executable`

The location of the Python interpreter

`sys.prefix`

The location of the Python environment

`sys.base_prefix`

The location of the full Python installation, same as `sys.prefix` outside of a virtual environment

`sys.path`

The list of directories searched when importing Python modules

The command `py -m sysconfig` prints a great deal of metadata compiled into the Python interpreter, such as the instruction set architecture, the build configuration, and the installation layout.

Python Modules

Modules come in various forms and shapes. If you've worked with Python, you've likely used most of them already. Let's go over the different kinds:

Simple modules

In the simplest case, a *module* is a single file containing Python source code. The statement `import string` executes the code in *string.py* and binds the result to the name `string` in the local scope.

Packages

Directories with *__init__.py* files are known as *packages*—they allow you to organize modules in a hierarchy. The statement `import email.message` loads the `message` module from the `email` package.

Namespace packages

Directories with modules but no *__init__.py* are known as *namespace packages*. You use them to organize modules in a common namespace such as a company name (say `acme.unicycle` and `acme.rocketsled`). Unlike with regular packages, you can distribute each module in a namespace package separately.

Extension modules

Binary extensions are dynamic libraries with Python bindings; an example is the standard `math` module. People write them for performance reasons or to make existing C libraries available as Python modules. Their names end in *.pyd* on Windows, *.dylib* on macOS, and *.so* on Linux.

Built-in modules

Some modules from the standard library, such as the `sys` and `builtins` modules, are compiled into the interpreter. The variable `sys.builtin_module_names` lists all of them.

Frozen modules

Some modules from the standard library are written in Python but have their bytecode embedded in the interpreter. Originally, only core parts of `importlib` got this treatment. Recent versions of Python freeze every module that's imported during interpreter startup, such as `os` and `io`.

NOTE

The term *package* carries some ambiguity in the Python world. It refers both to modules and to the artifacts used for distributing modules, also known as *distributions*. Unless stated otherwise, this book uses *package* as a synonym of *distribution*.

Bytecode is an intermediate representation of Python code that is platform-independent and optimized for fast execution. The interpreter compiles pure Python modules to bytecode when it loads them for the first time. Bytecode modules are cached in the environment in *.pyc* files under *__pycache__* directories.

INSPECTING MODULES AND PACKAGES WITH IMPORTLIB

You can find out where a module comes from using `importlib` from the standard library. Every module has an associated `ModuleSpec` object whose `origin` attribute contains the location of the source file or dynamic library for the module, or a fixed string like `"built-in"` or `"frozen"`. The `cached` attribute stores the location of the bytecode for a pure Python module. Example 2-1 shows the origin of each module in the standard library.

Example 2-1. Listing standard library modules and their origin

```python
import importlib.util
import sys

for name in sorted(sys.stdlib_module_names):
    if spec := importlib.util.find_spec(name):
        print(f"{name:30} {spec.origin}")
```

Environments also store metadata about installed third-party packages, such as their authors, licenses, and versions. Example 2-2 shows the version of each package in the environment using `importlib.metadata` from the standard library.

Example 2-2. Listing packages installed in the environment

```
import importlib.metadata

distributions = importlib.metadata.distributions(
for distribution in sorted(distributions, key=lar
    print(f"{distribution.name:30} {distribution.
```

Entry-point Scripts

Package installers like pip can generate entry-point scripts for third-party packages they install. Packages only need to designate the function that the script should invoke. This is a handy method to provide an executable for a Python application.

Platforms differ in how they let you execute entry-point scripts directly. On Linux and macOS, they're regular Python files with *execute* permission (see Example 2-3). Windows embeds the Python code in a binary file in the Portable Executable (PE) format—more commonly known as a *.exe* file. The binary launches the interpreter with the embedded code.[3]

Example 2-3. The entry-point script for pydoc

```
#!/usr/local/bin/python3.11 ❶
```

```
import pydoc ❷
if __name__ == "__main__": ❸
    pydoc.cli() ❹
```

❶ Request the interpreter from the current environment using a shebang.

❷ Load the module containing the designated entry-point function.

❸ Check that the script wasn't imported from another module.

❹ Finally, call the entry-point function to start up the program.

Python installations on Linux and macOS include entry-point scripts for some applications distributed with Python. The `idle` command starts *IDLE*, an integrated development environment (IDE) for Python, which is based on the Tcl/Tk GUI toolkit. The `pydoc` command starts a documentation browser that displays docstrings embedded in modules. (If you've ever called the built-in `help()` function, you've used its console viewer.)

NOTE

On Windows, you won't find IDLE and pydoc in the *Scripts* directory. IDLE is available from the Windows Start Menu. Pydoc does not come with an entry-point script—use `py -m pydoc` instead.

Most environments also include an entry-point script for pip itself. You should prefer the more explicit form `py -m pip` over the plain `pip` command though. It gives you more control over the target environment for the packages you install.

The script directory of a Python installation also contains some executables that aren't scripts, such as the interpreter, platform-specific variants of the interpreter, and the `python3.x-config` tool used for the build configuration of extension modules.

The Layout of Python Installations

In this section, I'll discuss how Python installations are structured internally, and where you can find them on the major platforms. The location and layout of Python installations varies quite a bit from system to system. The good news is you rarely have to care—a Python interpreter knows its environment. This section is supposed to help you when things go wrong—for example, when you've mistakenly installed a package system-wide instead of within a virtual environment, or when you’re wondering where a specific package may have come from.

Table 2-1 shows some common locations of Python installations. An installation might be nicely separated from the rest of your system, but not necessarily: On Linux, it goes into a shared location like */usr* or */usr/local*, with its files scattered across the filesystem. Windows systems, on the other hand, keep all files in a single place. Framework builds on macOS are similarly self-contained, although distributions may also install symbolic links into the traditional Unix locations.

Table 2-1. Locations of Python installations

Platform	Python installation
Windows (single-user)	*%LocalAppData%\Programs\Python\Python3x*
Windows (multi-user)	*%ProgramFiles%\Python3x*
macOS (Homebrew)	*/opt/homebrew/Frameworks/Python.framework/Vers*
macOS (python.org)	*/Library/Frameworks/Python.framework/Versions/3.x*
Linux (generic)	*/usr/local*
Linux (package manager)	*/usr*

[a] Homebrew on macOS Intel uses */usr/local* instead of */opt/homebrew*.

Table 2-2 provides a baseline for installation layouts on the major platforms: the locations of the interpreter, the standard library, third-party modules, and entry-point scripts within the installation. The location for third-party modules is known as the *site packages* directory, and the location for entry-point scripts as the *script* directory.

Table 2-2. Layout of Python installations

Files	Windows	Linux and macOS	Notes
Interpreter	installation root	*bin*	Virtual environments on Windows have the interpreter in *Scripts*.
Standard library	*Lib* and *DLLs*	*lib/python3.x*	Extension modules are located under *DLLs* on Windows. Fedora places the standard library under *lib64* instead of *lib*.

Files	**Windows**	**Linux and macOS**	**Notes**
Site packages	*Lib\site-packages*	*lib/python3.x/site-packages*	Debian and Ubuntu name the system site packages *dist-packages*. Fedora places extension modules under *lib64* instead of *lib*.
Scripts	*Scripts*	*bin*	

Linux distributions may have site packages and script directories under both */usr* and */usr/local*. These systems allow only the official package manager to write to the */usr* hierarchy. If you install packages using pip with administrative privileges, they end up in a parallel hierarchy under */usr/local*. (Don’t do this; use the package manager, the per-user environment, or a virtual environment instead.)

INSTALLATION SCHEMES

Python describes the layout of environments using *installation schemes*. Each installation scheme has a name and the locations of some well-known directories: `stdlib` and `platstdlib` for the standard library, `purelib` and `platlib` for third-party modules, `scripts` for entry-point scripts, `include` and `platinclude` for headers, and `data` for data files. The `plat*` directories are for platform-specific files like binary extensions.

The `sysconfig` module defines installation schemes for the major operating systems and the different kinds of environments—system-wide installations, per-user installations, and virtual environments. Downstream distributions like Debian and Fedora often register additional installation schemes. The main customer of installation schemes are package installers like pip, as they need to decide where the various parts of a Python package should go.

You can print the installation scheme for the current environment using the command `py -m sysconfig`. Example 2-4 shows how to list all available installation schemes. (You're not expanding configuration variables like the installation root here; they're only meaningful within the current environment.)

Example 2-4. Listing installation schemes

```
import sysconfig

for scheme in sorted(sysconfig.get_scheme_names(
    print(f"==> {scheme} <==")
    for name in sorted(sysconfig.get_path_names(
        path = sysconfig.get_path(name, scheme, e
        print(f"{name:20} {path}")
```

The Per-User Environment

The *per-user environment* allows you to install third-party packages for a single user. It offers two main benefits over installing packages system-wide: You don't need administrative privileges to install packages, and you don't affect other users on a multi-user system.

The per-user environment is located in the home directory on Linux and macOS and in the app data directory on Windows (see Table 2-3). It contains a site packages directory for every Python version. The script directory is shared across Python versions.[4]

Table 2-3. Location of per-user directories

Files	Windows	macOS (framework)	Linux
Per-user root	*%AppData%\Python*	*~/Library/Python/3.x*	*~/.local*
Site packages	*Python3x\site-packages*	*lib/python/site-packages*	*lib/pyth packag*
Scripts	*Scripts*	*bin*	*bin*

[a] Fedora places extension modules under *lib64*.

You install a package into the per-user environment using `pip install --user`. If you invoke `pip` outside of a virtual environment and pip finds that it cannot write to the system-wide installation, it will also default to this location. If the per-user environment doesn't exist yet, pip creates it for you.

TIP

The per-user script directory may not be on `PATH` by default. If you install applications into the per-user environment, remember to edit your shell profile to update the search path. Pip issues a friendly reminder when it detects this situation.

Per-user environments are not isolated environments: You can still import system-wide site packages if they're not shadowed by per-user modules with the same name. Likewise, distribution-owned Python applications can see modules from the per-user environment. Applications in the per-user environment also aren't isolated from each other. In particular, they cannot depend on incompatible versions of another package.

In "Installing Applications with Pipx", I'll introduce pipx, which lets you install applications in isolated environments. It uses the per-user script directory to put applications onto your search path, but relies on virtual environments under the hood.

Virtual Environments

When you're working on a Python project that uses third-party packages, it's usually a bad idea to install these packages into the system-wide environment. There are two main reasons why you want to avoid doing this: First, you're polluting a global namespace. Testing and debugging your projects gets a lot easier when you run them in isolated and reproducible environments. If two projects depend on conflicting versions of the same package, a single environment isn't even an option. Second, your distribution or operating system may have carefully curated the system-wide environment. Installing and uninstalling packages behind the back of its package manager introduces a real chance of breaking your system.

Virtual environments were invented to solve these problems. They're isolated from the system-wide installation and from each other. Under the hood, a virtual environment is a lightweight Python environment that stores third-party packages and a reference to its parent environment. Packages in virtual environments are only visible to the interpreter in the environment.

You create a virtual environment with the command `py -m venv` *`dir`*. The last argument is the location where you want the environment to exist—its root directory. The directory tree of a virtual environment looks much like a Python installation, except that some files are missing, most notably the entire standard library. Table 2-4 shows the standard locations within a virtual environment.

Table 2-4. Structure of a virtual environment

Files	Windows	Linux and macOS
Interpreter	*Scripts*	*bin*
Scripts	*Scripts*	*bin*
Site packages	*Lib\site-packages*	*lib/python3.x/site-packages*[a]
Environment Configuration	*pyvenv.cfg*	*pyvenv.cfg*

[a] Fedora places third-party extension modules under *lib64* instead of *lib*.

Virtual environments have their own interpreter, which is located in the script directory. On Linux and macOS, this is a symbolic link to the interpreter you used to create the environment. On Windows, it's a small wrapper executable that launches the parent interpreter.[5]

Virtual environments include pip as a means to install packages into them. Let's create a virtual environment, install `httpx` (an HTTP client library), and launch an interactive session. On Windows, enter the commands below.

```
> py -m venv venv
> venv\Scripts\python.exe -m pip install httpx
> venv\Scripts\python.exe -m pip install httpx
> venv\Scripts\python.exe
```

On Linux and macOS, enter the commands below. There's no need to spell out the path to the interpreter if the environment uses the well-known name *.venv*. The Python Launcher for Unix selects its interpreter by default.

```
$ py -m venv .venv
$ py -m pip install httpx
$ py
```

In the interactive session, use `httpx.get` to perform a `GET` request to a web host:

```
>>> import httpx
>>> httpx.get("https://example.com/")
<Response [200 OK]>
```

You might think that the interpreter must somehow hardcode the locations of the standard library and site packages. That's actually not how it works. Rather, the interpreter looks at the location of its own executable and checks its parent directory for a *pyvenv.cfg* file. If it finds one, it treats that file as a *landmark* for a virtual environment and imports third-party modules from the site packages directory beneath.

This explains how Python knows to import third-party modules from the virtual environment, but how does it find modules from the standard library? After all, they're neither copied nor linked into the virtual environment. Again, the answer lies in the *pyvenv.cfg* file: When you create a virtual environment, the interpreter records its own location under the `home` key in this file. If it later finds itself in a virtual environment, it looks for the standard library relative to that `home` directory.

NOTE

The name *pyvenv.cfg* is a remnant of the `pyvenv` script which used to ship with Python. The `py -m venv` form makes it clearer which interpreter you use to create the virtual environment—and thus which interpreter the environment itself will use.

While the virtual environment has access to the standard library in the system-wide environment, it's isolated from its third-party modules. Although not recommended, you can give the environment access to those modules as well, using the `--system-site-packages` option when creating the environment. The result is quite similar to the way a per-user environment works.

How does pip know where to install packages? The short answer is that pip asks the interpreter it's running on, and the interpreter derives the location from its own path—just like when you import a module.[6] This is why it's best to run pip with an explicit interpreter using the `py -m pip` idiom. If you invoke `pip` directly, the system searches your `PATH` and may come up with the entry-point script from a different environment.

Virtual environments come with the version of pip that was current when Python was released. This can be a problem when you're working with an old Python release. Create the environment with the option `--upgrade-deps` to ensure you get the latest pip release from the Python Package Index. This method also upgrades any additional packages that may be pre-installed in the environment.

NOTE

Besides pip, virtual environments may pre-install `setuptools` for the benefit of legacy packages that don't declare it as a build dependency. This is an implementation detail and subject to change, so don't assume `setuptools` will be present.

Activation Scripts

Virtual environments come with *activation scripts* in the script directory—these scripts make it more convenient to use a virtual environment from the command line, and they're provided for a number of supported shells and command interpreters. Here's the Windows example again, this time using the activation script:

```
> py -m venv venv
> venv\Scripts\activate
(venv) > py -m pip install httpx
(venv) > py
```

Activation scripts bring three features to your shell session:

- They prepend the script directory to the `PATH` variable. This allows you to invoke `python`, `pip`, and entry-point scripts without prefixing them with the path to the environment.
- They set the `VIRTUAL_ENV` environment variable to the location of the virtual environment. Tools like the Python Launcher use this variable to detect that the environment is active.
- They update your shell prompt to provide a visual reference which environment is active, if any. By default, the prompt uses the name of the directory where the environment is located.

TIP

You can provide a custom prompt using the option `--prompt` when creating the environment. The special value `.` designates the current directory; it's particularly useful when you're inside a project repository.

On macOS and Linux, you need to *source* the activation script to allow it to affect your current shell session. Here's an example for Bash and similar shells:

```
$ source .venv/bin/activate
```

Environments come with activation scripts for some other shells, as well. For example, if you use the Fish shell, source the supplied *activate.fish* script instead.

On Windows, you can invoke the activation script directly. There's an *Activate.ps1* script for PowerShell and an *activate.bat* script for *cmd.exe*. You don't need to provide the file extension; each shell selects the script appropriate for it.

```
> venv\Scripts\activate
```

PowerShell on Windows doesn't allow you to execute scripts by default, but you can change the execution policy to something more suited to development: The `RemoteSigned` policy allows scripts written on the local machine or signed by a trusted publisher. On Windows servers, this policy is already the default. You only need to do this once—the setting is stored in the registry.

```
> Set-ExecutionPolicy -ExecutionPolicy RemoteSig
```

Activation scripts provide you with a `deactivate` command to revert the changes to your shell environment. It's usually implemented as a shell function, and works the same on Windows, macOS, and Linux.

```
$ deactivate
```

Installing Applications with Pipx

In the previous section, you saw why it makes good sense to install your projects in separate virtual environments: unlike system-wide and per-user environments, virtual environments isolate your projects, avoiding dependency conflicts.

The same reasoning applies when you install third-party Python applications—say, a code formatter like Black or a packaging manager like Hatch. Applications tend to depend on more packages than libraries, and they can be quite picky about the versions of their dependencies.

Unfortunately, managing and activating a separate virtual environment for every application is cumbersome and confusing—and it limits you to using only a single application at a time. Wouldn't it be great if we could confine applications to virtual environments and still have them available globally?

That's precisely what pipx does, and it leverages a simple idea to make it possible: it copies the entry-point script for the application from its virtual environment into a directory on your search path. Entry-point scripts contain the full path to the environment's interpreter, so you can copy them anywhere you want, and they'll still work.

Let me show you how this works under the hood. The example commands below work on macOS and Linux using a Bash-like shell. First, you create a shared directory for the entry-point scripts of your applications and add it to your `PATH` environment variable:

```
$ mkdir bin
$ export PATH="$(pwd)/bin:$PATH"
```

Next, you install an application in a dedicated virtual environment—I've chosen the Black code formatter as an example:

```
$ py -m venv venvs/black
$ venvs/black/bin/python -m pip install black
Successfully installed black-22.12.0 [...]
```

Finally, you copy the entry-point script into the directory you created in the first step—that would be a script named `black` in the `bin` or `Scripts` directory of the environment:

```
$ cp venvs/black/bin/black bin
```

Now you can invoke `black` even though the virtual environment is not active:

```
$ black --version
black, 22.12.0 (compiled: no)
Python (CPython) 3.11.1
```

On top of this simple idea, the pipx project has built a cross-platform package manager for Python applications with a great developer experience.

TIP

If there’s a single Python application that you should install on a development machine, pipx is probably it. It lets you install, run, and manage all the other Python applications in a way that’s convenient and avoids trouble.

If your system package manager distributes pipx as a package, I recommend using that as the preferred installation method, as it’s more likely to provide good integration out-of-the-box, such as shell completion:

```
$ apt install pipx
$ brew install pipx
$ dnf install pipx
```

Otherwise, I recommend installing pipx into the per-user environment, like this:

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

As a post-installation step, update your `PATH` environment variable to include the shared script directory, using the `ensurepath` sub-command. If you didn't use the system package manager, this step also puts the `pipx` command itself on your search path.

```
$ py -m pipx ensurepath
```

If you don't already have shell completion for pipx, activate it by following the instructions for your shell, which you can print with this command:

```
$ pipx completions
```

With pipx installed on your system, you can use it to install and manage applications from the Python Package Index (PyPI). For example, here's how you would install Black with pipx:

```
$ pipx install black
```

You can also use pipx to upgrade an application to a new release, reinstall it, or uninstall it from your system:

```
$ pipx upgrade black
$ pipx reinstall black
$ pipx uninstall black
```

As a package manager, pipx keeps track of the applications it installs and lets you perform bulk operations across all of them. This is particularly useful to keep your development tools updated to the latest version and to reinstall them on a new version of Python.

```
$ pipx upgrade-all
$ pipx reinstall-all
$ pipx uninstall-all
```

You can also list the applications you've installed previously:

```
$ pipx list
```

The commands above provide all the primitives to manage global developer tools efficiently, but it gets better. Most of the time, you just want to use recent versions of your developer tools. You don't want the responsibility of keeping the tools updated, reinstalling them on new Python versions, or removing them when you no longer need them. Pipx allows you to run an application directly from PyPI without an explicit installation step. Let's use the classic Cowsay app to try it:

```
$ pipx run cowsay moo
  ___
| moo |
  ===
   \
    \
      ^__^
      (oo)\_______
      (__)\       )\/\
          ||----w |
          ||     ||
```

Behind the scenes, pipx installs Cowsay in a temporary virtual environment and runs it with the arguments you've provided. It keeps the environment around for a while,[7] so you don't end up reinstalling applications on every run. Use the `--no-cache` option to force pipx to create a new environment and reinstall the latest version.

TIP

Use `pipx run [app]` as the default method to install and run developer tools from PyPI. Use `pipx install [app]` if you need more control over application environments, for example if you need to install plugins. Replace `[app]` with the name of the app.

One situation where you may prefer to install an application explicitly is when the application supports plugins that extend its functionality. These plugins must be installed in the same environment as the application. For example, the packaging managers Hatch and Poetry both come with plugin systems.

Here's how you would install Hatch with a plugin that determines the package version from the version control system:

```
$ pipx install hatch
$ pipx inject hatch hatch-vcs
```

By default, pipx installs applications on the same Python version that it runs on itself. This may not be the latest stable version, particularly if you installed pipx using a system package manager like Apt. I recommend setting the environment variable `PIPX_DEFAULT_PYTHON` to the latest stable Python if that’s the case. Many developer tools you run with pipx create their own virtual environments; for example, virtualenv, Nox, tox, Poetry, and Hatch all do. It’s worthwhile to ensure that all downstream environments use a recent Python version by default.

```
$ export PIPX_DEFAULT_PYTHON=python3.11 # Linux a
> setx PIPX_DEFAULT_PYTHON python3.11   # Windows
```

Under the hood, pipx uses pip as a package installer. This means that any configuration you have for pip also carries over to pipx. A common use case is installing Python packages from a private index instead of PyPI, such as a company-wide package repository. You can use `pip config` to set the URL of your preferred package index persistently:

```
$ pip config set global.index-url https://exampl
```

Alternatively, you can set the package index for the current shell session only. Most pip options are also available as environment variables:

```
$ export PIP_INDEX_URL=https://example.com
```

Both methods cause pipx to install applications from the specified index.

Finding Python Modules

Python environments consist, first and foremost, of a Python interpreter and Python modules. Consequently, there are two mechanisms that play a key role in linking a Python program to an environment. *Interpreter discovery* is the process of locating the Python interpreter to execute a program. You've already seen the most important methods for locating interpreters:

- Entry-point scripts reference the interpreter in their environment directly, using a shebang or a wrapper executable (see "Entry-point Scripts").
- Shells locate the interpreter by searching directories on `PATH` for commands like `python`, `python3`, or `python3.x` (see "Locating Python Interpreters").
- The Python Launcher locates interpreters using the Windows Registry, `PATH` (on Linux and macOS), and the `VIRTUAL_ENV` variable (see "The Python Launcher for Windows" and "The Python Launcher for Unix").

- When you activate a virtual environment, the activation script puts its interpreter and entry-point scripts on `PATH`. It also sets the `VIRTUAL_ENV` variable for the Python Launcher and other tools (see "Virtual Environments").

In this section, we'll take a deep dive into the other mechanism that links programs to an environment: *module import*, or more specifically, how the import system locates Python modules for a program. In a nutshell, just like the shell searches `PATH` for executables, Python searches `sys.path` for modules. This variable holds a list of locations from where Python can load modules—most commonly, directories on the local filesystem.

The machinery behind the `import` statement lives in `importlib` from the standard library (see "Inspecting modules and packages with importlib"). The interpreter translates every use of the `import` statement into an invocation of the `__import__` function from `importlib`. The `importlib` module also exposes an `import_module` function that allows you to import modules whose names are only known at runtime.

Having the import system in the standard library has powerful implications: You can inspect and customize the import mechanism from within Python. For example, the import system supports loading modules from directories and from zip archives out of the box. But entries on `sys.path` can be anything really—say, a URL or a database query—as long as you register a function in `sys.path_hooks` that knows how to find and load modules from these path entries.

