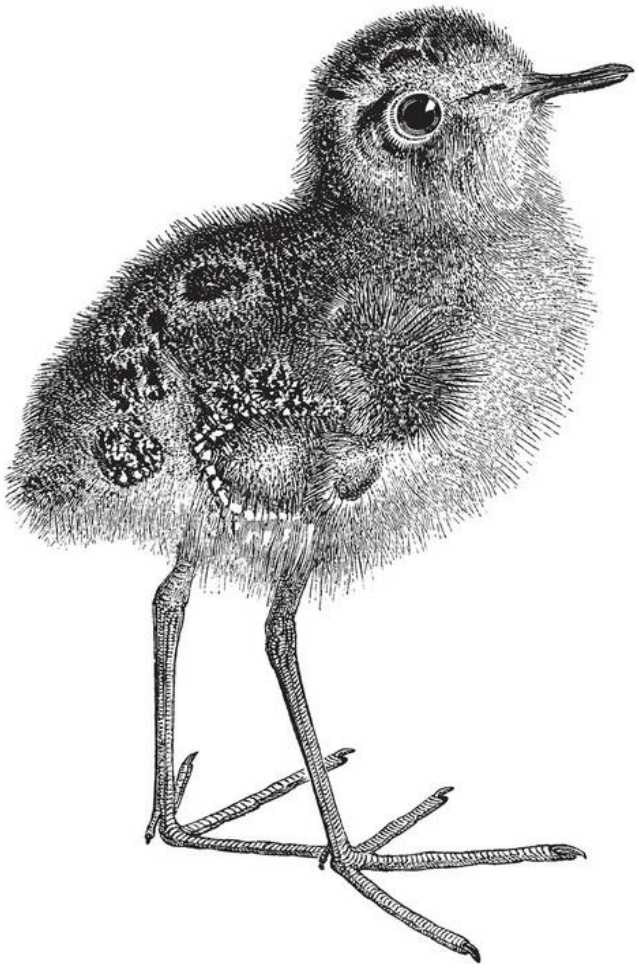
O'REILLY®

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

Hypermodern Python Tooling

by Claudio Jolowicz

Printed in the United States of America.

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

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`.