Module Objects

When you import a module, the import system returns a *module object*, an object of type `types.ModuleType`. Any global variable defined by the imported module becomes an attribute of the module object. This allows you to access the module variable in dotted notation (`module.var`) from the importing code.

Under the hood, module variables are stored in a dictionary in the `__dict__` attribute of the module object. (This is the standard mechanism used to store attributes of any Python object.) When the import system loads a module, it creates a module object and executes the module's code using `__dict__` as the global namespace. Somewhat simplified, it invokes the built-in `exec` function like this:

```
exec(code, module.__dict__)
```

Additionally, module objects have some special attributes. For instance, the `__name__` attribute holds the fully-qualified name of the module, like `email.message`. The `__spec__` module holds the *module spec*, which I'll talk about shortly. Packages also have a `__path__` attribute, which contains locations to search for submodules.

NOTE

Most commonly, the `__path__` attribute of a package contains a single entry: the directory holding its *__init__.py* file. Namespace packages, on the other hand, can be distributed across multiple directories.

The Module Cache

When you first import a module, the import system stores the module object in the `sys.modules` dictionary, using its fully-qualified name as a key. Subsequent imports return the module object directly from `sys.modules`. This mechanism brings a number of benefits:

Performance

> Import is expensive because the import system loads most modules from disk. Importing a module also involves executing its code, which can further increase startup time. The `sys.modules` dictionary functions as a cache to speed things up.

Idempotency

Importing modules can have side effects, for example by executing module-level statements. Caching modules in `sys.modules` ensures that these side effects happen only once. The import system also uses locks to ensure that multiple threads can safely import the same module.

Recursion

Modules can end up importing themselves recursively. A common case is circular imports, where module `a` imports module `b`, and `b` imports `a`. The import system supports this by adding modules to `sys.modules` *before* they're executed. When `b` imports `a`, the import system returns the (partially initialized) module `a` from the `sys.modules` dictionary, thereby preventing an infinite loop.

Module Specs

Conceptually, importing a module proceeds in two steps. First, given the fully-qualified name of a module, the import system locates the module and produces a *module spec*. The module spec (`importlib.machinery.ModuleSpec`) contains metadata about the module such as its name and location, as well as an appropriate *loader* for the module. Second, the import system creates a module object from the module spec and executes the module's code. The module object includes special attributes with most of the metadata from the module spec (see Table 2-5). These two steps are referred to as *finding* and *loading*, and the module spec is the link between them.

Table 2-5. Attributes of Modules and Module Specs

Module attribute	Module spec attribute	Description
`__name__`	`name`	The fully-qualified name of the module.
`__loader__`	`loader`	A loader object that knows how to execute the module's code.
`__file__`	`origin`	The location of the module. This is often the filename of a Python module, but it can also be a fixed string like "builtin" for built-in modules, or `None` for namespace packages (which don't have a single location).
`__path__`	`submodule_search_locations`	Locations to search for submodules, if the module is a package.
`__cached__`	`cached`	The location of the compiled bytecode for the module.

Module attribute	Module spec attribute	Description
`__package__`	`parent`	The fully-qualified name of the package that contains the module, or the empty string for top-level modules.

Finders and Loaders

The import system finds and loads modules using two kinds of objects. *Finders* (`importlib.abc.MetaPathFinder`) are responsible for locating modules given their fully-qualified name. When successful, their `find_spec` method returns a module spec with a loader; otherwise, it returns `None`. *Loaders* (`importlib.abc.Loader`) are objects with an `exec_module` function which loads and executes the module's code. The function takes a module object and uses it as a namespace when executing the module. The finder and loader can be the same object, in which case they're known as an *importer*.

Finders are registered in the `sys.meta_path` variable, and the import system tries each finder in turn. When a finder has returned a module spec with a loader, the import system creates and initializes a module object. The import system then passes the module object to the loader for execution.

By default, the `sys.meta_path` variable contains three finders, which handle different kinds of modules (see "Python Modules").

- `importlib.machinery.BuiltinImporter` for built-in modules
- `importlib.machinery.FrozenImporter` for frozen modules
- `importlib.machinery.PathFinder` to search modules on `sys.path`

The `PathFinder` is the central hub of the import machinery. It's responsible for every module that's not embedded into the interpreter, and searches `sys.path` to locate it.[8] The path finder uses a second level of finder objects known as *path entry finders* (`importlib.abc.PathEntryFinder`), each of which finds modules under a specific location on `sys.path`. The standard library provides two types of path entry finders, registered under `sys.path_hooks`:

- `zipimport.zipimporter` to import modules from zip archives

- `importlib.machinery.FileFinder` to import modules from a directory

Typically, modules are stored in directories on the filesystem, so `PathFinder` delegates its work to a `FileFinder`. The latter scans the directory for the module, and uses its file extension to determine the appropriate loader. There are three loaders for the different kinds of modules:

- `importlib.machinery.SourceFileLoader` for pure Python modules
- `importlib.machinery.SourcelessFileLoader` for bytecode modules
- `importlib.machinery.ExtensionFileLoader` for binary extension modules

The zip importer works similarly, except that it does not support extension modules. This is due to the fact that current operating systems don't allow loading dynamic libraries from a zip archive.

The Module Path

When your program cannot find a specific module, or when it imports the wrong version of a module, it can help to take a look at `sys.path`, the module path. But where do the entries on `sys.path` come from, in the first place? Let's unravel some of the mysteries around the module path.

NOTE

If you're curious, you can find the built-in logic for constructing `sys.path` in the CPython source code in *Modules/getpath.py*. Despite appearances, this is not an ordinary Python module. When you build Python, the code in this file is frozen to bytecode and embedded in the executable.

When the interpreter starts up, it constructs the module path in two steps. First, it builds an initial module path using some built-in logic. Most importantly, this initial path includes the standard library. Second, the interpreter imports the `site` module from the standard library. The `site` module extends the module path to include the site packages from the current environment. In this section, we'll take a look at how the interpreter constructs the initial module path with the standard library. The next section explains how the `site` module appends directories with site packages.

The locations on the initial module path fall into three categories, and they occur in the order given below:

1. The current directory or the directory of the Python script (if any)
2. The locations in the `PYTHONPATH` environment variable (if set)

3. The locations of the standard library

Let’s look at each in more detail.

The script or current directory

The first item on `sys.path` can be any of the following:

- If you ran `py script`, the directory where *script* is.
- If you ran `py -m module`, the current directory.
- Otherwise, the empty string, which also denotes the current directory.

Traditionally, this mechanism provided an easy way to structure an application: Just put the main entry-point script and all application modules in the same directory. During development, launch the interpreter from within that directory for interactive debugging, and your imports still work.

WARNING

Unfortunately, having the working directory on `sys.path` is quite unsafe, as an attacker (or you, mistakenly) can override the standard library by placing Python files in the victim’s directory.

Installing your application into a virtual environment is both a safer and more flexible option. This requires packaging the application, which is the topic of Chapter 3. From Python 3.11, you can use the `-P` interpreter option or the `PYTHONSAFEPATH` environment variable to omit the current directory from `sys.path`. If you invoke the interpreter with a script, this option also omits the directory where the script is located.

The `PYTHONPATH` variable

The `PYTHONPATH` environment variable provides another way to add locations before the standard library on `sys.path`. It uses the same syntax as the `PATH` variable. Avoid this mechanism for the same reasons as the current working directory and use a virtual environment instead.

The standard library

Table 2-6 shows the remaining entries on the initial module path, which are dedicated to the standard library. Locations are prefixed with the path to the installation, and may differ in details on some platforms.

Table 2-6. The standard library on `sys.path`

Windows	Linux and macOS	Description
python3x.zip	*lib/python3x.zip*	For compactness, the standard library can be installed as a zip archive. This entry is present even if the archive doesn't exist (which it normally doesn't).
lib/python3.x	*Lib*	Pure Python modules
lib/python3.x/lib-dynload	*DLLs*	Binary extension modules

The location of the standard library is not hardcoded in the interpreter (see "Virtual Environments"). Rather, Python looks for landmark files on the path to its own executable, and uses them to locate the current environment (`sys.prefix`) and the Python installation (`sys.base_prefix`). One such landmark file is *pyvenv.cfg*, which marks a virtual environment and points to its parent installation via the `home` key. Another landmark is *os.py*, the file containing the standard `os` module: Python uses *os.py* to discover the prefix outside of a virtual environment, and to locate the standard library itself.

Site Packages

The interpreter constructs the initial `sys.path` early on during initialization using a fairly fixed process. By contrast, the remaining locations on `sys.path`—known as *site packages*—are highly customizable and under the responsibility of a Python module named `site`.

The `site` module adds the following path entries if they exist on the filesystem:

User site packages

This directory holds third-party modules from the per-user environment. It's in a fixed location that depends on the OS (see "The Per-User Environment"). On Fedora and some other systems, there are two path entries, for pure Python modules and extension modules, respectively.

Site packages

This directory holds third-party modules from the current environment, which is either a virtual environment or a system-wide installation. On Fedora and some other systems, pure Python modules and extension modules are in separate directories. Many Linux systems also separate distribution-owned site packages under */usr* from local site packages under */usr/local*.

In the general case, the site packages are in a subdirectory of the standard library named *site-packages*. If the `site` module finds a *pyvenv.cfg* file on the interpreter path, it uses the same relative path as in a system installation, but starting from the virtual environment marked by that file. The `site` module also modifies `sys.prefix` to point to the virtual environment.

The `site` module provides a few hooks for customization:

.pth files

> Within site packages directories, any file with a *.pth* extension can list additional directories for `sys.path`, one directory per line. This works similar to `PYTHONPATH`, except that modules in these directories will never shadow the standard library. Additionally, *.pth* files can import modules directly—the `site` module executes any line starting with `import` as Python code. Third-party packages can ship *.pth* files to configure `sys.path` in an environment. Some packaging tools use *.pth* files behind the scenes to implement *editable installs*. An editable install places the source directory of your project on `sys.path`, making code changes instantly visible inside the environment.

The `sitecustomize` module

After setting up `sys.path` as described above, the `site` module attempts to import the `sitecustomize` module, typically located in the *site-packages* directory. This provides a hook for the system administrator to run site-specific customizations when the interpreter starts up.

The `usercustomize` module

If there is a per-user environment, the `site` module also attempts to import the `usercustomize` module, typically located in the user *site-packages* directory. You can use this module to run user-specific customizations when the interpreter starts up. Contrast this with the `PYTHONSTARTUP` environment variable, which allows you to specify a Python script to run before interactive sessions, within the same namespace as the session.

Summary

In this chapter, you've learned what Python environments are, where to find them, and how they look on the inside. At the core, a Python environment consists of the Python interpreter and Python modules, as well as entry-point scripts to run Python applications. Environments are tied to a specific version of the Python language.

There are three kinds of Python environments. *Python installations* are complete, stand-alone environments with an interpreter and the full standard library. *Per-user environments* are annexes to an installation where you can install modules and scripts for a single user. *Virtual environments* are lightweight environments for project-specific modules and scripts, which reference their parent environment via a *pyvenv.cfg* file. They come with an interpreter, which is typically a symbolic link or small wrapper for the parent interpreter, and with activation scripts for shell integration. Use the command `py -m venv` to create a virtual environment.

Finally, you've seen how Python uses `sys.path` to locate modules when you import them, and how the module path is constructed during interpreter startup. You've also learned how module import works under the hood, using finders and loaders as well as the module cache. Interpreter discovery and module import are the key mechanisms that link Python programs to an environment at runtime.

[1] There's also a *pythonw.exe* executable that runs programs without a console window, like GUI applications.

[2] For example, the standard `ssl` module uses OpenSSL, an open-source library for secure communication.

You can also execute a plain Python file on Windows if it has a *.py* or *.pyw* file extension—Windows installers associate these file extensions with the Python Launcher and register them in the `PATHEXT` environment variable. For example, Windows installations use this mechanism to launch IDLE.

Framework builds on macOS use a version-specific directory for scripts, as well. Historically, framework builds pioneered per-user installation before its standardization.

You could force the use of symbolic links on Windows via the `--symlinks` option—but don't. There are subtle differences in the way these work on Windows. For example, the File Explorer resolves the symbolic link before it launches Python, which prevents the interpreter from detecting the virtual environment.

Internally, pip queries the `sysconfig` module for an appropriate installation scheme, see "Installation Schemes". This module constructs the installation scheme using the build configuration of Python and the location of the interpreter in the filesystem.

At the time of writing, pipx caches temporary environments for 14 days.

For modules located within a package, the `__path__` attribute of the package takes the place of `sys.path`.

Chapter 3. Python Packages

In this chapter you'll learn how to package your Python projects for distribution. A *package* is a single file containing an archive of your code along with metadata that describes it, like the project name and version. You can install this file into a Python environment using pip, the Python package installer. You can also upload the package to a repository such as the Python Package Index (PyPI), a public server operated by the Python community. Having your package on PyPI means other people can install it, too—they only need to pass its name to `pip install`.

NOTE

Python folks use the word *package* for two distinct concepts. *Import packages* are Python modules that contain other modules, typically directories with an *__init__.py* file. *Distribution packages* are archive files for distributing Python software—they are the subject of this chapter.

Creating a package from your project makes it easy to share your code with others. Packaging also has a less obvious benefit: Installing your project as a package makes it a first-class citizen of a Python environment. The metadata in a package specifies the minimum Python version and any third-party packages it depends on. Installers ensure the environment matches these prerequisites; they even install missing project dependencies and upgrade those whose version doesn't match the requirements. Once installed, the package has an explicit link to the environment it's installed in. Compare this to running a script from your working directory, which may well end up on an outdated Python version, or in an environment that doesn't have all the dependencies installed.

Figure 3-1 shows the typical lifecycle of a package. Everything starts with a *project*: the source code of an application, library, or other piece of software that you're going to package for distribution (1). Next, you build a package from the project, an installable artifact with a snapshot of your project at this point in time (2). If author and user are the same person, they may install this package directly into an environment, say, for testing (5). If they are different people, it's more practical to upload the package to a *package index* (a fancy word for a package repository) (3). Think of a package index as a file server specifically for software packages, which allows people to retrieve packages by name and version. Once downloaded (4), a user can install your package into their environment (5). In real life, tools often combine downloading and installing, building and installing, and even building and publishing, into a single command.

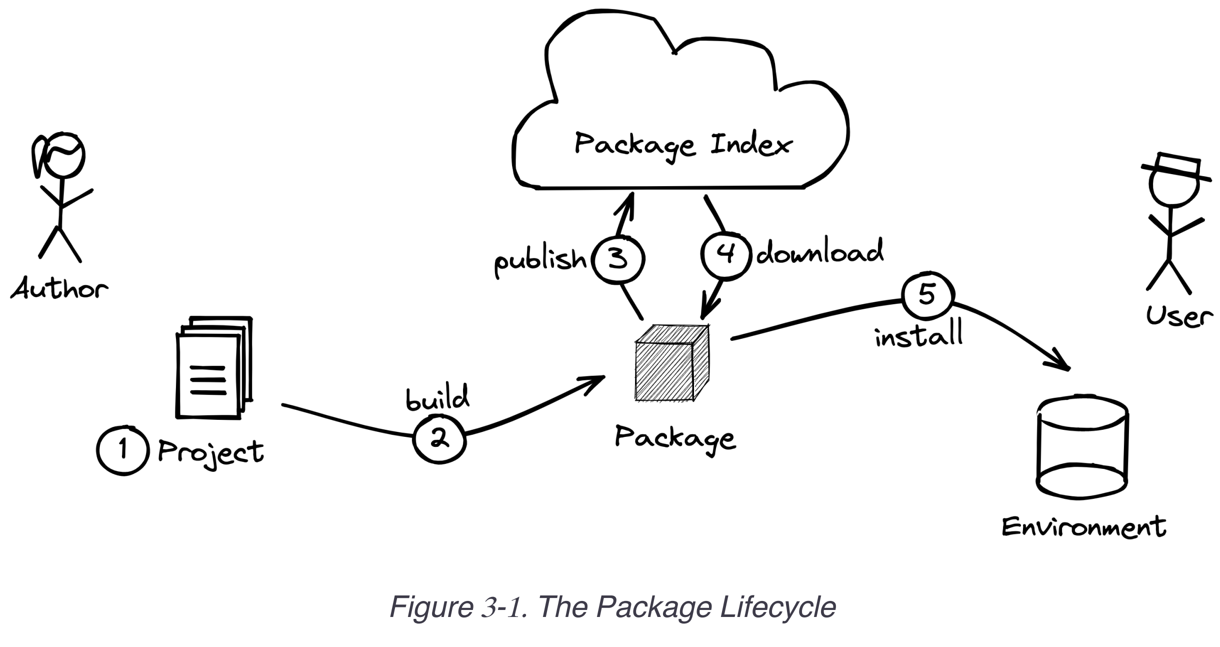

Figure 3-1. The Package Lifecycle

An Example Application

Many applications start out as small, ad-hoc scripts. Example 3-1 fetches a random article from Wikipedia and displays its title and summary in the console. The script restricts itself to the standard library, so it runs in any Python 3 environment.

Example 3-1. Displaying an extract from a random Wikipedia article

```
import json
import textwrap
import urllib.request

API_URL = "https://en.wikipedia.org/api/rest_v1/p

def main():
    with urllib.request.urlopen(API_URL) as respo
        data = json.load(response) ❸

    print(data["title"]) ❹
    print()
    print(textwrap.fill(data["extract"])) ❹

if __name__ == "__main__":
    main()
```

❶

The `API_URL` constant points to the REST API of the English Wikipedia—or more specifically, its `/page/random/summary` endpoint.

❷ The `urllib.request.urlopen` invocation sends an HTTP GET request to the Wikipedia API. The `with` statement ensures that the connection is closed at the end of the block.

❸ The response body contains the resource data in JSON format. Conveniently, the response is a file-like object, so the `json` module can load it like a file from disk.

❹ The `title` and `extract` keys hold the title of the Wikipedia page and a short plain text extract, respectively. The `textwrap.fill` function wraps the text so that every line is at most 70 characters long.

Store this script in a file *random_wikipedia_article.py* and take it for a spin. Here's a sample run:

```
> py random_wikipedia_article.py
Jägersbleeker Teich

The Jägersbleeker Teich in the Harz Mountains of
is a storage pond near the town of Clausthal-Zel
county of Goslar in Lower Saxony. It is one of th
that were created for the mining industry.
```

Why Packaging?

Sharing a script like Example 3-1 does not require packaging. You can publish it on a blog or a hosted repository, or send it to friends by email or chat. Python's ubiquity, the "batteries included" approach of its standard library, and its nature as an interpreted language make this possible. The Python programming language predates the advent of language-specific package repositories, and the ease of sharing modules with the world was a boon to Python's adoption in the early days.[1]

Distributing self-contained modules without packaging seems like a great idea at first: You keep your projects free of packaging cruft. They require no separate artifacts, no intermediate steps like building, and no dedicated tooling. But using modules as the unit of distribution also comes with limitations. Here are the pain points:

Distributing projects composed of multiple modules

At some point, your project will outgrow a (reasonably sized) single-file module. Once you break it up into multiple files, it becomes more cumbersome for users to consume your work, and for you to publish it.

Distributing projects with third-party dependencies

Python has a rich ecosystem of third-party packages that lets you stand on the shoulders of giants. But your users should not have to worry about installing and updating the modules that your code depends on.

Discovering the project

If you publish a package on PyPI, your users only need to know your project name to install its latest version. The situation is similar in a corporate environment, where developer machines are configured to use a company-wide package repository. People can also search for your project using various metadata fields like description, keywords, or classifiers.

Installing the project

Your users should be able to install the project with a single command, in a portable and safe way. In many situations, downloading the script and double-clicking it will not (reliably) work. Users should not need to place modules in specific directories, add shebangs with the interpreter location, set the executable bit on scripts, rename scripts, or create wrapper scripts.

Updating the project

Users need to determine if the project is up-to-date and upgrade it to the latest version if it isn't. As an author, you need a way to let your users benefit from new features, bug fixes, and improvements.

Running the project in the correct environment

You should not leave it up to chance if your program runs on a supported Python version, in an environment with the necessary third-party packages. Installers should check and, where possible, satisfy your prerequisites, and ensure that your code always runs in the environment intended for it.

Binary extensions

Python modules written in a compiled language like C or Rust require a build step. Ideally, you'll distribute pre-built binaries for the common platforms. You may also publish a source archive as a fallback, with an automated build step that runs on the end user's machine during installation.

Packaging solves all of these problems, and it's quite easy to add. You drop a declarative file named *pyproject.toml* into your project, a standard file that specifies the project metadata and its build system. In return, you get commands to build, publish, install, upgrade, and uninstall your package.

In summary, Python packages come with many advantages:

- You can easily install and upgrade them with pip
- You can publish them on repositories like PyPI
- They can depend on other packages, so pip installs all of them together
- Installed packages run in an environment that satisfies their requirements
- They can contain multiple modules and import packages
- They allow you to distribute pre-built binary extensions
- They allow you to publish source distributions with automated build steps

Packaging in a Nutshell

In this section, I'll take you on a whirlwind tour of Python packaging. Example 3-2 shows how to package the script from "An Example Application" with the bare minimum of project metadata—the project name and version. Place the script and the *pyproject.toml* file side-by-side in an otherwise empty directory.

Example 3-2. A minimal pyproject.toml file

```
[project]
name = "random-wikipedia-article"
```

```
version = "0.1"

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

In the `build-system` section, I've opted for `hatchling` as the *build backend*—the tool responsible for building packages behind the scenes. It comes with Hatch, a modern and standards-compliant Python project manager.

TIP

Change the project name to a name that uniquely identifies your project. Projects on the Python Package Index share a single namespace—their names are not scoped by the users or organizations owning the projects.

Installing Projects from Source

First, let's see how this file allows you to install the project locally. Open a terminal and change to the project directory. Next, create and activate a virtual environment (see "Virtual Environments").

Now you're ready to use pip to build and install your project:

```
$ py -m pip install .
```

```
Processing path/to/project
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: random-wi
  Building wheel for random-wikipedia-article (py
  Created wheel for random-wikipedia-article: …
  Stored in directory: …
Successfully built random-wikipedia-article
Installing collected packages: random-wikipedia-a
Successfully installed random-wikipedia-article-(
```

You can run the script by passing its import name to the `-m` interpreter option:

```
$ py -m random_wikipedia_article
```

Invoking the script directly only takes a line in the `project.scripts` section. Example 3-3 tells the installer to generate an entry-point script named like the project. The script invokes the `main` function from the Python module.

Example 3-3. A *pyproject.toml* file with an entry-point script

```
[project]
```

```
name = "random-wikipedia-article"
version = "0.1"

[project.scripts]
random-wikipedia-article = "random_wikipedia_arti

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

Let’s use pipx to streamline the process of installing the project into a virtual environment and placing the script on your `PATH`. (If you activated a virtual environment above, don’t forget to first deactivate it.)

```
$ pipx install .
  installed package random-wikipedia-article 0.1,
  These apps are now globally available
    - random-wikipedia-article
done!
```

You can now invoke the script directly:

```
$ random-wikipedia-article
```

When you’re making changes to the source code, it saves time to see those changes reflected in the environment immediately, without repeatedly installing the project. You could import your modules directly from the source tree during development. Unfortunately, you’d no longer have the installer check the requirements of your project, nor would you be able to access project metadata at runtime.

Editable installs achieve the best of both worlds by installing your package in a special way that redirects imports to the source tree. You can think of this mechanism as a kind of “hot reloading” for Python packages. It works with both pip and pipx:

```
$ py -m pip install --editable .
$ pipx install --editable .
```

Once you’ve installed your package in this way, you won’t need to reinstall it to see changes to the source code—only when you edit *pyproject.toml* to change the project metadata or add a third-party dependency.

Building Packages with build

If you want to deploy your code to other machines or share your project with the world, you’ll need to grab hold of the package instead of letting pip build it behind the scenes. Enter `build`, a dedicated build frontend that creates packages for a Python project:

```
$ pipx run build
* Creating venv isolated environment...
* Installing packages in isolated environment...
* Getting build dependencies for sdist...
* Building sdist...
* Building wheel from sdist
* Creating venv isolated environment...
* Installing packages in isolated environment...
* Getting build dependencies for wheel...
* Building wheel...
Successfully built random_wikipedia_article-0.1.t
 and random_wikipedia_article-0.1-py2.py3-none-ar
```

In Example 3-2, you designated `hatchling` as the build backend for your project. You can see from the output above that `build` used `hatchling` to perform the actual package build. In "Build Frontends and Build Backends", I'll explain in more detail how the two tools interact to produce packaging artifacts.

By delegating the work to `hatchling`, `build` creates an sdist and a wheel for the project (see "Wheels and Sdists"). It then places these packages in the *dist* directory.

Uploading Packages with Twine

Let's conclude this little tour of Python packaging by publishing the packages to TestPyPI, a separate instance of the Python Package Index intended for testing and experimentation.

First, register an account using the link on the front page of TestPyPI. Second, create an API token from your account page and copy the token to your preferred password manager. You can now upload the packages in *dist* using Twine, the official PyPI upload tool. Use `__token__` as the user name and the API token as the password.

```
$ pipx run twine upload --repository=testpypi di
Uploading distributions to https://test.pypi.org/
Enter your username: __token__
Enter your password: *********
Uploading random_wikipedia_article-0.1-py2.py3-no
Uploading random_wikipedia_article-0.1.tar.gz

View at:
https://test.pypi.org/project/random-wikipedia-ar
```

Congratulations, you have published your first Python package! Let's install the package again, this time from the index instead of the project directory:

```
$ pipx uninstall random-wikipedia-article
uninstalled random-wikipedia-article!

$ pipx install --index-url=https://test.pypi.org
  installed package random-wikipedia-article 0.1
  These apps are now globally available
    - random-wikipedia-article
done!
```

Just omit the `--repository` and `--index-url` options to use the real PyPI.

DO ONE THING, AND DO IT WELL

You may wonder why the Python community decided to split up responsibilities between several packaging tools. After all, many modern programming languages come with a single monolithic tool for building and packaging.

The answer has to do with the nature and history of the Python project: Python is a decentralized open-source project driven by a community of thousands of volunteers, with a history spanning more than three decades of organic growth. This makes it hard for a single packaging tool to cater to all demands and become firmly established. Python's strength lies in its rich ecosystem—and interoperability standards promote this diversity. As a Python developer, you have a choice of small single-purpose tools that play well together. This approach ties in with the UNIX philosophy of "Do one thing, and do it well." However, there are also tools that provide a more integrated workflow; we'll talk more about those in Chapter 5.

The pyproject.toml File

Python's project specification file uses *TOML* (Tom's Obvious Minimal Language), a cross-language format for configuration files that's both unambiguous and easy to read and write. The TOML website has an excellent introduction to the format. Its intuitive syntax will look familiar to anyone with a Python background. Lists are termed *arrays* in TOML and use the same notation as Python:

```toml
requires = ["hatchling", "hatch-vcs"]
```

Dictionaries are known as *tables* and come in several equivalent forms. You can put the key/value pairs on separate lines, preceded by the table name in square brackets:

```toml
[project]
name = "foo"
version = "0.1"
```

Inline tables contain all key/value pairs on the same line:

```toml
project = { name = "foo", version = "0.1" }
```

You can also use dotted notation to create a table implicitly:

```toml
project.name = "foo"
project.version = "0.1"
```

Python implementations like the standard `tomllib` module represent a TOML file as a dictionary, where keys are strings and values can be strings, integers, floats, dates, times, lists, or dictionaries. Here's what the *pyproject.toml* file from Example 3-2 looks like in Python:

```
{
  "project": {
    "name": "random-wikipedia-article",
    "version": "0.1"
  },
  "build-system": {
    "requires": ["hatchling"],
    "build-backend": "hatchling.build"
  }
}
```

A *pyproject.toml* file contains up to three tables:

`build-system`

Specifies how to build packages for the project (see "Build Frontends and Build Backends").

`project`

Holds the project metadata (see "Project Metadata").

`tool`

Stores configuration for each tool used by the project. For example, the Black code formatter uses `tool.black` for its configuration.

Build Frontends and Build Backends

Pip and `build` don't know how to assemble packaging artifacts from source trees. They delegate that work to the tool you declare in the `build-system` table (see Table 3-1). In this relationship, pip and `build` take the role of *build frontends*, the tools an end-user invokes to orchestrate the build process. The tool that does the actual building is known as the *build backend*.

Table 3-1. The `build-system` table

Field	Type	Description
`requires`	array of strings	The list of packages required to build the project
`build-backend`	string	The import name of the build backend in the format `package.module:object`
`build-path`	string	An entry for `sys.path` needed to import the build backend (optional)

Figure 3-2 shows how the build frontend and build backend collaborate to build a package. First, the build frontend creates a virtual environment, the *build environment*. Second, it installs the *build dependencies* into this environment—the packages listed under `requires`, which consist of the build backend itself as well as, optionally, plugins for that backend. Third, the build frontend triggers the actual package build by importing and invoking the *build backend interface*. This is a module or object declared in `build-backend`, which contains a number of functions with well-known signatures for creating packages and related tasks.

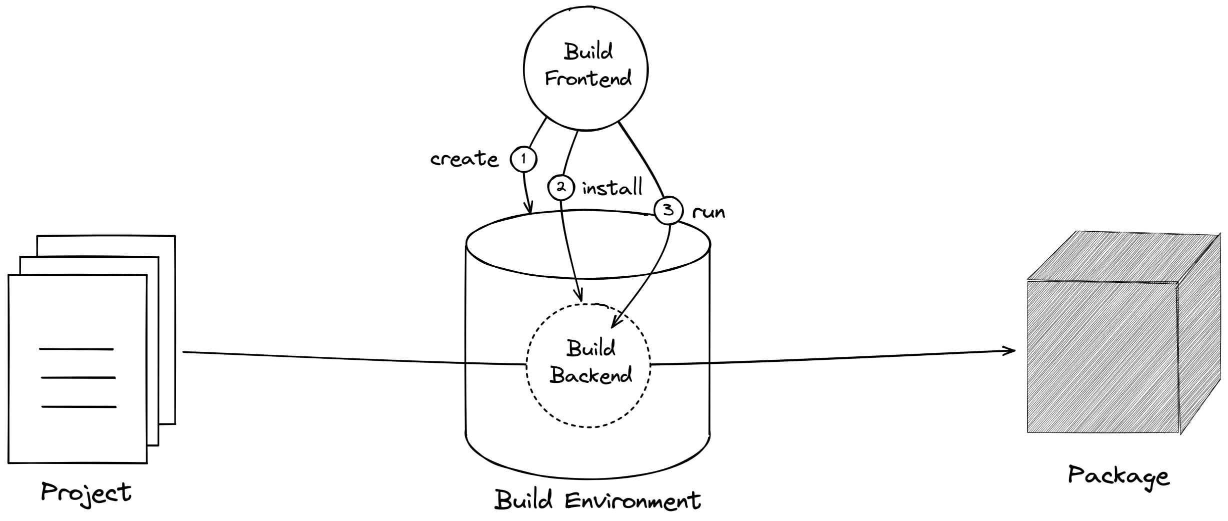

Figure 3-2. Build Frontend and Build Backend

Under the hood, pip performs the equivalent of the following commands when you install the project from its source directory:

```
$ py -m venv buildenv
$ buildenv/bin/python -m pip install hatchling
$ buildenv/bin/python
>>> import hatchling.build
>>> hatchling.build.build_wheel("dist")
'random_wikipedia_article-0.1-py2.py3-none-any.wh
>>>
$ py -m pip install dist/*.whl
```

While building in an isolated environment is the norm, some build frontends also give you the option to build in the current environment. In this case, the frontend checks that the environment satisfies the build dependencies. Build frontends never install build dependencies into your current environment. If they did, the build dependencies of different packages would be at risk of conflicting with each other as well as with their runtime dependencies.

Each build frontend can talk to any of a plethora of build backends—from traditional ones like `setuptools` to modern tools like Hatch and Poetry, or even exotic builders like `maturin` (for Python modules written in the Rust programming language) or `sphinx-theme-builder` (for Sphinx documentation themes).

Wheels and Sdists

You may have noticed that `build` placed not one but *two* packages for your project in the *dist* directory when you invoked it in "Packaging in a Nutshell":

- *random_wikipedia_article-0.1.tar.gz*
- *random_wikipedia_article-0.1-py2.py3-none-any.whl*

These artifacts are known as *wheels* and *sdists*. Wheels are ZIP archives with a *.whl* extension, while sdists are tar archives with gzip compression (*.tar.gz*). Wheels are *built distributions*—for the most part, installers simply extract them into the environment. Sdists, by contrast, are *source distributions*: they require an additional build step to produce an installable wheel.

The distinction between source distributions and built distributions may seem strange for an interpreted language, but remember that Python modules can also be written in a compiled language, for performance or to provide Python bindings for an existing library. In this case, source distributions provide a useful fallback for platforms where no pre-built wheels are available.

As a package author, you should build and publish both sdists and wheels for your releases. This gives users a choice (see Figure 3-3): They can download and install the wheel if their environment is compatible (which is always the case for a pure Python package). Or they can download the sdist and build and install a wheel from it locally.

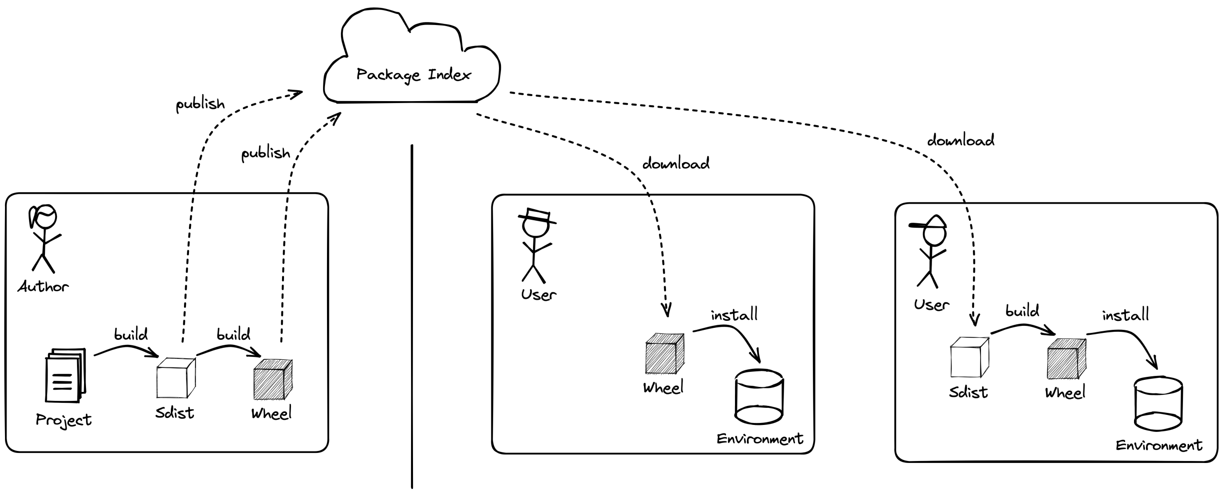

Figure 3-3. Wheels and Sdists

The `build` tool first creates an sdist from the project, and then uses that to create a wheel. Generally, a pure Python package has a single sdist and a single wheel for a given release. Binary extension modules, on the other hand, commonly come in wheels for a range of platforms and environments.

WHEEL COMPATIBILITY TAGS

Installers select the appropriate wheel for an environment using three so-called *compatibility tags* that are embedded in the name of each wheel file:

Python tag

> The target Python implementation

ABI tag

> The target *application binary interface* (ABI) of Python, which defines the set of symbols that binary extension modules can use to interact with the interpreter

Platform tag

> The target platform, including the processor architecture

Pure Python wheels are usually compatible with any Python implementation, do not require a particular ABI, and are portable across platforms. Wheels express such wide compatibility using the tags `py3-none-any`.

Wheels with binary extension modules, on the other hand, have more stringent compatibility requirements. Take a look at the compatibility tags of these wheels, for example:

- *numpy-1.24.0-cp311-cp311-macosx_10_9_x86_64.whl*
- *cryptography-38.0.4-cp36-abi3-manylinux_2_28_x86_64.whl*

The wheel for NumPy—a fundamental library for scientific computing—targets a specific Python implementation and version (CPython 3.11), operating system release (macOS 10.9 and above), and processor architecture (x86-64).

The wheel for Cryptography—another fundamental library, with an interface to cryptographic algorithms—demonstrates two ways to reduce the build matrix for binary distributions: The *stable ABI* is a restricted set of symbols that are guaranteed to persist across Python feature versions (`abi3`), and the `manylinux` tag advertises compatibility with a particular C standard library implementation (glibc 2.28 and above) across a wide range of Linux distributions.

Let's peek inside a wheel to a get a feeling for how Python code is distributed. You can extract wheels using the `unzip` utility to see the files installers would place in the *site-packages* directory. Execute the following commands in a shell on Linux or macOS, preferably inside an empty directory. If you're on Windows, you can follow along using the Windows Subsystem for Linux (WSL).

```
$ py -m pip download attrs
$ unzip attrs-22.2.0-py3-none-any.whl
$ ls -1
attr
attrs
attrs-22.2.0.dist-info
attrs-22.2.0-py3-none-any.whl

$ head -5 attrs-22.2.0.dist-info/METADATA
Metadata-Version: 2.1
Name: attrs
Version: 22.2.0
Summary: Classes Without Boilerplate
Home-page: https://www.attrs.org/
```

In our example, the wheel contains two import packages named `attr` and `attrs`, as well as a *.dist-info* directory with administrative files. The *METADATA* file contains the *core metadata* for the package, a standardized set of attributes that describe the package for the benefit of installers and other packaging tools. You can access the core metadata of installed packages at runtime using the standard library:

```
>>> from importlib.metadata import metadata
>>> metadata("attrs")["Version"]
22.2.0
```

```
>>> metadata("attrs")["Summary"]
Classes Without Boilerplate
```

In the next section, you’ll see how to embed core metadata in your own packages.

Project Metadata

Build backends write out core metadata fields based on what you specify in the `project` table of *pyproject.toml*. Table 3-2 provides an overview of all the fields you can use in the `project` table.

Table 3-2. The `project` table

Field	Type	Description
`name`	string	The project name
`version`	string	The version of the project
`description`	string	A short description of the project
`keywords`	array of strings	A list of keywords for the project
`readme`	string or table	A file with a long description of the project
`license`	table	The license governing the use of this project
`authors`	array of tables	The list of authors
`maintainers`	array of tables	The list of maintainers
`classifiers`	array of strings	A list of classifiers describing the project

Field	Type	Description
`urls`	table of strings	The project URLs
`dependencies`	array of strings	The list of required third-party packages
`optional-dependencies`	table of arrays of strings	Named lists of optional third-party packages (*extras*)
`scripts`	table of strings	Entry-point scripts
`gui-scripts`	table of strings	Entry-point scripts providing a graphical user interface
`entry-points`	table of tables of strings	Entry point groups
`requires-python`	string	The Python version required by this project
`dynamic`	array of strings	A list of dynamic fields

Two fields are essential and mandatory for every package: `project.name` and `project.version`. The project name uniquely identifies the project itself. The project version identifies a *release*—a published snapshot of the project during its lifetime. Besides the name and version, there are a number of optional fields you can provide, such as the author and license, a short text describing the project, or third-party packages used by the project (see Example 3-4).

Example 3-4. A pyproject.toml file with project metadata

```
[project]
name = "random-wikipedia-article"
version = "0.1"
description = "Display extracts from random Wikip
keywords = ["wikipedia"]
readme = "README.md"
license = { text = "MIT" }
authors = [{ name = "Your Name", email = "you@exa
classifiers = ["Topic :: Games/Entertainment :: F
urls = { Homepage = "https://yourname.dev/project
requires-python = ">=3.7"
dependencies = ["httpx>=0.23.1", "rich>=12.6.0"]
```

In the following sections, I'll take a closer look at the various project metadata fields.

NOTE

Most project metadata fields correspond to a core metadata field (and sometimes two). However, their names and syntax differ slightly—core metadata standards predate *pyproject.toml* by many years. As a package author, you can safely ignore the details of this translation and focus on the project metadata.

Naming Projects

The `project.name` field contains the official name of your project.

```
[project]
name = "random-wikipedia-article"
```

Your users specify this name to install the project with pip. This field also determines your project's URL on PyPI. You can use any ASCII letter or digit to name your project, interspersed with periods, underscores, and hyphens. Packaging tools normalize project names for comparison: all letters are converted to lowercase, and punctuation runs are replaced by a single hyphen (or underscore, in the case of package filenames). For example, `Awesome.Package`, `awesome_package`, and `awesome-package` all refer to the same project.

Project names are distinct from *import names*, the names users specify to import your code. The latter must be valid Python identifiers, so they can’t have hyphens or periods and can’t start with a digit. They’re case-sensitive and can contain any Unicode letter or digit. As a rule of thumb, you should have a single import package per distribution package and use the same name for both (or a straightforward translation, like `random-wikipedia-article` and `random_wikipedia_article`).

Versioning Projects

The `project.version` field stores the version of your project at the time you publish the release.

```toml
[project]
version = "0.1"
```

The Python community has a specification for version numbers to ensure that automated tools can make meaningful decisions, such as picking the latest release of a project. At the core, versions are a dotted sequence of numbers. These numbers may be zero, and trailing zeros can be omitted: `1`, `1.0`, and `1.0.0` all refer to the same version. Additionally, you can append certain kinds of suffixes to a version (see Table 3-3). The most common ones identify pre-releases: `1.0.0a2` is the second alpha release, `1.0.0b3` is the third beta release, `1.0.0rc1` is the first release candidate. Each of these precedes the next, and all of them precede the final release: `1.0.0`.

Python versions can use additional components as well as alternate spellings; refer to PEP 440 for the full specification.

Table 3-3. Version Identifiers

Release Type	Description	Examples
Final release	A stable, public snapshot (default)	`1.0.0`, `2017.5.25`
Pre-release	Preview of a final release to support testing	`1.0.0a1`, `1.0.0b1`, `1.0.0rc1`
Developmental release	A regular internal snapshot, such as a nightly build	`1.0.0.dev1`
Post-release	Corrects a minor error outside of the code	`1.0.0.post1`

The Python version specification is intentionally permissive. Two widely adopted cross-language standards attach additional meaning to version numbers: Semantic Versioning uses the scheme `major.minor.patch`, where `patch` designates bugfix releases, `minor` designates compatible feature releases, and `major` designates releases with breaking changes. Calendar Versioning uses date-based versions of various forms, such as `year.month.day`, `year.month.sequence`, or `year.quarter.sequence`.

Single-Sourcing the Project Version

Normally, you must declare all of the metadata for your project verbatim in the *pyproject.toml* file. But sometimes you want to leave a field unspecified and let the build backend fill in the value during the package build. For example, you may want to derive your package version from a Python module or Git tag instead of duplicating it in the `project` table.

Luckily, the project metadata standard provides an escape hatch in the form of *dynamic fields*. Projects are allowed to use a backend-specific mechanism to compute a field on the fly, as long as they list its name under the `dynamic` key.

```toml
[project]
dynamic = ["version", "readme"]
```

NOTE

The goal of the standards behind *pyproject.toml* is to let projects define their metadata statically, rather than rely on the build backend to compute the fields during the package build. This benefits the packaging ecosystem, because it makes metadata accessible to other tools. It also reduces cognitive overhead because build backends share a unified configuration format and populate the metadata fields in a straightforward and transparent way.

Dynamic fields are a popular method for single-sourcing the project version. For example, many projects declare their version at the top of a Python module, like this:

```
__version__ = "0.2"
```

Because updating a frequently changing item in several locations is tedious and error-prone, some build backends allow you to extract the version number from the code instead of duplicating it in *pyproject.toml*. This mechanism is specific to your build backend, so you configure it in the `tool` table of your backend. Example 3-5 demonstrates how this works with Hatch.

Example 3-5. Deriving the project version from a Python module

```
[project]
name = "random-wikipedia-article"
dynamic = ["version"] ❶

[tool.hatch.version]
path = "random_wikipedia_article.py" ❷

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

❶ This line marks the version field as dynamic.

❷ This line tells Hatch where to look for the `__version__` attribute.

The astute reader will have noticed that you don’t really need this mechanism to avoid duplicating the version. You can also declare the version in *pyproject.toml* as usual and read it from the installed metadata at runtime:

Example 3-6. Reading the version from the installed metadata

```
from importlib.metadata import version

__version__ = version("random-wikipedia-article"
```

But don’t go and add this boilerplate to all your projects yet. Reading the metadata from disk is not something you want to do during program startup. Third-party libraries like `click` provide mature implementations that perform the metadata lookup on demand, under the hood—for example, when the user specifies a command-line option like `--version`.

Unfortunately, this is usually not enough to truly single-source the version. It’s considered good practice to tag releases in your version control system (VCS) using a command like `git tag v1.0.0`. Luckily, a number of build backends come with plugins that extract the version number from Git, Mercurial, and similar systems. This technique was pioneered by the `setuptools-scm` plugin; for Hatch, you can use the `hatch-vcs` plugin (see Example 3-7).

Example 3-7. Deriving the project version from the version control system

```
[project]
name = "random-wikipedia-article"
dynamic = ["version"]

[tool.hatch.version]
source = "vcs"

[build-system]
requires = ["hatchling", "hatch-vcs"]
build-backend = "hatchling.build"
```

If you build this project from a repository and you've checked out the tag `v1.0.0`, Hatch will use the version `1.0.0` for the metadata. If you've checked out an untagged commit, Hatch will instead generate a developmental release like `0.1.dev1+g6b80314`.[2]

Entry-point Scripts

Entry-point scripts are small executables that launch the interpreter from their environment, import a module and invoke a function (see “Entry-point Scripts”). Installers like pip generate them on the fly when they install a package.

The `project.scripts` table lets you declare entry-point scripts. Specify the name of the script as the key and the module and function that the script should invoke as the value, using the format `module:function`.

```
[project.scripts]
random-wikipedia-article = "random_wikipedia_art
```

This declaration allows users to invoke the program using its given name:

```
$ random-wikipedia-article
```

The `project.gui-scripts` table uses the same format as the `project.scripts` table—use it if your application has a graphical user interface (GUI).

```
[project.gui-scripts]
random-wikipedia-article-gui = "random_wikipedia_
```

Entry Points

Entry-point scripts are a special case of a more general mechanism called *entry points*. Entry points allow you to register a Python object in your package under a public name. Python environments come with a registry of entry points, and any package can query this registry to discover and import modules, using the function `importlib.metadata.entry_points` from the standard library. Applications commonly use this mechanism to support third-party plugins.

The `project.entry-points` table contains these generic entry points. They use the same syntax as entry-point scripts, but are grouped in subtables known as *entry point groups*. If you want to write a plugin for another application, you register a module or object in its designated entry point group.

```
[project.entry-points.some_application]
my-plugin = "my_plugin"
```

You can also register submodules using dotted notation, as well as objects within modules, using the format `module:object`:

```
[project.entry-points.some_application]
my-plugin = "my_plugin.submodule:plugin"
```

Let’s look at an example to see how this works. Random Wikipedia articles make for fun little fortune cookies, but they can also serve as *test fixtures*[3] for developers of Wikipedia viewers and similar apps. Let’s turn the app into a plugin for the Pytest testing framework. (Don’t worry if you haven’t worked with Pytest yet; I’ll cover testing in depth in Chapter 6.)

Pytest allows third-party plugins to extend its functionality with test fixtures and other features. It defines an entry point group for such plugins named `pytest11`. You can provide a plugin for Pytest by registering a module in this group. Let’s also add Pytest to the project dependencies.

```
[project]
dependencies = ["pytest"]

[project.entry-points.pytest11]
random-wikipedia-article = "random_wikipedia_art
```

For simplicity, I’ve chosen the top-level module that hosted the `main` function in Example 3-1. Next, extend Pytest with a test fixture returning a random Wikipedia article, as shown in Example 3-8.

Example 3-8. Test fixture with a random Wikipedia article

```
import json
import urllib.request

import pytest

API_URL = "https://en.wikipedia.org/api/rest_v1/p

@pytest.fixture
def random_wikipedia_article():
    with urllib.request.urlopen(API_URL) as respo
        return json.load(response)
```

A developer of a Wikipedia viewer can now install your plugin next to Pytest. Test functions use your test fixture by referencing it as a function argument (see Example 3-9). Pytest recognizes that the function argument is a test fixture and invokes the test function with the return value of the fixture.

Example 3-9. A test function that uses the random article fixture

```
# test_wikipedia_viewer.py
```

```
def test_wikipedia_viewer(random_wikipedia_articl
    print(random_wikipedia_article["title"]) ❶
    print(random_wikipedia_article["extract"])
    assert False ❷
```

❶ A real test would run the viewer instead of `print()`.

❷ Fail the test so we get to see the full output.

You can try this out yourself in an active virtual environment in the project directory:

```
$ py -m pip install .
$ py -m pytest test_wikipedia_viewer.py
============================= test session starts
platform darwin -- Python 3.11.1, pytest-7.2.1, p
rootdir: ...
plugins: random-wikipedia-article-0.1
collected 1 item

test_wikipedia_viewer.py F

=================================== FAILURES ====
____________________________ test_wikipedia_viewe

    def test_wikipedia_viewer(random_wikipedia_ar
        print(random_wikipedia_article["title"])
        print(random_wikipedia_article["extract"]
>       assert False
E       assert False

test_wikipedia_viewer.py:4: AssertionError
----------------------------- Captured stdout ca
Halgerda stricklandi
Halgerda stricklandi is a species of sea slug, a
marine gastropod mollusk in the family Discodori
=========================== short test summary i
FAILED test_wikipedia_viewer.py::test_wikipedia_
============================== 1 failed in 1.10s
```

Authors and Maintainers

The `project.authors` and `project.maintainers` fields contain the list of authors and maintainers for the project. Each item in these lists is a table with `name` and `email` keys—you can specify either of these keys or both.

```
[project]
authors = [{ name = "Your Name", email = "you@exa
maintainers = [
  { name = "Alice", email = "alice@example.com" 
  { name = "Bob", email = "bob@example.com" },
]
```

The meaning of the fields is somewhat open to interpretation. If you start a new project, I recommend including yourself under `authors` and omitting the `maintainers` field. Long-lived open-source projects typically list the original author under `authors`, while the people in charge of ongoing project maintenance appear as `maintainers`.

The Description and README

The `project.description` field contains a short description as a string. This field will appear as the subtitle of your project page on PyPI. Some packaging tools also use this field when displaying a compact list of packages with human-readable descriptions.

```
[project]
description = "Display extracts from random Wiki
```

The `project.readme` field is typically a string with the relative path to the file with the long description of your project. Common choices are *README.md* for a description written in Markdown format and *README.rst* for the reStructuredText format. The contents of this file appear on your project page on PyPI.

```toml
[project]
readme = "README.md"
```

Instead of a string, you can also specify a table with `file` and `content-type` keys.

```toml
[project]
readme = { file = "README", content-type = "text/
```

You can even embed the long description in the *pyproject.toml* file using the `text` key.

```toml
[project]
readme.text = """
# Display extracts from random Wikipedia articles

Long description follows...
"""
readme.content-type = "text/markdown"
```

Writing a README that renders well is not trivial—often, the project description appears in disparate places, like PyPI, a repository hosting service like GitHub, and inside official documentation on services like Read the Docs. If you need more flexibility, you can declare the field dynamic and use a plugin like `hatch-fancy-pypi-readme` to assemble the project description from multiple fragments.

Keywords and Classifiers

The `project.keywords` field contains a list of strings that people can use to search for your project.

```
[project]
keywords = ["wikipedia"]
```

The `project.classifiers` field contains a list of classifiers to categorize the project in a standardized way.

```
[project]
classifiers = [
    "Development Status :: 3 - Alpha",
    "Environment :: Console",
    "Topic :: Games/Entertainment :: Fortune Cook
]
```

PyPI maintains the official registry of classifiers for Python projects. They are known as *Trove classifiers*[4] and consist of hierarchically organized labels separated by double colons (see Table 3-4).

Table 3-4. Trove Classifiers

Classifier Group	Description	Example
Development Status	How mature this release is	`Development Status :: 5 - Production/Stable`
Environment	The environment in which the project runs	`Environment :: No Input/Output (Daemon)`
Operating System	The operating systems supported by the project	`Operating System :: OS Independent`
Framework	Any framework used by the project	`Framework :: Flask`
Audience	The kind of users served by the project	`Intended Audience :: Developers`
License	The license under which the project is distributed	`License :: OSI Approved :: MIT License`

Classifier Group	Description	Example
Natural Language	The natural languages supported by the project	`Natural Language :: English`
Programming Language	The programming language the project is written in	`Programming Language :: Python :: 3.12`
Topic	Various topics related to the project	`Topic :: Utilities`

The Project URLs

The `project.urls` table allows you to point users to your project homepage, source code, documentation, issue tracker, and similar project-related URLs. Your project page on PyPI links to these pages using the provided key as the display text for each link. It also displays an appropriate icon for many common names and URLs.

```
[project.urls]
Homepage = "https://yourname.dev/projects/random-
Source = "https://github.com/yourname/random-wiki
Issues = "https://github.com/yourname/random-wiki
```

```
Documentation = "https://readthedocs.io/random-wi
```

The License

The `project.license` field is a table where you can specify your project license under the `text` key or by reference to a file under the `file` key. You may also want to add the corresponding Trove classifier for the license.

```
[project]
license = { text = "MIT" }
classifiers = ["License :: OSI Approved :: MIT Li
```

I recommend using the `text` key with a SPDX license identifier such as “MIT” or “Apache-2.0”.[5] The Software Package Data Exchange (SPDX) is an open standard backed by the Linux Foundation for communicating software bill of material information, including licenses.

If you’re unsure which open source license to use for your project, choosealicense.com provides some useful guidance. For a proprietary project, it’s common to specify “proprietary”. You can also add a special Trove classifier to prevent accidental upload to PyPI.

```toml
[project]
license = { text = "proprietary" }
classifiers = [
    "License :: Other/Proprietary License",
    "Private :: No Upload",
]
```

The Required Python Version

Use the `project.requires-python` field to specify the versions of Python that your project supports.[6]

```toml
[project]
requires-python = ">=3.7"
```

Most commonly, people specify the minimum Python version as a lower bound, using a string with the format `>=3.x`. The syntax of this field is more general and follows the same rules as *version specifiers* for project dependencies (see Chapter 4).

Tools like Nox and tox make it easy to run checks across multiple Python versions, helping you ensure that the field reflects reality. As a baseline, I recommend requiring the oldest Python version that still receives security updates. You can find the end-of-life dates for all current and past Python versions on the Python Developer Guide.

There are three main reasons to be more restrictive about the Python version. First, your code may depend on newer language features—for example, structural pattern matching was introduced in Python 3.10. Second, your code may depend on newer features in the standard library—look out for the “Changed in version 3.x” notes in the official documentation. Third, it could depend on third-party packages with more restrictive Python requirements.

Some packages declare upper bounds on the Python version, such as `>=3.7,<4`. This practice is discouraged, but depending on such a package may force you to declare the same upper bound for your own package. Dependency solvers can’t downgrade the Python version in an environment; they will either fail or, worse, downgrade the package to an old version with a looser Python constraint. A future Python 4 is unlikely to introduce the kind of breaking changes that people associate with the transition from Python 2 to 3.

WARNING

Don’t specify an upper bound for the required Python version unless you *know* that your package is not compatible with any higher version. Upper bounds cause disruption in the ecosystem when a new version is released.

Dependencies and Optional Dependencies

The remaining two fields, `project.dependencies` and `project.optional-dependencies`, list any third-party packages on which your project depends. You'll take a closer look at these fields—and dependencies in general—in the next chapter.

Summary

Packaging allows you to publish releases of your Python projects, using source distributions (*sdists*) and built distributions (*wheels*). These artifacts contain your Python modules, together with project metadata, in an archive format that end users can easily install into their environments. The standard *pyproject.toml* file defines the build system for a Python project as well as the project metadata. Build frontends like pip and `build` use the build system information to install and run the build backend in an isolated environment. The build backend assembles an sdist and wheel from the source tree and embeds the project metadata. You can upload packages to the Python Package Index (PyPI) or a private repository, using a tool like Twine.

[1] The Python Package Index (PyPI) did not come about for more than a decade. Even the venerable Comprehensive Perl Archive Network (CPAN) did not exist in February 1991, when Guido van Rossum published the first release of Python on Usenet.

In case you’re wondering, the `+g6b80314` suffix is a *local version identifier* that designates downstream changes, in this case using output from the command `git describe`.

Test fixtures set up objects that you need to run repeatable tests against your code.

The Trove project was an early attempt to provide an open-source software repository, initiated by Eric S. Raymond.

As of this writing, a Python Enhancement Proposal (PEP) is under discussion that changes the `project.license` field to a string using SPDX syntax and introduces a separate `project.license-files` key for license files that should be distributed with the package (see PEP 639).

You can also add Trove classifiers for each supported Python version. Some backends backfill classifiers for you—Poetry does this out of the box for Python versions and project licenses.

Chapter 4. Dependency Management

Python programmers benefit from a rich ecosystem of third-party libraries and tools. Standing on the shoulders of giants comes at a price: The packages you depend on for your projects generally depend on a number of packages themselves. All of these are moving targets—as long as any project is alive, its maintainers will publish a stream of releases to fix bugs, add features, and adapt to the evolving ecosystem.

Managing dependencies is one of the great pain points of maintaining software over time. You need to keep your project up-to-date, if only to close security vulnerabilities in a timely fashion. Usually this means updating your dependencies to the latest version—few open-source projects have the resources to distribute security updates for older releases. You'll be updating dependencies all the time, so making the process as frictionless, automated, and reliable as possible comes with a huge payoff.

Dependencies of a Python project are the third-party packages that must be installed in the project environment.[1] Most commonly, the reason for a dependency is that the project imports a module that comes distributed with the package. We also say that the project *requires* a package, or that a package is a *requirement* for the project.

Another reason for a dependency is that your project requires a third-party tool for developer tasks such as running the test suite or building documentation. This kind of dependency is known as a *development dependency*: you only need those packages during development, not during normal usage of your project. A related case would be the build dependencies you have seen in Chapter 3, which are required to build your own packages from a project.

This chapter explains how to manage dependencies effectively. In the next section, you'll learn how to specify dependencies in *pypro-*

ject.toml, as part of the project metadata. Afterwards, I'll talk about development dependencies and requirement files. Finally, I'll explain how you can *lock* dependencies to precise versions for reliable deployments and repeatable checks.

As a working example, let's enhance `random-wikipedia-article` from Example 3-1 with the httpx library, a fully featured HTTP client that supports both synchronous and asynchronous requests, as well as the newer (and far more efficient) protocol version HTTP/2. Let's also enhance the output of the program using rich, a library for rich text and beautiful formatting in the terminal.

Example: Consuming an API with HTTPX

Wikipedia asks developers to set a `User-Agent` header with contact details. Example 4-1 shows how you can use `httpx` to send a request to the Wikipedia API with the header. You could also use the standard library to send a `User-Agent` header with your requests. But `httpx` offers a more intuitive, explicit, and flexible interface, even when you're not using any of its advanced features. Try it out:

Example 4-1. Using `httpx` to consume the Wikipedia API

```
import textwrap
import textwrap
import httpx

API_URL = "https://en.wikipedia.org/api/rest_v1/
USER_AGENT = "random-wikipedia-article/0.1 (Cont

def main():
    headers = {"User-Agent": USER_AGENT}

    with httpx.Client(headers=headers) as client
        response = client.get(API_URL, follow_re
        response.raise_for_status() ❸
        data = response.json() ❹

    print(data["title"])
    print()
    print(textwrap.fill(data["extract"]))
```

❶ When creating a client instance, you can specify headers that it should send with every request—like the `User-Agent` header. Using the client as a context manager ensures that the network connection is closed at the end of the `with` block.

❷ This line performs two HTTP `GET` requests to the API. The first one goes to the *random* endpoint, which responds with a redirect to the actual article. The second one follows the redirect.

❸ The `raise_for_status` method raises an exception if the server response indicates an error via its status code.

❹ The `json` method abstracts the details of parsing the response body as JSON.

Example: Console Output with Rich

While we're at it, let's also improve the look and feel of the program. Example 4-2 uses `rich`, a library for console output, to display the article title in bold. This hardly scrapes the surface of `rich`'s formatting options—take a look at the official docs for more details.

Example 4-2. Using `rich` to enhance console output

```
import httpx
from rich.console import Console

def main():
    ...

    console = Console(width=72, highlight=False)

    console.print(data["title"], style="bold") ❷

    console.print()
    console.print(data["extract"])
```

1. Console objects provide a feature-rich `print` method for console output. Setting the console width to 72 characters replaces our earlier call to `textwrap.fill`. You'll also want to disable automatic syntax highlighting, since you're formatting prose rather than data or code.

2. The `style` keyword allows you to set the title apart using a bold font.

Specifying Dependencies for a Project

If you haven't done so yet, create and activate a virtual environment for the project, and perform an editable install from the current directory (`.`):

```
$ py -m venv .venv
$ py -m pip install --editable .
```

At this point, you may be tempted to install `httpx` and `rich` manually into the environment. Instead, add these packages to the project dependencies in *pyproject.toml*. This ensures that whenever users install your project into an environment, they install `httpx` and `rich` along with it.

```
[project]
name = "random-wikipedia-article"
version = "0.1"
dependencies = ["httpx", "rich"]
...
```

If you reinstall the project, you'll see that pip installs the dependencies as well:

```
$ py -m pip install --editable .
```

Dependency specifications allow you to include some other pieces of information about a package, besides package names: version specifiers, extras, and environment markers. The following sections explain what these are. The first, and arguably the most important, specifies with which versions of the package your project is compatible.

Version Specifiers

Version specifiers define the range of acceptable versions for a package. When you add a new dependency, it's a good idea to include its current version as a lower bound—unless you know your project is compatible with older releases (and needs to support them). Update this lower bound whenever you start relying on newer features of the package.

```
[project]
dependencies = ["httpx>=0.23.1", "rich>=12.6.0"]
```

Having lower bounds on your dependencies may not seem to matter much, as long as you install your package in an isolated environment. Installers will choose the latest version for all your dependencies if there are no constraints from other packages. But there are three reasons why you should care. First, libraries are typically installed alongside other packages. Second, even applications are not always installed in isolation; for example, Linux distros may want to package your application for the system-wide environment. Third, lower bounds help you detect version conflicts in your own dependency tree early. This can happen if you depend on a recent release of some package, while one of your other dependencies only works with older releases of that package.

What about *upper* bounds—should you guard against newer releases that might break your project? I recommend avoiding upper bounds unless you know your project is incompatible with the new version of a dependency (see “Upper Version Bounds in Python”). Later in this chapter, I’ll talk about *lock files*. These request “known good” versions of your dependencies when deploying services and when running automated checks. Lock files are a much better solution to dependency-induced breakage than upper bounds.

If a botched release breaks your project, publish a bugfix release to exclude that specific broken version:

```toml
[project]
dependencies = ["awesome>=1.2,!=1.3.1"]
```

If a dependency breaks compatibility permanently, use an upper bound as a last resort until you’re able to adapt to the incompatible changes:

```toml
[project]
dependencies = ["awesome>=1.2,<2"]
```

WARNING

Excluding versions after the fact has a pitfall that you need to be aware of. Dependency resolvers can decide to downgrade your project to a version without the exclusion, and upgrade the dependency anyway.

UPPER VERSION BOUNDS IN PYTHON

Some people routinely include upper bounds in version constraints, especially for dependencies that follow Semantic Versioning. In this widely adopted versioning scheme, major versions signal any *breaking changes*, or incompatible changes to the public API. As engineers, we err on the side of safety to build robust products, so at first glance, guarding against major releases seems like something any responsible person would do. Even if most of them don't break your project, isn't it better to opt in after you have a chance to test the release?

Unfortunately, upper version bounds quickly lead to unsolvable dependency conflicts. Python environments (unlike Node.js environments, in particular) can only contain a single version of each package. Libraries that put upper bounds on their own dependencies also prevent downstream projects from receiving security and bug fixes for those packages. Before adding an upper version bound, I therefore recommend that you carefully consider the costs and benefits.

What exactly constitutes a breaking change is less defined than it may seem. For example, should a project increment its major version every time it drops support for an old Python version? Even in clear cases, a breaking change will only break your project if it affects the part of the public API that your project actually uses. By contrast, many changes that will break your project are not marked by a version number (they're simply bugs). In the end, you'll still rely on automated tests to discover "bad" versions and deal with them after the fact.

Version specifiers support several operators, as shown in Table 4-1. You can use the equality and comparison operators you know from Python: `==`, `!=`, `<=`, `>=`, `<`, and `>`. The `==` operator also supports wildcards (`*`), albeit only at the end of the version string; in other words, you can require the version to match a particular prefix, such as `1.2.*`. There's also a `===` operator to perform a simple character-by-character comparison, which is best used as a last resort for non-standard versions. Finally, the compatible release operator `~=` specifies that the version should be greater than or equal to the given value, while still starting with the same prefix. For example, `~=1.2.3` is equivalent to `>=1.2.3,==1.2.*`, and `~=1.2` is equivalent to `>=1.2,==1.*`.

Table 4-1. Version Specifiers

Operator	Name	Description
`==`	Version matching clause	Versions must compare equal after normalization, such as stripping off trailing zeros.
`!=`	Version exclusion clause	The inverse of the `==` operator
`<=`, `>=`	Inclusive ordered comparison clause	Performs lexicographical comparison, with pre-releases preceding final releases
`<`, `>`	Exclusive ordered comparison clause	Similar as above, but the versions must not compare equal
`~=`	Compatible release clause	Equivalent to `>=x.y,==x.*` to the specified precision

Operator	Name	Description
`===`	Arbitrary equality clause	Simple string comparison for non-standard versions

You don’t need upper bounds to exclude pre-releases, by the way. Even when pre-releases are newer than the latest stable release, version specifiers exclude them by default due to their expected instability. There are only three situations where pre-releases are valid candidates: when they’re already installed, when they’re the only ones satisfying the dependency specification, and when you request them explicitly, using a clause like `>=1.0.0rc1`.

Extras and Optional Dependencies

Suppose you want to use the newer HTTP/2 protocol with `httpx`. This only requires a small change to the code that creates the HTTP client:

```
def main():
    headers = {"User-Agent": USER_AGENT}
    with httpx.Client(headers=headers, http2=True
        ...
```

Under the hood, `httpx` delegates the gory details of speaking HTTP/2 to another package (named `h2`). That dependency is not pulled in by default, however. This way, users who don't need the newer protocol get away with a smaller dependency tree. You do need it here, so activate the optional feature using the syntax `httpx[http2]`:

```
[project]
dependencies = ["httpx[http2]>=0.23.1", "rich>=12
```

Optional features that require additional dependencies are known as *extras*, and you can have more than one. For example, you could specify `httpx[http2,brotli]` to allow decoding responses with *Brotli compression*, which is a compression algorithm developed at Google and commonly used in web servers and content delivery networks.

Let's take a look at this situation from the point of view of `httpx`. The `h2` and `brotli` dependencies are optional, so `httpx` does not declare them under `project.dependencies`, but under `project.optional-dependencies`. Here's what that looks like:

Example 4-3. Optional dependencies of `httpx` (simplified)

```
[project]
name = "httpx"

[project.optional-dependencies]
http2 = ["h2>=3,<5"]
brotli = ["brotli"]
```

The `optional-dependencies` field is a TOML table and can hold multiple lists of dependencies, one for each extra provided by the package. Otherwise, optional dependencies are specified in the same way as normal dependencies, including version constraints.

If you add an optional dependency to your project, how do you use that dependency in the code—should you check if your package was installed with the extra? The most common technique is to simply import the optional package—but catch the `ImportError` exception in case the user did not activate the extra:

```
try:
    import h2
except ImportError:
    h2 = None

# Check h2 before use.
if h2 is not None:
    ...
```

This pattern is so common in Python, even beyond imports, that it has a name: “Easier to Ask Forgiveness than Permission” (EAFP).[2]

Environment Markers

The third piece of metadata you can specify for a dependency is environment markers. But before I explain what these are, let me show you an example of where they come in handy.

If you looked at the `User-Agent` header in Example 4-1 and thought, “I should not have to repeat the version number in the code”, you’re absolutely right. As you’ve seen in “Single-Sourcing the Project Version”, you can read the version of your package from its metadata in the environment. Example 4-4 shows how you can use the function `importlib.metadata.metadata` to construct the `User-Agent` header from the `Name`, `Version`, and `Author-email` core metadata fields. These fields correspond to the `name`, `version`, and `authors` fields in the project metadata.[3]

Example 4-4. Using `importlib.metadata` to build a `User-Agent` header

```
from importlib.metadata import metadata
```

```
USER_AGENT = "{Name}/{Version} (Contact: {Author-

def build_user_agent():
    fields = metadata("random-wikipedia-article"
    return USER_AGENT.format_map(fields) ❷

def main():
    headers = {"User-Agent": build_user_agent()}
    ...
```

❶ The `metadata` function retrieves the core metadata fields for the package.

❷ The `str.format_map` function replaces each placeholder using a lookup in the provided mapping (the core metadata fields).

The `importlib.metadata` library was introduced in Python 3.8. Does this mean you're out of luck if your project needs to support an older Python version? Fortunately not. Many additions to the standard library come with third-party *backports* that provide the same interface to legacy environments. In the case of `importlib.metadata`, you can fall back to the `importlib-metadata` backport from PyPI.

You only need backports in environments that use specific Python versions. *Environment markers* allow you to communicate this fact in your dependency specification: they're conditions that installers evaluate on the interpreter of the target environment. Thanks to this mechanism, you can ask for a dependency to be installed only on specific operating systems, processor architectures, Python implementations, or Python versions. See Table 4-2 for the full list of environment markers at your disposal.

Going back to our example, you can use the `python_version` marker to require `importlib-metadata` only in environments with a Python version older than 3.8:

```toml
[project]
dependencies = [
    "httpx[http2]>=0.23.1",

    "rich>=12.6.0",
    "importlib-metadata>=5.2.0; python_version <
]
```

The import name for the backport is `importlib_metadata`, while the standard library module is named `importlib.metadata`. Use the EAFP technique from "Extras and Optional Dependencies" to import the appropriate module in your code:

```
try:
    from importlib.metadata import metadata
except ImportError:
    from importlib_metadata import metadata
```

Like extras, environment markers gate the dependency behind a condition. The difference is that extras are triggered by users, while markers are triggered by the target environment. In both cases, you need to account for the possibility that importing the package may fail.

Markers support the same equality and comparison operators as version specifiers (see Table 4-1). Additionally, you can use `in` and `not in` to match a substring against the marker. You can also combine multiple markers using the boolean operators `and` and `or`. Here's a contrived example that combines all of these features:

```
[project]
dependencies = ["""
  awesome-package; python_full_version <= '3.8.1
    and (implementation_name == 'cpython' or impl
    and sys_platform == 'darwin'
    and 'arm' in platform_version
"""]
```

I’ve also relied on TOML’s support for multi-line strings here, which uses triple quotes just like Python. Dependency specifications cannot span multiple lines, so you have to escape any newlines with a backslash.

Table 4-2. Environment Markers

Environment Marker	Standard Library	Description	Examples
`os_name`	`os.name()`	The operating system family	`posix`, `nt`
`sys_platform`	`sys.platform()`	The platform identifier	`linux`, `darwin`, `win32`
`platform_system`	`platform.system()`	The system name	Linux, Darwin, Windows
`platform_release`	`platform.release()`	The operating system release	`22.1.0`
`platform_version`	`platform.version()`	The system release	`Darwin Kernel Version 22.1.0: ...`
`platform_machine`	`platform.machine()`	The processor architecture	`x86_64`, `arm64`
`python_version`	`platform.python_version_tuple()`	The Python feature version in the format `x.y`	`3.11`

Environment Marker	Standard Library	Description	Examples
`python_full_version`	`platform.python_version()`	The full Python version	`3.11.0`, `3.12.0a1`
`platform_python_implementation`	`platform.python_implementation()`	The Python implementation	`CPython`, `PyPy`
`implementation_name`	`sys.implementation.name`	The Python implementation	`cpython`, `pypy`
`implementation_version`	`sys.implementation.version`	The Python implementation version	`3.11.0`, `3.12.0a1`

The `python_version` and `implementation_version` markers apply further transformations to their source in the standard library; see PEP 508 for details.

For completeness, there is another environment marker named `extra`, which I haven't listed in Table 4-2. Build backends use the `extra` marker when they generate the core metadata field for an optional dependency. For example, the core metadata fields for the `http2` extra from Example 4-3 look like this:

```
Provides-Extra: http2
Requires-Dist: h2<5,>=3; extra == 'http2'
```

Development Dependencies

Development dependencies are third-party packages that you require during development. As a developer, you might use the `pytest` testing framework to run the test suite for your project, the Sphinx documentation system to build its docs, or a number of other tools to help with project maintenance. Your users, on the other hand, don't need to install any of these packages to run your code.

As a concrete example, let's add a small test for the `build_user_agent` function from Example 4-4. Create a module `test_random_wikipedia_article.py` in your project directory with the code from Example 4-5.

Example 4-5. Testing the generated `User-Agent` header

```
from random_wikipedia_article import build_user_a
```

```
def test_build_user_agent(): ❷
    assert 'random-wikipedia-article' in build_us
❸
```

❶ Import the function under test, `build_user_agent`.

❷ Define the test function; `pytest` looks for functions whose names start with `test`.

❸ Use the `assert` statement to check for the project name in the generated header.

You could just import and run the test from Example 4-5 manually. But even for a tiny test like this, `pytest` adds three useful features. First, you can run the test by invoking `pytest` without arguments—and this is true for any other tests you may add to your project. Second, `pytest` produces a summary with the test results. Third, `pytest` rewrites assertions in your tests to give you friendly, informative messages when they fail.

Let's run the test with `pytest`. Create and activate a virtual environment in your project, then enter the commands below to install and run `pytest` alongside your project:

```
$ py -m pip install .
$ py -m pip install pytest
$ py -m pytest
========================= test session starts ===
platform darwin -- Python 3.11.1, pytest-7.2.1, p
rootdir: ...
plugins: anyio-3.6.2
collected 1 item

test_random_wikipedia_article.py .



========================== 1 passed in 0.12s ====
```

For now, things look great. Tests ensure that your project can evolve without breaking things, and the test for `build_user_agent` is a first step in that direction. Installing and running the test runner is a small infrastructure cost compared to these long-term benefits.

But how do you ensure that tools like `pytest` are set up correctly? Each of your projects is going to have slightly different requirements. You could add a small note to the project `README` for your contributors (and your future self). But eventually there may be more tools: plugins for `pytest`, tools to build the documentation, tools to analyze code for common bugs. You could add those tools to the project dependencies. But that would be wasteful; your users don't need those packages to run your code.

Python doesn't yet have a standard way to declare the development dependencies of a project. Generally, people use one of three approaches to fill the gap: optional dependencies, requirements files, or dependency groups. In this section you'll learn how to declare development dependencies using extras and optional dependencies. Requirements files allow you to list dependencies in a packaging-agnostic way, outside of the project metadata; I'll introduce them in the next section. Dependency groups are a feature of project managers, which I'll cover in Chapter 5.

Let's recap why keeping track of development dependencies is helpful:

- You don't need to remember how to set up the development environment for each project.
- You make life easier for any potential contributors as well.
- It helps with automating checks, both locally and in Continuous Integration (CI).
- You can make sure you get compatible versions of the tools. Your test suite may not work with some versions of `pytest`, and your docs may not build (or not look good) on all versions of Sphinx.

Extras are groups of optional dependencies that are recorded in the project metadata (see "Extras and Optional Dependencies"). The extras mechanism provides all the necessary ingredients to track development dependencies. The packages aren't installed by default, they can be grouped under declarative names like `tests` or `docs`, and they come with the full expressivity of dependency specifications, such as version constraints and environment markers. Example 4-6 shows how you can use extras to represent the development dependencies of a project.

Example 4-6. Using extras to represent development dependencies

```
[project.optional-dependencies]
tests = ["pytest>=7.2.1", "pytest-sugar>=0.9.6"]
docs = ["sphinx>=6.1.3"]
```

I've added `pytest-sugar` to the `tests` extra, which is a `pytest` plugin that formats the test output in a nice way. There's also a `docs` extra for building documentation with Sphinx; I've added it to demonstrate that you can have multiple groups of dependencies, but you won't be using it in this chapter.

Contributors can now install the test dependencies using the `tests` extra:

```
$ py -m pip install -e .[tests]
$ py -m pytest
```

You can also provide a `dev` extra that combines all the development dependencies, to make it easy to set up a local development environment. Instead of repeating all the dependencies, you can just reference the other extras, as shown in Example 4-7:[4]

Example 4-7. Providing a `dev` extra with all development dependencies

```
[project.optional-dependencies]
tests = ["pytest>=7.2.1", "pytest-sugar>=0.9.6"]
docs = ["sphinx>=6.1.3"]
dev = ["random-wikipedia-article[tests,docs]"]
```

There's a slight mismatch between development dependencies and extras. The primary purpose of extras is to give users control over the installation of packages that aren't needed for the primary functionality of a project. Development dependencies are never intended to be installed by users, so they're somewhat misplaced in the public metadata of a package.

Another mismatch is that you can't install extras without the project itself. Development tools don't necessarily need your project to be installed. For example, linters analyze your source code for bugs and potential improvements. You can just run them on the Python code without installing the project into the environment.

Requirements Files

Unlike extras, *requirements files* are dependency specifications that aren't part of the project metadata. You share them with your contributors using the version control system, not with your users using distribution packages, which is a good thing. What's more, requirements files don't implicitly include your project in the dependencies. That shaves off time from all tasks that don't need the project installed, such as documentation builds and linting.

At their core, *requirements files* are plain text files where each line is a dependency specification (see Example 4-8).

Example 4-8. A simple requirements.txt file

```
# requirements.txt
pytest>=7.2.1
pytest-sugar>=0.9.6
sphinx>=6.1.3
```

You can install the dependencies listed in a requirements file using pip:

```
$ py -m pip install -r requirements.txt
```

The file format is not standardized; in fact, each line of a requirement file is essentially an argument for `pip install`. In addition to dependency specifications (which *are* standardized), it can have URLs and file paths, optionally prefixed by `-e` for an editable install, as well as global options such as `-r` to include another requirements file. The file format also supports Python-style comments (with a leading `#` character) and line continuations (with a trailing `\` character).

Requirements files are commonly named *requirements.txt*, but variations are common. For example, you could have a *dev-requirements.txt* for development dependencies or a *requirements* directory with one file per dependency group. Let's replicate Example 4-7 using the third option:

Example 4-9. Using requirements files to specify development dependencies

```
# requirements/tests.txt
-e .
```

```
pytest>=7.2.1
pytest-sugar>=0.9.6

# requirements/docs.txt
sphinx>=6.1.3

# requirements/dev.txt
-r tests.txt
-r docs.txt
```

NOTE

Paths in *requirements.txt* are evaluated relative to the current directory. However, if you include other requirement files using `-r`, their paths are evaluated relative to the including file.

Create and activate a virtual environment, then run the following commands to install the development dependencies and run the test suite:

```
$ py -m pip install -r requirements/dev.txt
$ py -m pytest
```

Note that tests do need the project to be installed, so I've listed it in *requirements/tests.txt*.

Requirements files solve some of the shortcomings of extras when used for development dependencies. On the downside, requirements files are not backed by an interoperability standard, and they create some clutter in your project directory compared to having a dedicated section in *pyproject.toml*. In Chapter 5, I'll introduce a dependency-group feature offered by some Python project managers that leverages *pyproject.toml* without polluting the project metadata.[5]

Locking Dependencies

You've specified your dependencies and development dependencies, installed them in a development environment, run your test suite and whichever other checks you have in place: everything looks good, and you're ready to deploy your code to production.

There's just one little hitch. How can you be sure that you install the same dependencies in production as you did when you ran your checks? The more exposure your production code gets, the more worrying the possibility that it might run with a buggy or, worse, hijacked dependency. It could be a direct dependency of your project or a package deeper down in the dependency tree—an *indirect dependency*.

Even if you take care to upgrade your dependencies to the latest version when testing, a new release could come in just before you deploy. You can also end up with different dependencies if your development environment does not match the production environment exactly: the mismatch can cause installers to evaluate environment markers and wheel compatibility tags differently.[6] Tooling configuration or state can also cause different results—for example, pip might install from a different package index or from a local cache.

The problem is compounded if one of your dependencies doesn't provide wheels for the target environment—and it's common for binary extension modules to lag behind when a new Python version sees the light. The installer must then build a wheel from the sdist on the fly, which introduces more uncertainty: your installs are now only as reproducible as your builds. And in the worst case, that package could compute its own dependencies dynamically during build time.

Presumably, somewhere in your deployment process, there's a line like this:

```
py -m pip install my-awesome-app
```

The installer will honor all version constraints from the `dependencies` table in *pyproject.toml*. But as you saw above, it won't select the same packages on every run and every system. You need a way to define the exact set of packages required by your application, and you want its environment to be an exact image of this package inventory. This process is known as *pinning*, or *locking*, the project dependencies.

What if you replace each version range in *pyproject.toml* with a single version? Here's how that would look like for `random-wikipedia-article`:

```
[project]
dependencies = [
    "httpx[http2]==0.23.3",
    "rich==13.3.1",
    "importlib-metadata==6.0.0; python_version <
]
```

There are a couple of problems with this approach. First, you've only pinned the direct dependencies. The application communicates via HTTP/2 using `h2`, a dependency of `httpx`—but `h2` isn't mentioned in the list above. Should you add indirect dependencies to the `dependencies` table? That list would quickly become hard to maintain. And you'd start to rely on implementation details of the packages you actually import in your code.

Second, you've lost valuable information about the packages with which your application is compatible. Pip's dependency resolver used that information to compute the versions above in the first place, but you won't have it the next time you want to upgrade your dependencies. Losing that information also makes it that much harder to install the application in a different environment—for example, when your production environment upgrades to a new Python release.

Freezing Requirements with pip

Luckily, you know a format for specifying dependencies outside of the `dependencies` table: requirements files. In "Requirements Files", you wrote them by hand, but pip can also generate this file from an existing environment.

```
$ py -m pip install .
$ py -m pip freeze
anyio==3.6.2
Brotli==1.0.9
certifi==2022.12.7
h11==0.14.0
h2==4.1.0
hpack==4.0.0
httpcore==0.16.3
httpx==0.23.3
hyperframe==6.0.1
idna==3.4
```

```
idna==3.4
markdown-it-py==2.1.0
mdurl==0.1.2
Pygments==2.14.0
random-wikipedia-article @ file:///path/to/projec
rfc3986==1.5.0
rich==13.3.1
sniffio==1.3.0
```

You could store this list in *requirements.txt* and add the file to your project repository—omitting the line with the project path. When deploying your project to production, you could install the project and its dependencies like this:

```
$ py -m pip install -r requirements.txt
$ py -m pip install .
```

If you've paid close attention, you may have noticed that the requirements file didn't list `importlib-metadata`. That's because you ran `pip freeze` in an environment with a recent Python version—`importlib-metadata` is only required on Python 3.7 and below. If your production environment uses such an old version of Python, your deployment will break: you need to lock your dependencies in an environment that matches production.

TIP

Lock your dependencies on the same Python version, Python implementation, operating system, and processor architecture as those used in production. If you deploy to multiple environments, generate a requirements file for each one, using a naming convention like `win32-py311-requirements.txt`.

Generating a *requirements.txt* with `pip freeze` gives you fairly reliable deployments, but it comes with a few limitations. First, you'll need to install your dependencies into an environment every time you want to refresh the requirements file. That's especially cumbersome if you upgrade a single dependency at a time—which makes it easier to pinpoint failure to particular releases in your dependency tree.

Second, it's easy to pollute the requirements file inadvertently. If you quickly install a package locally to try it out or debug a failure, remember to create the environment from scratch afterward, or the package may end up in production the next time you update the requirements file. Uninstalling the package may not be enough: the installation can have side effects on your dependency tree, such as upgrading or downgrading other packages or pulling in additional dependencies.

And there's a third limitation of freezing: Requirements files allow you to specify package hashes for each dependency. These hashes add another layer of security to your deployments because they enable you to install only vetted packaging artifacts in production. For example, the following requirements file lists SHA256 hashes over the sdist and wheel for an `httpx` release:

```
httpx==0.23.3 \
  --hash=sha256:9818458eb565bb54898ccb9b8b251a2878
  --hash=sha256:a211fcce9b1254ea24f0cd6af9869b3d29
```

There may be additional hashes for releases with multiple wheels, as is common with binary extension modules.

Unfortunately, `pip freeze` does not have access to these hashes (or the packages over which they should be computed). It takes an inventory of an environment, and environments don't record hashes for the packages you install into them.

Pinning Dependencies with pip-tools

The pip-tools project provides dependency locking for your projects without these limitations. It generates requirements directly from *pyproject.toml* and other files; optionally, you can include package hashes for every requirement. Under the hood, pip-tools leverages pip and its dependency resolver.

Pip-tools comes with two commands: `pip-compile`, to create a requirements file from dependency specifications, and `pip-sync`, to apply the requirements file to an existing environment. Alternatively, you can invoke pip-tools via the Python Launcher, using the commands `py -m piptools compile` and `py -m piptools sync` (see Table 4-3). Note that `py -m` expects the import name of pip-tools, which is `piptools` without a hyphen. Import names must be legal Python identifiers, so they cannot contain hyphens.

Table 4-3. Pip-tools scripts and subcommands

Entry-point script	**Equivalent subcommand**	**Description**
`pip-compile`	`py -m piptools compile`	Compile *requirements.txt* from *pyproject.toml* and other files
`pip-sync`	`py -m piptools sync`	Synchronize virtual environments with *requirements.txt*

Install pip-tools in an environment that matches the target environment for your project—the environment where you're going to apply the requirements file. Assuming that the development environment for your project resembles its production environment, the simplest option is to install pip-tools into that environment:

```
$ py -m pip install pip-tools
```

Just like `py -m pip` is preferable to plain `pip`, I recommend `py -m piptools` over `pip-compile` and `pip-sync`. Here's an example invocation:

```
$ py -m piptools compile
```

You can also install pip-tools globally using pipx—but the same caveat applies. The pipx-managed environment must closely match the target environment for the requirements file. Specify the Python version and implementation using the `--python` option of `pipx install`. Additionally, use the `--suffix` option to rename the entry-point scripts with a suffix indicating the interpreter on which they run. This will save you a headache when one of your projects or environments needs a different interpreter than the others—and it allows you to install multiple versions of pip-tools globally.

For example, here's how you'd install pip-tools for a Python 3.7 environment using PyPy. (This example assumes that your system has a `pypy3.7` command.)

```
$ pipx install --python=pypy3.7 --suffix=-pypy3.
    installed package pip-tools 6.12.2 (pip-tools-
    installed using Python 3.7.13 (...)
  [PyPy 7.3.9 with GCC Apple LLVM 13.1.6 (clang-13
  These apps are now globally available
    - pip-compile-pypy3.7
    - pip-sync-pypy3.7
done!
```

This gives you a global command to compile requirements for any project using `pypy3.7`:

```
$ pip-compile-pypy3.7
```

By default, `pip-compile` writes to the file *requirements.txt* when processing dependencies from *pyproject.toml*. You can use the `--output-file` option to specify a different destination, including `-` for standard output. The tool also prints requirements to standard error with syntax highlighting, unless you specify `--quiet` to switch off terminal output.

The option `--generate-hashes` allows you to include SHA256 hashes for each package listed in the requirements file. This makes the installation more deterministic and reproducible. It is also important if you work in an environment where every artifact that goes into production must be carefully screened to prevent *supply chain attacks*—attacks exploiting vulnerabilities in dependencies rather than the application itself. Hashes also have the side effect that pip refuses to install packages without them, so either all packages have hashes, or none do. As a consequence, hashes prevent you from installing dependencies that are not listed in the requirements file.

For backward compatibility, pip-tools still defaults to pip's legacy dependency resolver. It also excludes `pip` and `setuptools` from requirements files, since pinning these packages is considered unsafe for legacy dependencies that distribute sdists without a *pyproject.toml* file. For new projects, you should opt into the future default behavior using the options `--resolver=backtracking` and `--allow-unsafe`.

Pip-tools annotates the file with useful comments indicating the origin of each dependency, as well as a header containing the command used to generate the file. I'm disabling both of these for brevity here, but you may prefer to keep them in your requirements files.

Without further ado, let's generate a *requirements.txt* file for `random-wikipedia-article`. (Omit the backslash and line break when you enter this command at your prompt.)

```
$ py -m piptools compile \
    --resolver=backtracking --allow-unsafe --no-
anyio==3.6.2
brotli==1.0.9
```

```
certifi==2022.12.7
h11==0.14.0
h2==4.1.0
hpack==4.0.0
httpcore==0.16.3
httpx[brotli,http2]==0.23.3
hyperframe==6.0.1
idna==3.4
markdown-it-py==2.1.0
mdurl==0.1.2
pygments==2.14.0
rfc3986[idna2008]==1.5.0
rich==13.3.1
sniffio==1.3.0
```

As before, install the requirements file in the target environment using pip, followed by the name of the package itself. There are a couple of pip options you can use to harden the installation: the option `--no-deps` ensures that you only install packages listed in the requirements file, and the option `--no-cache-dir` prevents pip from reusing downloaded or locally built artifacts.

```
$ py -m pip install -r requirements.txt
$ py -m pip install --no-deps --no-cache-dir .
```

It's best to update your dependencies periodically. How often depends on the project, but once per week is a good default for an application running in production. Otherwise, you may need to perform a "big bang" upgrade under time pressure: fixing a security vulnerability in one of your dependencies could, in the worst case, force you to migrate your application to multiple new APIs. Tools like Dependabot and Renovate greatly help with this chore by opening pull requests with automated dependency upgrades.

You can either upgrade all your dependencies at once, or one dependency at a time. Use the `--upgrade` option to upgrade all dependencies to their latest version, or pass a specific package with the `--upgrade-package` option. For example, here's how you'd upgrade `rich` to the latest version:

```
$ py -m piptools compile --upgrade-package=rich
```

If you don't want to create the target environment from scratch, you can use pip-tools to synchronize it with the updated requirements file. Don't install pip-tools in the target environment for this, as your dependencies may conflict with those of pip-tools. Instead, use pipx to install pip-tools globally, then specify the `--python-executable` option to point it to the target environment:

```
$ pipx install pip-tools
$ pipx install pip-tools
$ pip-sync --python-executable=venv/bin/python
$ venv/bin/python -m pip install --no-deps --no-
```

This example assumes that the target environment is under *venv* in the current directory. On Windows, the interpreter path would be *venv\Scripts\python.exe* instead. Also, `pip-sync` always removes the project itself, so remember to re-install it after synchronizing the dependencies.

So far, you've seen how to lock dependencies for reliable and reproducible deployments, but locking is also beneficial during development. By sharing the requirements file with your team and with contributors, you put everybody on the same page: every developer uses the same dependencies when running the test suite, building the documentation, or performing similar tasks. And by using the same requirements during Continuous Integration, you avoid surprises when developers publish their changes to a shared repository. However, to truly reap these benefits, you'll need to widen your view to include development dependencies.

In "Development Dependencies", you saw two ways to declare development dependencies: extras and requirements files. Pip-tools supports both as inputs. That's right: you can use requirements files as inputs for other requirements files. But let's start with extras.

You can pass an extra to `pip-compile` using, well, the `--extra` option. If your project has a `dev` extra, generate the requirements file for development like this:

```
$ py -m piptools compile --extra=dev --output-fi
    --resolver=backtracking --allow-unsafe
```

If you have finer-grained extras, the process is the same. You may want to store the requirements files in a *requirements* directory instead to avoid clutter.

WARNING

Unfortunately, pip-tools doesn't currently support recursive extras like the `dev` extra in Example 4-7. If you run into this limitation, either specify the development dependencies in requirements files or duplicate the dependencies from the other extras.

If you specify development dependencies in requirements files outside of the project metadata, pass each of these files to pip-tools in turn. By convention, input requirements use the *.in* extension, while output requirements—the locked ones—use the *.txt* extension. If you follow the *requirements.in* naming convention, pip-tools will derive the names of the output files as appropriate.

Example 4-10 shows how you’d set this up for the `tests`, `docs`, and `dev` requirements from Example 4-9.

Example 4-10. Using requirements files to specify development dependencies

```
# requirements/tests.in
pytest>=7.2.1
pytest-sugar>=0.9.6

# requirements/docs.in
sphinx>=6.1.3

# requirements/dev.in
-r tests.in
-r docs.in
```

Unlike in Example 4-9, I haven’t included the project itself in the input requirements. If I did, pip-tools would insert the full project path, which may not be the same for every developer. Instead, pass *pyproject.-toml* together with *tests.in* and *dev.in* to lock the entire set of dependencies together. You’ll also have to specify the output file explicitly if you pass more than a single input file. When installing from the resulting files, remember to install the project as well.

```
$ py -m piptools compile requirements/tests.in py
      --output-file=requirements/tests.txt
      --output-file=requirements/tests.txt
$ py -m piptools compile requirements/docs.in
$ py -m piptools compile requirements/dev.in pyp
      --output-file=requirements/dev.txt
```

You may wonder why I bothered to compile *dev.txt* at all. Couldn't I have just referenced the generated *docs.txt* and *tests.txt* files? In fact, it's essential to let the dependency resolver see the full picture—all the input requirements. If you simply install separately locked requirements on top of each other, you may well end up with conflicting dependencies.

Table 4-4 summarizes the command-line options for `pip-compile` you've seen in this chapter:

Table 4-4. Selected command-line options for `pip-compile`

Option	Description
`--generate-hashes`	Include SHA256 hashes for every packaging artifact
`--resolver=backtracking`	Don't use the legacy dependency resolver
`--allow-unsafe`	Pin packages like setuptools, which were deemed unsafe for legacy projects
`--output-file`	Specify the destination file, or `-` for standard output
`--quiet`	Do not print the requirements to standard error
`--no-header`	Omit the header with the command used to generate the file
`--no-annotations`	Omit the comments indicating the origin of each dependency
`--upgrade`	Upgrade all dependencies to their latest version

Option	Description
`--upgrade-package=PACKAGE`	Upgrade a specific package to its latest version
`--extra=EXTRA`	Include dependencies from the given extra in *pyproject.toml*

Summary

In this chapter, you've learned how to declare the project dependencies using *pyproject.toml* and how to declare development dependencies using either extras or requirements files. You've also learned how to lock dependencies for reliable deployments and reproducible checks using pip-tools. In the next chapter, you'll see how the project manager Poetry helps with dependency management using dependency groups and lock files.

1 In a wider sense, the dependencies of a project consist of all software packages that users require to run its code. This includes the interpreter, the standard library, third-party modules, and system libraries. Conda supports this generalized notion of dependencies, and so do distro-level package managers like `apt`, `dnf`, and `brew`.

2 Its counterpart also has a name: "Look Before You Leap" (LBYL).

[illegible] For the sake of simplicity, this code doesn’t handle multiple authors—which one ends up in the header is undefined.

[illegible] This technique is sometimes called *recursive optional dependencies*.

[illegible] For completeness, there’s a fourth way to handle development dependencies, and that’s not to declare them as project dependencies at all. Instead, you automate the environment creation using a tool like Nox, tox, or Hatch and include the dependencies as part of that. Chapter 6 covers test automation with Nox in detail.

[illegible] See “Environment Markers” and “Wheel Compatibility Tags”.

Chapter 5. Managing Projects with Poetry

In the preceding chapters, you've seen all the building blocks for publishing production-quality Python packages. You've learned how to write a *pyproject.toml* for your project, how to create an environment and install the project with `venv` and `pip`, and how to build packages and upload them with `build` and `twine`.

By standardizing the build backend interface and project metadata, *pyproject.toml* has broken the setuptools monopoly and brought diversity to the packaging ecosystem. At the same time, defining a Python package has gotten easier: the legacy boilerplate of *setup.py* and untold configuration files is gone, replaced with a single well-specified file with great tooling support.

Yet, some problems remain.

Before you can work on a *pyproject.toml*-based project, you need to research packaging workflows, configuration files, and associated tooling. You have to choose one of a number of available build backends (see "Build Frontends and Build Backends")—and many people don't know what those are, let alone how to choose them. Important aspects of Python packages remain unspecified—for example, how project sources are laid out and which files should go into the packaging artifacts.

Dependency and environment management could be easier, too. You need to handcraft your dependency specifications and compile them with pip-tools, cluttering your project with requirements files. And it can be hard to keep track of the many Python environments on a typical developer system.

The project-management tool Poetry has been addressing these problems since (and even before) the standards governing *pyproject.toml* were taking shape. Its friendly command-line interface lets you perform most of the tasks related to packaging, dependencies, and environments. Poetry brings its own standards-compliant build backend, `poetry.core` —but you can remain blissfully unaware of this fact. It also comes with a strict dependency resolver and locks all dependencies by default, behind the scenes.

Poetry abstracts away many of the details I've covered in the preceding three chapters. Still, learning about packaging standards and the low-level tooling that implements them is well worth your while. Poetry itself largely works in the framework defined by packaging standards, even though it also ventures into new territory. Standard mechanisms like dependency specifications or virtual environments power Poetry's central features, and Poetry-managed projects leverage interoperability standards when interacting with package repositories, build frontends, and installers. An understanding of the underlying mechanisms helps you debug situations where Poetry's convenient abstractions break down, such as when misconfigurations or bugs cause packages to be installed in the wrong environment. Finally, the experience of the past decades teaches us that packaging tools come and go, while packaging standards are here to stay.

THE EVOLUTION OF PYTHON PROJECT MANAGERS

A decade ago, Python packaging was firmly in the hands of three tools: setuptools, virtualenv, and pip. You'd use setuptools to create Python packages, virtualenv to set up virtual environments, and pip to install packages into them. Everybody did. Around 2016—the same year that the *pyproject.toml* file became standard—things started to change.

In 2015, Thomas Kluyver began developing flit, an alternative build tool that could create packages and publish them to PyPI. In 2016, Donald Stufft from the pip maintainer team started working on Pipfile, a proposed replacement for requirements files, including a specification of lock files. In 2017, his work led to Kenneth Reitz's pipenv, which allows you to manage dependencies and environments for Python applications and deploy them in a reproducible way. Pipenv deliberately didn't package your application: you'd just keep a bunch of Python modules in a Git repository.

Poetry, started in 2018 by Sébastien Eustace, was the first tool to provide a unified approach to packaging, dependencies, and environments—and its adoption quickly spiraled. Two other tools follow a similarly holistic approach: PDM, started by Frost Ming in 2019, and Hatch by Ofek Lev in 2017. Hatch has recently grown in popularity, especially among tooling and library developers.

Poetry, Hatch, and PDM each provide a single user interface and a streamlined workflow for Python packaging as well as for environment and dependency management. As such, they have come to be known as *Python project managers*.

Installing Poetry

Install Poetry globally using pipx, to keep its dependencies isolated from the rest of the system:

```
$ pipx install poetry
```

A single Poetry installation works with multiple Python versions. However, Poetry uses its own interpreter as the default Python version. For this reason, it's worthwhile to install Poetry on the latest stable Python release. When installing a new feature release of Python, reinstall Poetry like this:

```
$ pipx reinstall --python=python3.12 poetry
```

You can omit the `--python` option if pipx already uses the new Python version (see "Installing Applications with Pipx").

When a prerelease of Poetry becomes available, you can install it side-by-side with the stable version:

```
$ pipx install poetry --suffix=@preview --pip-ar
```

Above, I’ve used the `--suffix` option to rename the command so you can invoke it as `poetry@preview`, while keeping `poetry` as the stable version. The `--pip-args` option lets you pass options to pip, like `--pre` for including prereleases.

NOTE

Poetry also comes with an official installer. You can download the installer and run it with Python. It’s not as flexible as pipx, but it provides a simple and readily available alternative.

Upgrade Poetry periodically to receive improvements and bugfixes:

```
$ pipx upgrade poetry
```

Type `poetry` on its own to check your installation of Poetry. Poetry prints its version and usage to the terminal, including a useful listing of all available subcommands.

```
$ poetry
```

Having successfully installed Poetry, you may want to enable tab completion for your shell. Use the command `poetry help completions` for shell-specific instructions. For example, the following command enables tab completion in the Bash shell:

```
$ poetry completions bash >> ~/.bash_completion
```

Restart your shell for the changes to take effect.

Creating a Project

You can create a new project using the command `poetry new`. As an example, I'll use the `random-wikipedia-article` project from previous chapters. Run the following command in the parent directory where you want to keep your new project:

```
$ poetry new --src random-wikipedia-article
```

After running this command, you'll see that Poetry created a project directory named *random-wikipedia-article*, with the following structure:

```
random-wikipedia-article
├── README.md
├── pyproject.toml
├── src
│   └── random_wikipedia_article
│       └── __init__.py
└── tests
    └── __init__.py
```

In earlier chapters, the project contained a single top-level module, *random_wikipedia_article.py*. In the listing above, you've replaced this with an import package—a directory with an *__init__.py* file. The `--src` option instructs Poetry to place that import package in a subdirectory named *src* rather than directly in the project directory.

THE *SRC* LAYOUT

Until a few years ago, package authors placed the import package directly in the project directory. These days, a project layout with *src*, *tests*, and *docs* directories at the top is becoming more common.

Keeping the import package tucked away under *src* has practical advantages. During development, the current directory often appears at the start of `sys.path`. Without a *src* layout, you may be importing your project from its source code, not from the package you've installed in the project environment. In the worst case, your tests could fail to detect issues in a release you're about to publish.

On the other hand, whenever you *want* to execute the source code itself, editable installs achieve this by design. With a *src* layout, packaging tools can implement editable installs by adding the *src* directory to `sys.path`—without the side effect of making unrelated Python files importable.

Let's take a look at the generated *pyproject.toml* (Example 5-1):

Example 5-1. A pyproject.toml file for Poetry

```
[tool.poetry]
```

```
name = "random-wikipedia-article"
version = "0.1.0"
description = ""
authors = ["Your Name <you@example.com>"]
readme = "README.md"
packages = [{include = "random_wikipedia_article'

[tool.poetry.dependencies]
python = "^3.11"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Poetry has created a standard `build-system` table with its build backend, `poetry.core`. This means anybody can install your project from source using pip—no need to set up, or even know about, the Poetry project manager. Similarly, you can build packages using any standard build frontend, such as `build`. Try it yourself from within the project directory:

```
$ pipx run build
* Creating venv isolated environment...
* Installing packages in isolated environment...
* Getting build dependencies for sdist...
* Building sdist...
* Building wheel from sdist
* Creating venv isolated environment...
* Installing packages in isolated environment...
* Getting build dependencies for wheel...
* Building wheel...
Successfully built random_wikipedia_article-0.1.(
 and random_wikipedia_article-0.1.0-py3-none-any.
```

The Project Metadata

You may be surprised to see the project metadata appear under `tool.poetry` instead of the familiar `project` table (see "Project Metadata"). The Poetry project has promised support for the project metadata standard in its next major release. As you can see in Table 5-1, most fields have the same name and the same, or similar, syntax and meaning.

Let's fill in the metadata for the project. Example 5-2 shows how you'd translate Example 3-4 for the `tool.poetry` table. I've omitted the dependencies section here; you'll use the command-line interface to add the project dependencies later.

Example 5-2. Metadata for a Poetry project

```
[tool.poetry]
```

```
name = "random-wikipedia-article"
version = "0.1.0"
description = "Display extracts from random Wikip
keywords = ["wikipedia"]
license = "MIT" ❶
classifiers = [
    "License :: OSI Approved :: MIT License",
    "Development Status :: 3 - Alpha",
    "Environment :: Console",
    "Topic :: Games/Entertainment :: Fortune Cook
]
authors = ["Your Name <you@example.com>"] ❷
readme = "README.md" ❸
homepage = "https://yourname.dev/projects/random-
repository = "https://github.com/yourname/random-
documentation = "https://readthedocs.io/random-wi
packages = [{include = "random_wikipedia_article"

[tool.poetry.urls]
Issues = "https://github.com/yourname/random-wiki

[tool.poetry.scripts]
random-wikipedia-article = "random_wikipedia_arti
```

❶ The `license` field is a string with a SPDX identifier, not a table.

❷ The `authors` field contains strings in the format `name <email>`, not tables. Poetry pre-populates the field with your name and email using the local Git configuration.

❸ The `readme` field is a string with the file path. You can also specify multiple files as an array of strings, such as *README.md* and *CHANGELOG.md*. Poetry concatenates them with a blank line in between.

❹ Poetry has dedicated fields for some project URLs, namely its homepage, repository, and documentation; for other URLs, there's also a generic `urls` table.

Table 5-1. Metadata fields in `tool.poetry`

Field	Type	Description	Related `project` field
`name`	string	The project name	`name`
`version`	string	The version of the project	`version`
`description`	string	A short description of the project	`description`
`keywords`	array of strings	A list of keywords for the project	`keywords`
`readme`	string or array of strings	A file or list of files with a long description of the project	`readme`
`license`	string	A SPDX license identifier, or “Proprietary”	`license`
`authors`	array of strings	The list of authors	`authors`

Field	Type	Description	Related `project` field
`maintainers`	array of strings	The list of maintainers	`maintainers`
`classifiers`	array of strings	A list of classifiers describing the project	`classifiers`
`homepage`	string	The URL of the project homepage	`urls`
`repository`	string	The URL of the project repository	`urls`
`documentation`	string	The URL of the project documentation	`urls`
`urls`	table of strings	The project URLs	`urls`
`dependencies`	array of strings or tables	The list of required third-party packages	`dependencies`

Field	Type	Description	Related `project` field
`extras`	table of arrays of strings	Named lists of optional third-party packages	`optional-dependencies`
`groups`	table of arrays of strings	Named lists of development dependencies	*none*
`scripts`	table of strings or tables	Entry-point scripts	`scripts`
`plugins`	table of tables of strings	Entry point groups	`entry-points`

Some `project` fields have no direct equivalent under `tool.poetry`:

- There’s no `requires-python` field; instead, you specify the required Python version in the `dependencies` table, using the `python` key.

- There’s no dedicated field for GUI scripts; use `plugins.gui_scripts` instead.
- There’s no `dynamic` field—all metadata is Poetry-specific, so declaring dynamic fields wouldn’t make much sense.

Before we move on, let’s check that the *pyproject.toml* file is valid. Poetry provides a convenient command to validate the TOML file against its configuration schema:

```
$ poetry check
All set!
```

The Package Contents

Poetry allows you to specify which files and directories to include in the distribution—a feature still missing from the *pyproject.toml* standards (Table 5-2).

Table 5-2. Package content fields in `tool.poetry`

Field	Type	Description
`packages`	array of tables	Patterns for modules to include in the distribution
`include`	array of strings or tables	Patterns for files to include in the distribution
`exclude`	array of strings or tables	Patterns for files to exclude from the distribution

Each table under `packages` has an `include` key with a file or directory. You can use `*` and `**` wildcards in their names and paths, respectively. The `from` key allows you to include modules from subdirectories such as *src*. Finally, you can use the `format` key to restrict modules to a specific distribution format; valid values are `sdist` and `wheel`.

The `include` and `exclude` fields allow you to list other files to include in, or exclude from, the distribution. Poetry seeds the `exclude` field using the *.gitignore* file, if present. Instead of a string, you can also use a table with `path` and `format` keys for sdist-only or wheel-only files. Example 5-3 shows how to include the test suite in source distributions.

Example 5-3. Including the test suite in source distributions

```
packages = [{include = "random_wikipedia_article'
include  = [{path = "tests", format = "sdist"}]
```

The Source Code

Copy the contents of the *random_wikipedia_article.py* script into the *__init__.py* file in the new project. Example 5-4 has the full listing.

Example 5-4. The source code for `random-wikipedia-article`

```
import httpx
from rich.console import Console

try:
    from importlib.metadata import metadata
except ImportError:
    from importlib_metadata import metadata

API_URL = "https://en.wikipedia.org/api/rest_v1/p
USER_AGENT = "{Name}/{Version} (Contact: {Author-


def main():
    fields = metadata("random-wikipedia-article")
    headers = {"User-Agent": USER_AGENT.format_ma

    with httpx.Client(headers=headers, http2=True
        response = client.get(API_URL, follow_re
        response.raise_for_status()
        data = response.json()

    console = Console(width=72, highlight=False)
    console.print(data["title"], style="bold")
    console.print()
    console.print(data["extract"])
```

You’ve declared an entry-point script in the `scripts` section in *pyproject.toml*, so users can invoke the application as `random-wikipedia-article`. If you’d like to also allow users to invoke the program with `py -m random_wikipedia_article`, create a *__main__.py* module next to *__init__.py* as shown in Example 5-5.

Example 5-5. The *__main__.py* module

```
from random_wikipedia_article import main

main()
```

Managing Dependencies

Let’s add the dependencies for `random-wikipedia-article`, starting with Rich, the console output library:

```
$ poetry add rich
Creating virtualenv random-wikipedia-article-eBrM
Using version ^13.3.1 for rich

Updating dependencies
Resolving dependencies... (0.2s)

Writing lock file

Package operations: 4 installs, 0 updates, 0 remo

  • Installing mdurl (0.1.2)
  • Installing markdown-it-py (2.2.0)
  • Installing pygments (2.14.0)
  • Installing rich (13.3.1)
```

If you inspect *pyproject.toml* after running this command, you’ll find that Poetry has added Rich to the `dependencies` table (Example 5-6):

Example 5-6. The `dependencies` table after adding Rich

```toml
[tool.poetry.dependencies]
python = "^3.11"
rich = "^13.3.1"
```

Poetry also created a virtual environment for the project and installed the package into it (see “Managing Environments”).

Caret Constraints

The caret (`^`) is a Poetry-specific extension borrowed from npm, the package manager for Node.js. Caret constraints allow releases with the given minimum version, except those that may contain breaking changes according to the Semantic Versioning standard. After `1.0.0`, this means patch and minor releases, but not major releases. Before `1.0.0`, only patch releases are allowed—this is because in the `0.*` era, even minor releases are allowed to introduce breaking changes.

Caret constraints are similar to tilde constraints (see “Version Specifiers”), but the latter only allow the last version segment to increase. For example, the following constraints are equivalent:

```toml
rich = "^13.3.1"
rich = ">=13.3.1,<14"
```

On the other hand, tilde constraints typically exclude minor releases:

```
rich = "~13.3.1"
rich = ">=13.3.1,==13.3.*"
```

SHOULD YOU CAP DEPENDENCIES?

It's unfortunate that Poetry adds upper bounds to dependencies by default. For libraries, the practice prevents downstream users from receiving fixes and improvements, since constraints aren't scoped to the packages that introduce them, as in Node.js. Many open-source projects don't have the resources to backport fixes to past releases. For applications, dependency locking provides a better way to achieve reliable deployments.

The situation is similar but worse for the Python requirement. Excluding Python 4 by default will cause disruption across the ecosystem when the core Python team eventually releases a new major version. It's unlikely that Python 4 will come anywhere near Python 3 in terms of incompatible changes. Poetry's constraint is contagious in the sense that dependent packages must also introduce it. And it's impossible for Python package installers to satisfy—they can't downgrade the environment to an earlier version of Python.

Whenever possible, replace caret constraints in *pyproject.toml* with simple lower bounds (`>=`), especially for Python itself. Afterward, refresh the lock file using the command `poetry lock --no-update`.

Extras and Environment Markers

Let’s add the other two dependencies of `random-wikipedia-article`, the HTTP client library `httpx` and the `importlib-metadata` backport. Like in Chapter 4, you’ll activate the `http2` extra for HTTP/2 support, and restrict the backport to Python versions before 3.8.

```
$ poetry add httpx --extras=http2
$ poetry add importlib-metadata --python='<3.8'
```

Poetry updates the *pyproject.toml* file accordingly:

```
[tool.poetry.dependencies]
python = "^3.11"
rich = "^13.3.1"
httpx = {version = "^0.23.3", extras = ["http2"]}
importlib-metadata = {version = "^6.0.0", python
```

Besides `--python`, the `poetry add` command supports a `--platform` option to restrict dependencies to a specific operating system, such as Windows. This option accepts a platform identifier in the format used by the standard `sys.platform` attribute: `linux`, `darwin`, `win32`. For other environment markers, edit *pyproject.-*

toml and use the `markers` property in the TOML table for the dependency:

```
[tool.poetry.dependencies]
awesome = {version = ">=1", markers = "implementa
```

The Lock File

Poetry records the current version of each dependency in a file named *poetry.lock*, including SHA256 hashes for its packaging artifacts. If you take a peek inside the file, you'll notice TOML stanzas for `rich`, `httpx`, and `importlib-metadata`, as well as their direct and indirect dependencies, somewhat like the one shown in Example 5-7:

Example 5-7. The TOML stanza for Rich in *poetry.lock* (simplified)

```
[[package]]
name = "rich"
version = "13.3.1"
python-versions = ">=3.7.0"
files = [
    {file = "rich-13.3.1-py3-none-any.whl", hash
    {file = "rich-13.3.1.tar.gz", hash = "sha256
]
```

Use the command `poetry show` to display the locked dependencies on the terminal. Here's what the output looked like after I added Rich:

```
$ poetry show
markdown-it-py 2.2.0  Python port of markdown-it.
mdurl          0.1.2  Markdown URL utilities
pygments       2.14.0 Pygments is a syntax highli
rich           13.3.1 Render rich text, tables, p
```

You can also display the dependencies as a tree to visualize their relationship:

```
$ poetry show --tree
rich 13.3.1 Render rich text, tables, progress ba

├── markdown-it-py >=2.1.0,<3.0.0
│   └── mdurl >=0.1,<1.0
└── pygments >=2.14.0,<3.0.0
```

Resolving dependencies up front means that you can deploy applications securely and reliably, and it allows larger teams to share a common baseline during development. Deterministic and repeatable installations also avoid surprises when you run checks during Continuous Integration (CI). Commit *poetry.lock* to source control to reap these benefits.

Poetry's lock file is designed to work across operating systems and Python interpreters—an important difference from the requirements files that pip-tools generates. If your project supports Windows, macOS, and Linux on the four most recent feature versions of Python, you would need to compile a dozen requirements files—and those would only cover a single processor architecture and Python implementation. Having a single lock file for all environments makes dependency locking a much more practical development workflow than it would otherwise be.

If you edit *pyproject.toml* yourself, remember to update the lock file to reflect your changes:

```
$ poetry lock --no-update
Resolving dependencies... (0.1s)

Writing lock file
```

Without the `--no-update` option, Poetry also upgrades each locked dependency to the latest version covered by its constraint.

You can also check if the *poetry.lock* file is consistent with *pyproject.-toml*:

```
$ poetry lock --check
```

Updating Dependencies

You can update all dependencies in the lock file to their latest versions using a single command:

```
$ poetry update
```

You can also provide a specific direct or indirect dependency to update:

```
$ poetry update rich
```

The `poetry update` command doesn't modify the project metadata in *pyproject.toml*. It only updates dependencies within the compatible version range. If you want to bump a dependency to a release not yet covered by the version constraint, the easiest method is to add the dependency again, with the special `latest` version:

```
$ poetry add rich@latest
```

This bumps the lower bound of the caret constraint to the latest version of Rich. You can also specify the new constraint yourself after the `@`-sign:

```
$ poetry add rich@'>=13.3.1'
```

This is a handy method for removing an upper bound that also keeps the lock file and project environment up-to-date.

If you no longer need a package for your project, you can remove it using `poetry remove`:

```
$ poetry remove importlib-metadata
```

Managing Environments

As you can see in the output of `poetry add` and `poetry update`, these commands don't just update dependencies in the *pyproject.toml* and *poetry.lock* files. They also install them into your project environment, keeping it synchronized with the lock file.

You can enter the project environment by launching a shell session with `poetry shell`. Poetry activates the virtual environment using the activation script for your current shell.

```
$ poetry shell
```

With the environment activated, you can run the application from the shell prompt. But first, you need to install it into the environment. Poetry performs an editable install of the project, so the environment reflects any code changes immediately.

```
(random-wikipedia-article-py3.11) $ poetry insta
```

Run the application, then exit the environment.

```
(random-wikipedia-article-py3.11) $ random-wikipe
(random-wikipedia-article-py3.11) $ exit
```

You can also run the application in your current shell session, using the command `poetry run`:

```
$ poetry run random-wikipedia-article
```

The `poetry run` command is also handy for starting an interactive Python session:

```
$ poetry run python
```

You can have multiple environments for a project.

Let’s add an environment for Python 3.7 to test the `importlib-metadata` backport. First, you need to declare support for Python 3.7 by updating the `python` dependency in *pyproject.toml*:

```
[tool.poetry.dependencies]
python = "^3.7"
```

Refresh the lock file to bring it in sync with the updated *pyproject.toml*:

```
$ poetry lock --no-update
```

Now you’re ready to create and activate the environment for Python 3.7:

```
$ poetry env use 3.7
Creating virtualenv random-wikipedia-article-eBrI
Using virtualenv: .../random-wikipedia-article-eI
```

Instead of a version like `3.7`, you could also specify a command like `pypy3` for the PyPy implementation, or a full path like `/usr/bin/python3` for the system Python.

Finally, install the project into the new environment:

```
$ poetry install
Installing dependencies from lock file

Package operations: 18 installs, 0 updates, 0 rer


  • ...
  • Installing zipp (3.14.0)
  • Installing httpx (0.23.3)
  • Installing importlib-metadata (6.0.0)
  • ...

Installing the current project: random-wikipedia-
```

As expected, Poetry installed `importlib-metadata` into the Python 3.7 environment.

When you've created multiple environments, it's useful to see them listed in the terminal. Poetry provides a command for this, too:

```
$ poetry env list
```

```
random-wikipedia-article-eBrMbJNp-py3.11
random-wikipedia-article-eBrMbJNp-py3.7 (Activate
```

Use the command `poetry env info --path` to display the location of the current environment. By default, Poetry creates virtual environments in a shared folder. There's a configuration setting to keep each environment in a *.venv* directory inside its project instead:

```
$ poetry config virtualenvs.in-project true
```

Even without this setting, Poetry uses the *.venv* directory in the project if it already exists.

When you no longer need an environment, remove it like this:

```
$ poetry env remove 3.7
Deleted virtualenv: .../random-wikipedia-article-
```

You can create a clean slate by removing all environments at once:

```
$ poetry env remove --all
```

Development Dependencies

Poetry allows you to declare development dependencies, organized in groups. Development dependencies are not part of the project metadata and are invisible to end users. Let's add a dependency group for testing:

```
$ poetry add --group=tests pytest
```

Poetry has added the dependency group under the `group` table in *pyproject.toml*:

```
[tool.poetry.group.tests.dependencies]
pytest = "^7.2.1"
```

You're in for a surprise if you try to add a `docs` group with Sphinx, the documentation generator. Sphinx has dropped support for Python 3.7, so Poetry kindly refuses to add it to your dependencies. You could drop Python 3.7 yourself, but Poetry suggests another option—you can restrict Sphinx to Python 3.8 and newer:

```
$ poetry add --group=docs sphinx --python='>=3.8
```

By default, Poetry installs dependency groups into the project environment, but you can mark groups as optional using an `optional = true` declaration in *pyproject.toml*. The `poetry install` command has several options that provide finer-grained control over which dependencies are installed into the project environment.

Table 5-3. Installing dependencies with `poetry install`

Option	Description
`--with GROUP`	Include a dependency group in the installation.
`--without GROUP`	Exclude a dependency group from the installation.
`--only GROUP`	Exclude all other dependency groups from the installation.
`--no-root`	Exclude the project itself from the installation.
`--only-root`	Exclude all dependencies from the installation.
`--sync`	Remove packages from the environment unless scheduled for installation.

You can specify a single group or multiple groups (separated by commas). The special group `main` refers to packages listed in the `tool.poetry.dependencies` table in *pyproject.toml*. Use the option `--only=main` to exclude all development dependencies from an installation. Similarly, the option `--without=main` lets you restrict an installation to development dependencies.

Package Repositories

You can build Poetry-managed projects using standard tooling like `build`, or you can use the Poetry command-line interface:

```
$ poetry build
Building random-wikipedia-article (0.1.0)
  - Building sdist
  - Built random_wikipedia_article-0.1.0.tar.gz
  - Building wheel
  - Built random_wikipedia_article-0.1.0-py3-none
```

Poetry places these artifacts in the conventional *dist* directory.

Before you can upload the packages to the Python Package Index (PyPI), you need a PyPI account and an API token to authenticate with the repository (see "Packaging in a Nutshell"). Next, add the API token to Poetry:

```
$ poetry config pypi-token.pypi my-token
```

You can now publish your package using `poetry publish`:

```
$ poetry publish
Publishing random-wikipedia-article (0.1.0) to Py
 - Uploading random_wikipedia_article-0.1.0-py3-n
 - Uploading random_wikipedia_article-0.1.0.tar.g
```

You can also collapse the two commands into one:

```
$ poetry publish --build
```

If you want to publish your project to a package repository other than PyPI, you need to first add that repository to your Poetry configuration. For example, here's how you would add TestPyPI to your configured repositories. TestPyPI is a separate instance of the Python Package Index for testing distributions.

```
$ poetry config repositories.testpypi https://tes
```

Since TestPyPI is a separate instance, you need to create an account as well as an API token, and configure Poetry to use that token when uploading to TestPyPI:

```
$ poetry config pypi-token.testpypi my-token
```

You can now specify the repository when publishing your project:

```
$ poetry publish --repository=testpypi
```

If your package repository uses HTTP basic authentication with a username and password, configure the credentials for the repository like this:

```
$ poetry config http-basic.my-repo username
```

The command will prompt you for the password and store it in the system keyring, if available, or in the *auth.toml* file on disk. Alternatively, you can also configure credentials via environment variables:

```
export POETRY_PYPI_TOKEN_PYPI=<token>
export POETRY_HTTP_BASIC_PYPI_USERNAME=<username
export POETRY_HTTP_BASIC_PYPI_PASSWORD=<password
```

Poetry also supports repositories that are secured by mutual TLS or use a custom certificate authority; see the official documentation for details.

So far, I've talked about how Poetry supports custom repositories on the publisher side—how to upload your package to a repository other than PyPI. Poetry also supports custom repositories on the consumer side; in other words, you can add packages from other repositories to your project. While upload targets are a per-user setting, alternative package sources are a per-project setting and stored in *pyproject.-toml*.

```
$ poetry source add --secondary example https://
```

Poetry only searches a secondary package source if the package wasn't found on PyPI. If you want to use a different package repository than PyPI by default, you can configure the default package source:

```
$ poetry source add --default example https://exa
```

You configure credentials for package sources just like you do for repositories:

```
$ poetry config http-basic.example username
```

You can now add packages from the alternate sources:

```
$ poetry add httpx --source=example
```

WARNING

If you use secondary package sources, make sure to specify the source when adding a dependency. If you don't, Poetry searches all package sources when looking up the package. This isn't just inefficient; it opens the door for a so-called *dependency confusion attack*, where an attacker uploads a malicious package to PyPI with the same name as an internal package.

Use the command `poetry source show` to list the package sources for the current project:

```
$ poetry source show
```

Extending Poetry with Plugins

Poetry comes with a plugin system that lets you extend its functionality. Use pipx to inject the plugin into Poetry's environment:

```
$ pipx inject poetry plugin
```

Replace *`plugin`* with the name of the plugin on PyPI.

If the plugin affects the build stage of your project, you should also add it to the build requirements in *pyproject.toml*. See "The Dynamic Versioning Plugin" for an example.

By default, pipx upgrades applications without the injected packages. Use the option `--include-injected` to also upgrade application plugins:

```
$ pipx upgrade --include-injected poetry
```

If you no longer need the plugin, remove it from the injected packages:

```
$ pipx uninject poetry plugin
```

In this section, I'll introduce you to three useful plugins for Poetry:

- `poetry-plugin-export` allows you to generate requirements and constraints files
- `poetry-plugin-bundle` lets you deploy the project to a virtual environment
- `poetry-dynamic-versioning` populates the project version from the VCS

Generating Requirements Files with the Export Plugin

Poetry’s lock file is great to ensure that everybody on your team, and every deployment environment, ends up with the exact same dependencies. But what do you do if you can’t use Poetry in some context? For example, you may need to deploy your project on a system that only has a Python interpreter and the bundled pip.

As of this writing, there’s no lock file standard in the wider Python world; each packaging tool that supports locking implements its own format.[1] None of these full-fledged lock file formats has support in pip.

The closest thing we do have to a standard lock file would be requirements files. You can pin packages to an exact version, require their artifacts to match cryptographic hashes, and use environment markers to restrict packages to specific Python versions and platforms. Wouldn’t it be nice if you could generate one from your *poetry.lock*, for interoperability with non-Poetry environments?

This is precisely what the export plugin achieves, and it comes distributed with Poetry, so you don’t even need to install it. The plugin powers the `poetry export` command, which sports a `--format` option to specify the output format. By default, the command writes to the standard output stream; use the `--output` option to specify a destination file.

```
$ poetry export --format=requirements.txt --outpu
```

Distribute the requirements file to the target system and use pip to install the dependencies (typically followed by installing a wheel of your project).

```
$ python3 -m pip install -r requirements.txt
```

Exporting to requirements format is useful beyond deploying. Many tools work with requirements files as the de-facto industry standard. For example, you can scan a requirements file for dependencies with known security vulnerabilities using a tool like safety.

Deploying Environments with the Bundle Plugin

In the previous section, I showed you how to deploy your project on a system without Poetry. If you do have Poetry available, there's a simpler way. "I know", you say, "I can install my project with `poetry install`." While that's technically true, this method has a few limitations. Poetry performs an editable install, so you'll be running your application from the source tree. That may not be acceptable in a production environment, and it limits your ability to ship the virtual environment to another destination. Also, you'll install into an environment managed by Poetry, typically somewhere under your home directory.

The bundle plugin allows you to deploy your project and locked dependencies to a virtual environment of your choosing. It creates the environment, installs the dependencies from the lock file, then builds and installs a wheel of your project.

Install the plugin with pipx:

```
$ pipx inject poetry poetry-plugin-bundle
```

After installation, you should see a new `poetry bundle` subcommand. Let's bundle the project into a virtual environment in a directory named *production*. Use the `--python` option to specify the interpreter for the environment.

```
$ poetry bundle venv --python=/usr/bin/python3 p

  • Bundled random-wikipedia-article (0.1.0) intc
```

You can test the environment by activating it and invoking the application or invoke the entry-point script directly. (Replace *bin* with

Scripts if you're on Windows.)

```
$ production/bin/random-wikipedia-article
```

The bundle plugin is a great way to create a minimal Docker image for production (Example 5-8). Docker supports *multi-stage builds*, where you have a full-fledged build environment for your project in the first stage—including tools like Poetry or even a compiler toolchain for binary extension modules—but only a minimal runtime environment in the second stage. This allows you to ship slim images to production, greatly speeding up deployments and reducing bloat in your production environments.

Example 5-8. Multi-stage Dockerfile with Poetry

```
FROM python:3.11 AS builder ❶
RUN python -m pip install pipx
ENV PATH="/root/.local/bin:${PATH}"
RUN pipx install poetry
RUN pipx inject poetry poetry-plugin-bundle
WORKDIR /src
COPY . .
RUN poetry bundle venv --python=/usr/local/bin/py


FROM python:3.11 ❷
COPY --from=builder /venv /venv ❸
```

```
CMD ["/venv/bin/random-wikipedia-article"] ❹
```

❶ The first `FROM` directive introduces the build stage, where you build and install your project.

❷ The second `FROM` directive defines the image that you deploy to production.

❸ The `COPY` directive allows you to copy the virtual environment over from the build stage.

❹ The `CMD` directive lets you run the entry-point script when users invoke `docker run` with the image.

If you have Docker installed, you can try this by creating a *Dockerfile* with the contents from Example 5-8 in your project and running the following commands from the project directory:

```
$ docker build -t random-wikipedia-article .
$ docker run --rm -ti random-wikipedia-article
```

After the second command, you should see the output from `random-wikipedia-article` in your terminal.

The Dynamic Versioning Plugin

Version numbers have the annoying habit of duplicating throughout your code base and infrastructure. Partly, that’s a good thing, because it means you’re explicit about the snapshot of your codebase that gets installed. But it also creates needless redundancy, adding to the chores involved in getting your application up and running. During development, you often need the version number in three separate places: the version control system (where it may be stored as a Git tag), the build system (the `version` key in *pyproject.toml*), and the code, so you can display the version to users (often a `__version__` attribute in a Python module).

The dynamic versioning plugin allows you to derive the version for the project metadata from a Git tag. You can then reference it in the code using the standard `importlib.metadata.version` function. Install it with pipx, like this:

```
$ pipx inject poetry poetry-dynamic-versioning
```

Additionally, you need to list the plugin in your build requirements in *pyproject.toml*, so build frontends like pip know they need it to build your project. Somewhat unusually, the plugin brings its own build backend, which wraps the one provided by Poetry:

```
[build-system]
requires = ["poetry-core>=1.0.0", "poetry-dynamic
build-backend = "poetry_dynamic_versioning.backen
```

In the `tool` section, configure the plugin to derive the version from the VCS:

```
[tool.poetry-dynamic-versioning]
enable = true
vcs = "git"
style = "semver"
```

Poetry still requires the `version` key in its own section. You should set it to `0` to indicate that the key is unused.

```
[tool.poetry]
version = "0"
```

You can now add a Git tag to set your project version:

```
$ git tag v1.0.0
$ poetry build
Building random-wikipedia-article (1.0.0)
  - Building sdist
  - Built random_wikipedia_article-1.0.0.tar.gz
  - Building wheel
  - Built random_wikipedia_article-1.0.0-py3-none
```

Summary

Poetry provides a unified workflow to manage packaging, dependencies and environments. Poetry projects are interoperable with standard tooling: you can build them with `build` and upload them to PyPI with `twine`. But the Poetry command-line interface also provides handy commands for these tasks and many more.

Poetry records the precise working set of packages in its lock file, giving you deterministic deployments and checks, as well as a consistent experience when collaborating with others. Poetry can also track development dependencies for you and organizes them in dependency groups that can be installed separately or together, as desired. You can extend Poetry with plugins—for example, to bundle the project into a virtual environment for deployment or to derive the version number from Git.

If you need reproducible deployments for an application, if your team develops on multiple operating systems, or if you just feel that standard tooling adds too much overhead to your workflows, you should give Poetry a try.

Apart from Poetry’s own *poetry.lock* and the closely related PDM lock file format, there’s pipenv’s *Pipfile.lock* and the `conda-lock` format for Conda environments.

Chapter 6. Testing with pytest

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the sixth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mail@claudiojolowicz.com.

If you think back to when you wrote your first programs, you may recall a common, recurring experience: You had an idea for how a program could help with a real-life task, spent a sizable amount of time coding the program from top to bottom, only to be confronted with screens full of disheartening error messages when you finally ran it. Or worse, it gave you results that were sometimes subtly wrong.

There are a few lessons we've all learned from experiences like this. One is to start simple, and to keep it simple as you iterate on the program. Another lesson is to test early and repeatedly. Initially, this may just mean to run the program manually and validate that it does what it should. Later on, if you break the program into smaller parts, you can test those parts in isolation and in an automated fashion. As a side effect, the program gets easier to read and work on, too.

In this chapter, I'll talk about how testing can help you produce value early and consistently. Good tests amount to an executable specification of the code you own. They set you free from tribal knowledge in a team or company, and they speed up your development by giving you immediate feedback on changes. The chapter focuses on the tooling side of things, but there's so much more to good testing practices. Luckily, other people have written fantastic texts about this topic. Here are three of my personal favorites:

- Architectural Patterns in Python, by Harry Percival and Bob Gregory. Released March 2020. O'Reilly Media, Inc. ISBN: 9781492052203
- Test-Driven Development with Python, 2nd Edition, by Harry Percival. Released August 2017. O'Reilly Media, Inc. ISBN: 9781491958650
- Working Effectively with Legacy Code, by Michael Feathers. Released September 2004. Pearson. ISBN: 9780131177055

Writing a Test

Example 6-1 revisits the original Wikipedia example from Chapter 3. The program is as simple as it gets—and yet, it's far from obvious how you'd write tests for it. The `main` function has no inputs and no outputs—only side effects such as writing to the standard output stream. How would you test a function like this?

Example 6-1. The `random_wikipedia_article` module

```
def main():
    with urllib.request.urlopen(API_URL) as respo
        data = json.load(response)

    print(data["title"])
    print()
    print(textwrap.fill(data["extract"]))

if __name__ == "__main__":
    main()
```

Let's write an *end-to-end test* that runs the program in a subprocess and checks that it completes with non-empty output. End-to-end tests run the entire program the way an end-user would. Example 6-2 shows how you might do this. For now, you can place its code in a file *test_random_wikipedia_article.py* next to the module.

Example 6-2. A test for `random_wikipedia_article`

```
def test_output():
    process = subprocess.run(
        [sys.executable, "-m", "random_wikipedia_
        capture_output=True,
        check=True,
    )
    assert process.stdout
```

TIP

By convention, tests are functions whose names start with `test`. Use the built-in `assert` statement to check for expected behavior, such as the program output not being empty.

You can run the test by invoking `pipx run pytest` from the directory containing both modules. However, this won't work if your project has any third-party dependencies. Tests must be able to import your project and its dependencies, so you need to install pytest and your project in the same environment.

If you use Poetry to manage your project, add pytest to its dependencies using `poetry add`:

```
$ poetry add --group=tests pytest
```

You can now run pytest in the project environment:

```
$ poetry run pytest
========================= test session starts ===
platform darwin -- Python 3.11.3, pytest-7.3.1, p
rootdir: ...
collected 1 item

test_random_wikipedia_article.py .

========================== 1 passed in 0.01s ====
```

If you don’t use a project manager like Poetry, use a `tests` extra or a requirements file to record the dependency on pytest, as explained in “Development Dependencies”. After updating and activating your project environment, run pytest like this:

```
$ py -m pytest
```

Once your test suite consists of more than a single module, keep it under a *tests* directory. For larger projects, consider turning your test suite into a Python package that mirrors the layout of the package under test. This lets you have test modules in different sub-packages with the same name, and it gives you the option to import helper modules such as common test utilities.

Designing for Testability

Writing more fine-grained tests for the program is much harder. The API endpoint returns a random article, so *which* title and summary should the tests expect? Every invocation sends an HTTP request to the real Wikipedia API and receives the server response. Those network roundtrips will make the test suite excruciatingly slow—and you can only run tests when your machine is connected to the Internet.

Python programmers have an arsenal of tools at their disposal for situations like this. Most of these involve some form of *monkey patching*, which replaces ("patches") functions or objects at runtime to make the code easier to test. For example, you can capture program output by replacing `sys.stdout` with a file-like object that writes to an internal buffer for later inspection. You can replace `urlopen` with a function that returns canned HTTP responses of your liking. Libraries like `responses`, `respx`, or `vcr.py` provide high-level interfaces that monkey patch the HTTP machinery behind the scenes. More generic approaches use the standard `unittest.mock` module or the `pretend` library.

While these tools serve their purpose, I'd encourage you to focus on the root of the problem: Example 6-1 has no separation of concerns. A single function serves as the application entry point, communicates with an external API, and presents the results on the console. This makes it hard to test its features in isolation.

The program also lacks abstraction, in two distinct ways. First, it doesn't encapsulate implementation details when interfacing with other systems, like interacting with the Wikipedia API or writing to the terminal. Second, its central concept—the Wikipedia article—only appears as an amorphous JSON object: the program doesn't abstract its domain model in any way.

Example 6-3 shows a simple refactoring that makes the code more testable, without resorting to monkey patching or third-party libraries. While this version of the program is longer than the original one, it expresses its logic more clearly and is more amenable to change. Good tests don't just catch bugs: they improve the design of your code.

Example 6-3. Refactoring for testability

```
@dataclass
```

```
class Article:
    title: str = ""
    summary: str = ""

def fetch(url):
    with urllib.request.urlopen(url) as response
        data = json.load(response)
    return Article(data["title"], data["extract"]

def show(article, file):
    summary = textwrap.fill(article.summary)
    file.write(f"{article.title}\n\n{summary}\n"

def main():
    article = fetch(API_URL)
    show(article, sys.stdout)
```

The refactoring extracts `fetch` and `show` functions from `main`. It also defines an `Article` class as the common denominator of these functions. Let's see how these changes allow you to test the parts of the program in isolation and in a repeatable way.

For the `fetch` function, tests can set up a local HTTP server and perform a *roundtrip* check, as shown in Example 6-4: You serve an `Article` instance via HTTP, fetch the article from the server, and check that the served and fetched instances are equal.[1]

Example 6-4. Testing the `fetch` function

```
def test_fetch():
    article = Article("Lorem Ipsum", "Lorem ipsur
    with serve(article) as url:
        assert article == fetch(url)
```

The `show` function accepts any file-like object. While `main` passes `sys.stdout`, tests can pass an `io.StringIO` instance to store the output in memory. Example 6-5 uses this technique to check that the output ends with a newline. The final newline ensures the output doesn't run into the next shell prompt.

Example 6-5. Testing the `show` function

```
def test_final_newline():
    article = Article("Lorem Ipsum", "Lorem ipsur
    file = io.StringIO()
    show(article, file)
    assert file.getvalue().endswith("\n")
```

Here are some other properties of the `show` function that you might check for in your tests:

- It should include all the words of the title and summary.

- There should be a blank line after the title.
- The summary should not exceed a line length of 72 characters.

There's another benefit to the refactoring, as far as testing is concerned: The functions hide the implementation behind an interface that only involves your problem domain—URLs, articles, files. This means your tests are less likely to break when you swap out the implementations inside those functions.

Go ahead and change the program to use `httpx` and `rich` like you did in Chapter 4.[2] You won't need to adapt your tests. In fact, it's the whole point of tests to give you confidence that your program still works after making changes like this. Mocks and monkey patches, on the other hand, are brittle: They tie your test suite to implementation details, making it increasingly hard to change your program down the road.

In this chapter, I'll show you how to write tests with pytest, a popular third-party testing framework. But before we get to that, let me show you how to write tests using only the standard library. You'll understand better what exactly it is that pytest buys you, compared to an approach that doesn't involve third-party dependencies.

The unittest Framework

The standard library includes a testing framework in the `unittest` module. This object-oriented framework was inspired by JUnit, an influential Java testing library from the early days of Test Driven Development, which gave rise to many similar frameworks in other languages. Replace the module *test_random_wikipedia_article.py* with the contents of Example 6-6.

Example 6-6. Testing with unittest

```
class TestShow(unittest.TestCase):
    def setUp(self):
        self.file = io.StringIO()

    def test_final_newline(self):
        article = Article("lorem", "ipsum dolor"
        show(article, self.file)
        self.assertEqual("\n", self.file.getvalue

    def test_all_words(self):
        article = Article("lorem ipsum", "dolor s
        show(article, self.file)
        for word in ("lorem", "ipsum", "dolor", '
            self.assertIn(word, self.file.getvalu
```

As before, tests are functions whose names start with `test`, but the functions are enclosed in a test class that derives from `unittest.TestCase`. The test class has several responsibilities:

- It allows the framework to run each test.
- It allows tests to check for expected properties, using `assert*` methods.
- It allows you to prepare a test environment for each test, using the `setUp` method.

In Example 6-6, the `setUp` method initializes an output buffer for the `show` function to write to. Sometimes you need to clean up the test environment after each test; for this purpose, you can define a corresponding `tearDown` method.

Run the test suite using the command `py -m unittest` from the project directory.

```
$ py -m unittest
..
------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Thanks to the `unittest` library, you can test Python modules without taking on a third-party dependency. If you're already familiar with an JUnit-style framework from another language, you’ll feel right at home. But there are also some problems with this design. The framework forces you to place tests in a class inheriting from `unittest.TestCase`. That boilerplate hurts readability compared to a module with simple test functions. The class-based design also leads to strong coupling between test functions, the test environment, and the framework itself. Finally, every assertion method (like `assertEqual` and `assertIn` in Example 6-6) constitutes a unique little snowflake, which betrays a lack of expressivity and generality.

The pytest Framework

These days, the third-party framework `pytest` has become somewhat of a de-facto standard in the Python world. Tests written with pytest are simple and readable—you write most tests as if there was no framework, using basic language primitives like functions and assertions. At the same time, the framework is powerful and expressive, as you’ll see shortly. Finally, pytest is extensible and comes with a rich ecosystem of plugins.

A SMALL HISTORY OF PYTEST

Pytest originated in the PyPy project, an alternative, fast implementation of Python written in Python itself. Early on, there was an attempt to create a separate standard library called `std`, later renamed to `py`. While `py` is no longer actively developed, its testing module `py.test` (based on the earlier `std.utest`) became an independent thriving project under the name `pytest`.

Example 6-4 and Example 6-5 show what tests look like when written using pytest: they are simple functions whose names start with `test`. Checks are just generic assertions—pytest rewrites the language construct to provide rich error reporting in case of a test failure.

TIP

If you have a test suite written with `unittest`, there's no need to rewrite it to start using pytest—pytest "speaks" `unittest`, too. Use pytest as a test runner right away and rewrite your test suite incrementally later.

Every test for the `show` function starts by setting up an output buffer. You can use a *fixture* to remove this code duplication. Fixtures are simple functions declared with the `pytest.fixture` decorator:

```
@pytest.fixture
def file():
    return io.StringIO()
```

Tests (and fixtures) can use a fixture by including a function parameter with the same name. When pytest invokes the test function, it passes the return value of the fixture function. For example, here's Example 6-5 rewritten to use the fixture:

```
def test_final_newline(file):
    article = Article("lorem", "ipsum dolor")
    show(article, file)
    assert file.getvalue().endswith("\n")
```

The `file` fixture isn't coupled to any specific test, so it can be reused freely across the test suite. That sets it apart from the approach used in Example 6-6, where the test environment is only accessible to test methods defined in the same class or class hierarchy.

And there's another difference: test methods in a `unittest.TestCase` share a single test environment; by contrast, test functions in pytest can use any number of fixtures. For example, you could extract the test article into an `article` fixture.

If every test used the same article, you'd likely miss some edge cases, though—you don't want your program to crash if an article comes with an empty summary. Example 6-7 shows how you can run a test against a number of articles.

Example 6-7. Running tests against multiple articles

```
articles = [
    Article(),
    Article("test"),
    Article("test", "lorem ipsum dolor"),
    Article(
        "Lorem ipsum dolor sit amet, consectetur
        "Nulla mattis volutpat sapien, at dapibus
    ),
]

@pytest.mark.parametrize("article", articles)
def test_final_newline(article, file):
    show(article, file)
    assert file.getvalue().endswith("\n")
```

Advanced Techniques for Fixtures

If you want to parametrize many tests in the same way, you can create a *parametrized fixture*, a fixture with multiple values, as shown in Example 6-8. As before, pytest runs the test once for each article in `articles`.

Example 6-8. Parametrized fixture for running tests against multiple articles

```
@pytest.fixture(params=articles)
def article(request):
    return request.param

def test_final_newline(article, file):
    show(article, file)
    assert file.getvalue().endswith("\n")
```

So what did you gain here? For one thing, you don't need to decorate each test with `pytest.mark.parametrize`. And there's another advantage if your tests aren't all in the same module: You can place any fixture in a file named *conftest.py* and use it across your entire test suite without importing it.

The syntax for parametrized fixtures is somewhat arcane, though. To keep things simple, I like to define a small helper for it:

```
def parametrized_fixture(*params):
    return pytest.fixture(params=params)(lambda
    return pytest.fixture(params=params)(lambda
```

You can use the helper like this:

```
article = parametrized_fixture(Article(), Article
```

Fixtures can get large and expensive. The remainder of this section introduces some techniques that are useful when that happens:

- A *session-scoped fixture* is created only once per test run.
- Fixtures can be generators, allowing you to clean up resources after use.
- Fixtures can depend on other fixtures, making your test code more modular.
- A *factory fixture* returns a function for creating test objects, instead of the test object itself.

Example 6-4 showed a test that fetches an article from a local server. Its `serve` helper function takes an article and returns a URL for fetching the article. More precisely, it returns the URL wrapped in a *context manager*, an object for use in a `with` block. This allows `serve` to clean up after itself when you exit the `with` block—say, shut down the server.[3]

Clearly, firing up and shutting down a web server for every test is quite expensive. Would it help to turn the server into a fixture? At first glance, not much—every test gets its own instance of a fixture. However, you can instruct pytest to create a fixture only once during the entire test session:

```
@pytest.fixture(scope="session")
def httpserver():
    ...
```

That looks more promising, but how do you shut down the server when the tests are done with it? Up to now, your fixtures only needed to prepare a test object and return it. You can't run code after a `return` statement. You *can* run code after a `yield` statement, however—and so pytest allows you to define a fixture as a generator. Example 6-9 shows the resulting `httpserver` fixture.

Example 6-9. The `httpserver` fixture

```
@pytest.fixture(scope="session")
def httpserver():
    with http.server.HTTPServer(("localhost", 0),
        thread = threading.Thread(target=server.s
        thread.start()

        yield server
        server.shutdown()
        thread.join()
```

I've omitted the actual request handling for brevity—let's assume the server response returns `server.article` in its body.[4] There's still a missing piece, though: How would you define the `serve` function, now that it depends on a fixture to do its work?

You can access a fixture from a test and from another fixture. Defining `serve` inside `test_fetch` does little to simplify the test. So let's define `serve` inside its own fixture—after all, fixtures can return any object, including functions. Example 6-10 shows what that looks like in practice.

Example 6-10. The `serve` fixture

```
@pytest.fixture
def serve(httpserver): ❶
    def f(article): ❷
        httpserver.article = article
        return f"http://localhost:{httpserver.se
    return f
```

❶ The outer function defines a `serve` fixture, which depends on `httpserver`.

❷ The inner function is the `serve` function you call in your tests.

The `serve` function no longer returns a context manager, just a plain URL—the `httpserver` fixture handles all of the setting up and tearing down. As a result, you can simplify the test case quite a bit (Example 6-11). It's truly a roundtrip test now!

Example 6-11. Testing the `fetch` function using fixtures

```
def test_fetch(article, serve):
    assert article == fetch(serve(article))
```

Extending pytest with Plugins

One of the best things about pytest is the ecosystem that has evolved around it. Thanks to pytest's extensible design, anybody can write a pytest plugin and publish it to PyPI; see [Link to Come] for a minimal example. If you're interested in writing your own plugin, take a look at the `cookiecutter-pytest-plugin` template to start out with a solid project structure.

Pytest plugins perform a variety of functions (see Table 6-1). These include executing tests in parallel or in random order, presenting or reporting test results in a custom way, or integrating with frameworks and other tools. Many plugins provide useful fixtures, to interact with external systems, say, or to create *test doubles*, which is the umbrella term for the various kinds of objects tests use in lieu of the real objects used in production code.

Table 6-1. A Selection of Pytest Plugins

Category	Plugins
Execution	`pytest-randomly`, `pytest-reverse`, `pytest-fast-first`, `pytest-repeat`, `pytest-xdist`, `pytest-timeout`, `pytest-instafail`, `pytest-picked`, `pytest-watch`, `pytest-rerunfailures`
Presentation and Reporting	`pytest-rich`, `pytest-sugar`, `pytest-icdiff`, `pytest-spec`, `pytest-html`, `pytest-reportlog`, `pytest-duration-insights`
Frameworks	`pytest-flask`, `pytest-django`, `pytest-asyncio`, `pytest-twisted`, `pytest-anyio`
Fake Servers	`pytest-httpserver`, `pytest-server-fixtures`, `pytest-devpi-server`, `pytest-pyramid-server`
Fake Data	`pytest-faker`, `pytest-factoryboy`, `pytest-mimesis`

Category	Plugins
Mocks	`pytest-mock`, `respx`, `pytest-responses`, `pytest-vcr`, `pytest-freezegun`
Filesystem and Storage	`pytest-datadir`, `pytest-git`, `pytest-svn`, `pytest-virtualenv`
Tooling	`pytest-cov`, `pytest-selenium`, `pytest-profiling`, `pytest-monitor`
Documentation	`xdoctest`, `pytest-codeblocks`, `pytest_docfiles`
Snapshot testing	`pytest-pinned`
Configuration	`pytest_profiles`
Extra Checks	`pytest-checkipdb`, `pytest-modified-env`, `typeguard`

Let's use the `pytest-httpserver` plugin to implement the `serve` fixture from Example 6-10. First, add the plugin to your development dependencies. For example, here's how you would add the plugin to a Poetry-managed project:

```
$ poetry add --group=tests pytest-httpserver
```

Next, remove the existing `httpserver` fixture; the plugin provides a fixture with the same name. Finally, modify the `serve` fixture as shown in Example 6-12.

Example 6-12. The `serve` fixture using `pytest-httpserver`

```
@pytest.fixture
def serve(httpserver):
    def f(article):
        handler = httpserver.expect_request("/")
        handler.respond_with_json(
            {"title": article.title, "extract": a
        )
        return httpserver.url_for("/")
    return f
```

As you can see, plugins can save you much work in testing your code. In this case, you've been able to test the `fetch` function without implementing your own test server.

Summary

In this chapter, you've learned how to use pytest to test your Python projects.

In pytest, tests are functions that exercise your code and check for expected behavior using the `assert` builtin. Prefix their name—and the name of the containing modules—with `test`, and pytest will discover them automatically. Fixtures are reusable functions or generators that set up and tear down test objects. Use them in a test by including a parameter named like the fixture. Plugins for pytest can provide useful fixtures, as well as modify test execution, enhance reporting, and much more.

It's a prime characteristic of good software that it's easy to change. Any piece of code used in the real world must adapt to evolving requirements and an ever-changing environment. Tests promote ease of change in several ways:

- They drive software design towards loosely coupled building blocks that you can test in isolation: fewer inter-dependencies also mean fewer barriers to change.
- Tests document and enforce expected behavior. That gives you the freedom and confidence to continuously refactor your codebase—keeping it maintainable as it grows and transforms.
- Last but not least, tests reduce the cost of change by detecting defects early. The earlier you catch an issue, the cheaper it gets

> to work out the root cause and develop a fix. Conversely, there's no more expensive way to uncover a bug than shipping it to production (and hearing about it from your users).

But how do you know that your test suite is, in fact, up to this task? When all tests pass, how confident are you that the latest change won't break the world? You might call this the *sensitivity* of the test suite: the probability of a test failure when there's a defect in the code.[5] For example, did you cover all the edge cases of each function or did you test only the "happy paths" of your program?

There are really several aspects to this question of completeness. One aspect is that your test suite should capture your expectations towards the code. A great way to ensure this is to always write a test first when you add or change behavior, before the code that makes it pass. Another aspect is that you should test your software with the various inputs and environmental constraints that you expect it to encounter in the real world—a notoriously hard problem. And finally, the test suite should exercise all of your code: every line of source code and every branch in its control flow.

It turns out that this last aspect, known as *test coverage*, is one that you can measure quantitatively, and it's the subject of the Chapter 7.

Don't worry about the `serve` function for now—you'll implement it later.

You can pass the file-like object to `Console` using its `file` parameter.

You can implement the function yourself using `http.server` from the standard library, as well as the `threading` module and the `@contextmanager` decorator from `contextlib`.

If you'd like to try this yourself, derive a class from `http.server.BaseHTTPRequestHandler` and pass it to `HTTPServer`. Its `do_GET` method needs to write a response with `server.article` in JSON format.

There's also the related question of *specificity*, the probability of tests passing if the code is free of defects. One example for low specificity is flakiness—tests that fail intermittently due to external factors, like connectivity, timing, or system load. Another example are tests that break when you change implementation details, even though the software behaved as expected.

Chapter 7. Measuring Coverage with Coverage.py

Coverage tools record each executed statement while you run your code. After completion, they report the overall percentage of executed statements with respect to the entire codebase. You can use this measurement as a reasonable proxy for the completeness of your test suite, an upper bound: If your tests cover 100% of your code, you may not be guaranteed that the code is free of bugs. However, if the tests cover any less than that, they will definitely not detect any bugs in the parts without test coverage. Therefore, you should check that all code changes come with adequate test coverage.

How does all of this work in Python? The interpreter allows you to register a callback—a *trace function*—using the function `sys.settrace` from the standard library. From that point onwards, the interpreter invokes the callback whenever it executes a line of code—as well as in some other situations, like entering or returning from functions or raising exceptions. Coverage tools register a trace function that records each executed line of source code in a local database.

By the way, coverage tools are not limited to measuring test coverage, even though it's their primary purpose. They can also help you determine which modules in a large codebase are used by each endpoint of an API. Or you could use them to find out how much of your project is documented in code examples. What they do is simple: they record each line in your source code when you run it, be that from test cases or in another way.

The standard trace module

Python's standard library includes a coverage tool in the `trace` module. Let's use it to measure test coverage for the `random-`

wikipedia-article project. First, determine where your standard library and third-party libraries live, so you can exclude them from the measurement using the --ignore-dir option. Activate the project environment and install the project in editable mode, for example using the commands poetry shell and poetry install. Finally, run the test suite via trace, as shown below:[1]

```
$ py -m trace --module pytest --count --missing
--ignore-dir ~/Library
========================= test session starts ===
platform darwin -- Python 3.11.3, pytest-7.3.1, p
rootdir: ...
plugins: httpserver-1.0.8, anyio-3.7.0
collected 21 items

test_random_wikipedia_article.py ...............

========================== 21 passed in 1.54s ===
lines   cov%   module   (path)
   32    84%   random_wikipedia_article   (...)
   50    94%   test_random_wikipedia_article   (
```

If you're curious why the trace module reports less than 100% coverage, take a look at the generated **.cover* files, next to the respective source files. They contain the source code annotated with execution counts; missing lines are marked by the string `>>>>>>`. But don't worry about those missing lines too much now, you'll find out all about them in a bit.

As you can see, measuring coverage using the standard library alone is quite cumbersome, even for a simple use case as this. While the `trace` module is interesting as an early proof of concept, I don't recommend using it for any real world project. Instead, you should use the third-party package `coverage`.

Using Coverage.py

Coverage.py is a mature, widely use, and feature-complete code coverage tool for Python. Add `coverage` to your test dependencies as shown below.[2] The `toml` extra allows `coverage` to read its configuration from *pyproject.toml* on Python versions without TOML support in the standard library (before 3.11).

```
$ poetry add --group=tests coverage[toml]
```

In the simplest case, measuring code coverage is a two-step process. First, you run a program while gathering coverage data (`coverage run`). For test coverage, this means running the test suite under `coverage`. Second, you compile an aggregated report from the coverage data (`coverage report`).

You can invoke `coverage run` with any Python script, followed by its command-line arguments. Alternatively, you can use its `-m` option with an importable module, similarly to how you'd run `py -m pytest`. This second method ensures that you run pytest from the current environment, so let's use it here:

```
$ poetry run coverage run -m pytest
```

This command creates a file *.coverage* in the current directory. Under the hood, this file is just a SQLite database, so feel free to poke around if you have the `sqlite3` tool ready on your system.

The coverage report includes the overall percentage of code coverage and a break-down per source file. If you specify the `--show-missing` option, the report also lists the individual statements that are missing from coverage, identified by line number.

```
$ poetry run coverage report --show-missing
Name                                   Stmts   Miss
---------------------------------------------------
random_wikipedia_article.py               32      5
test_random_wikipedia_article.py          37      0
---------------------------------------------------
```

```
TOTAL                                         69       5
```

As before, the `random_wikipedia_article` module has a coverage of 84%, because 5 of the 32 statements in that module never showed up during the tests. By contrast, `coverage` reports 100% coverage for the test module itself, as you would expect. (This result differs from the earlier one reported by `trace`, which didn't correctly account for a list comprehension in a test case.)

TIP

Measuring code coverage for your test suite may seem strange—but you should always do it. It ensures that you notice when tests aren't run by mistake, and it can help you identify unreachable code in the tests. This boils down to a more general piece of advice: Treat your tests the same way you would treat any other code.

Let's add the `--show-missing` option to the tool configuration, so you don't have to specify it every time. Each `coverage` subcommand has its own subtable under `tool.coverage` in *pyproject.toml*, so the option corresponds to the following configuration setting:

```
[tool.coverage.report]
show_missing = true
```

If your project consists of more than a single Python module, you should also specify your top-level import package in the configuration. This allows `coverage` to report modules even if they're missing from coverage entirely, rather than just those that showed up during execution. If your tests are organized in a `tests` package with multiple test modules, list that package as well:

```
[tool.coverage.run]
source = ["random_wikipedia_article", "tests"]
```

Measuring across Multiple Environments

Let's take a closer look at those missing statements now. The `Missing` column in the coverage report lists them by line number. You can use your code editor to display the source code with line numbers, or the standard `cat -n` command on Linux and macOS. Here are the first two missing lines:

```
try:                                              #  7
    from importlib.metadata import metadata       #  8
except ImportError:                               #  9
    from importlib_metadata import metadata       # 1
```

Here, the coverage report tells you that you never tested the program with the `importlib_metadata` backport. This isn't a shortcoming of your test suite, but it *is* a shortcoming of how you're running it. You need to test your program on all supported Python versions, including Python 3.7, which doesn't have `importlib.metadata` in the standard library.

Let's run the tests on Python 3.7. If you added `python-httpserver` in "Extending pytest with Plugins", you'll need to revert back to the home-grown `httpserver` fixture, as the plugin doesn't support Python 3.7. Next, switch to an environment with Python 3.7 and install the project:

```
$ poetry env use 3.7
$ poetry install
```

By default, `coverage run` overwrites any existing coverage data, but you can tell it to append instead using the `--append` option:

```
$ poetry run coverage run --append -m pytest
$ poetry run coverage report
```

You should no longer see lines `9` and `10` reported as missing, which confirms that the tests indeed ran against the backport instead of the standard library.

If you look closely at the `try...except` statement above, you may notice that running on Python 3.7 alone would have been enough—the old Python version exercises all of the lines, because it has to try both imports in turn.

But in a certain sense, coverage *is* incomplete on Python 3.7: The tests never exercise the code path where control passes from the `try` block directly to the following code, skipping over the `except` handler. You can tell coverage to capture the control flow of your program, by measuring not isolated lines of source code, but transitions from one line to the next. This type of code coverage is known as *branch coverage*, and since it is more precise than its counterpart (*statement coverage*), you should enable it by default:

```
[tool.coverage.run]
branch = true
```

Parallel Coverage

With a single coverage data file, it's easy to erase data accidentally by omitting the `--append` option. You could configure `coverage run` to append by default, but that's error-prone, too: If you forget to run `coverage erase` periodically, you end up with stale data in your report.

There's a better way to gather coverage across multiple environments. The `coverage` tool allows you to record coverage data in separate files for each run. The option for enabling this behavior is named `--parallel`. (The option name is somewhat misleading; it has nothing to do with parallel execution.) If your tests run on more than a single Python version—or in more than a single process, as I'll explain below—it's a good idea to enable parallel mode by default in *pyproject.toml*:

```
[tool.coverage.run]
parallel = true
```

Even in parallel mode, coverage reports are based on a single data file. Before reporting, you'll therefore need to merge the data files using the command `coverage combine`. That changes the two-step process from above into a three-step one: `coverage run` — `coverage combine` — `coverage report`.

Let's put all of this together then. First, you run the test suite on each supported Python version. For brevity, I'm only showing Python 3.7 and 3.11 here. I'm also omitting `poetry install` since you've already installed the project into those environments.

```
$ poetry env use 3.7
$ poetry run coverage run -m pytest
$ poetry env use 3.11
$ poetry run coverage run -m pytest
```

At this point, you'll have multiple data files in your current directory. Their names start with *.coverage*, followed by the machine name, process ID, and a random number. The command `coverage combine` aggregates those files into a single *.coverage* file. By default, it also removes the individual files.

```
$ poetry run coverage combine
Combined data file .coverage.Claudios-MBP.26719.(
Combined data file .coverage.Claudios-MBP.26766.:
```

If you run `coverage report` again, you'll notice that lines `9` and `10` are no longer missing:

```
$ poetry run coverage report
Name                                  Stmts   Miss
```

```
-----------------------------------------------
random_wikipedia_article.py           32       3
test_random_wikipedia_article.py      37       0
-----------------------------------------------
TOTAL                                 69       3
```

Measuring in Subprocesses

The remaining missing lines in the coverage report correspond to the body of the `main` function, and its invocation at the end of the file. This is surprising—the end-to-end test from Example 6-2 runs the entire program, so all of those lines are definitely being tested.

If you think about how coverage measurement works, you can maybe guess what's going on here. The end-to-end test runs the program in a separate process, on a separate instance of the Python interpreter. In that process, `coverage` never had the chance to register its trace function, so none of those executed lines were recorded anywhere. Fortunately, the `coverage` tool provides a public API to enable tracing for the current process: the `coverage.process_start` function.

You could invoke this function from inside `random_wikipedia_article`, but it would be better if you didn't have to modify your application to support code coverage. As it turns out, a somewhat obscure feature of Python environments allows you to invoke the function during interpreter startup (see [Link to Come]). The interpreter executes lines in a *.pth* file in the site directory, as long as they start with an `import` statement. This means that you can activate coverage by installing a *coverage.pth* file into the environment, with the following contents:

```
import coverage; coverage.process_start()
```

Install this file to the *site-packages* directory of your project environment. You can determine its precise location like this:

```
$ poetry run python -c 'import sysconfig; print(
```

There's still a missing piece, though: You need to communicate to `coverage` where its configuration file lives, using the environment variable `COVERAGE_PROCESS_START`. Here's an example using the Bash shell:

```
$ export COVERAGE_PROCESS_START=pyproject.toml
```

For Powershell, use the following syntax instead:

```
> $env:COVERAGE_PROCESS_START = 'pyproject.toml'
```

If you re-run the test suite, the coverage report should now consider the program to have full coverage:

```
$ poetry run coverage report
Name                                  Stmts   Miss
--------------------------------------------------
random_wikipedia_article.py              32      0
test_random_wikipedia_article.py         37      0
--------------------------------------------------
TOTAL                                    69      0
```

By the way, this only worked because you enabled parallel coverage earlier. Without it, the main process would overwrite the coverage data from the subprocess, since both would be using the same data file.

At this point, you probably think that this is way too much work for all but the largest projects. If you had to take these steps manually each time, I'd agree. Bear with me though until Chapter 7, where I'll explain how to automate testing and coverage reporting with Nox. Automation can give you the full benefit of strict checks at minimal cost.[3]

Setting a Coverage Target

Any coverage percentage under 100% means that some of your source code doesn't run during the tests. As a consequence, your tests won't detect bugs in those parts. You can, and generally should, configure the `coverage report` step to fail if the percentage is below 100%:

```
[tool.coverage.report]
fail_under = 100
```

If you're working on a new project, there really isn't any meaningful coverage target other than 100%. That doesn't mean that you should test every single line of source code, no matter the cost. For example, you might have a log statement in your code to debug a rarely occurring situation. The log statement may be difficult to exercise from a test; at the same time, log statements are typically low-risk trivial code, so writing that test won't give you much additional confidence. In this case, you can exclude the line from coverage using a special comment:

```
if rare_condition:
    print("got rare condition")  # pragma: no cov
```

When you decide to exclude code from coverage, base your decision on the cost-benefit ratio of writing a test, not merely on how cumbersome testing would be. When you start working with a new library or interfacing with a new system, it can be hard to figure out how to test your code. Time and again, I've found it paid off to write the difficult test, or to refactor to make testing easier. Time and again, those tests ended up detecting bugs that would likely have gone unnoticed and caused problems in production.

Legacy projects often consist of a large codebase with minimal test coverage. As a general rule, coverage in such projects should monotonically increase—changes shouldn't lead to a drop in coverage. You'll often find yourself in a dilemma here: To test, you need to refactor the code; but refactoring is too risky without tests. Your first step should be to find the minimal, safe refactoring to increase testability. Often, this will consist of breaking a dependency of the code under test. For example, if you need to test a function that, among many things, connects to the production database, consider adding an optional parameter so you can pass the connection from the outside when testing.

Summary

You can measure the extend to which the test suite exercises your project using the `coverage` tool. This is particularly useful to discover edge cases for which you don't have a test. Branch coverage captures the control flow of your program, instead of just isolated lines of source code. Parallel coverage allows you to measure coverage across multiple environments; you'll need to combine the data files before reporting. Measuring coverage in subprocesses requires setting up a *.pth* file and an environment variable.

Measuring test coverage effectively for a project requires some amount of configuration, as well as the right tool incantations. In the next chapter, you'll see how you can automate these steps with Nox. You'll set up checks that give you confidence in your changes, while staying out of your way most of the time.

1 Enter the command on a single line.

2 Don't forget to quote the square brackets if you're a zsh user, as they're special characters in that shell.

3 The widely used pytest plugin `pytest-cov` aims to run `coverage` in the right way behind the scenes. After installing the plugin, run pytest with the `--cov` option to enable the plugin. You can still configure coverage in *pyproject.toml*. Subprocess coverage works out of the box, as well as some other forms of parallel execution. On the other hand, while trying to be helpful, the plugin also adds a layer of in-

direction. You may find that running `coverage` directly gives you more fine-grained control and a better understanding of what's going on under the covers.

About the Author

Claudio Jolowicz is a software engineer with 15 years of industry experience in C++ and Python, and an open-source maintainer active in the Python community. He is the author of the Hypermodern Python blog and project template, and co-maintainer of Nox, a Python tool for test automation. In former lives, Claudio has worked as a lawyer and as a full-time musician touring from Scandinavia to West Africa. Get in touch with him on Twitter: @cjolowicz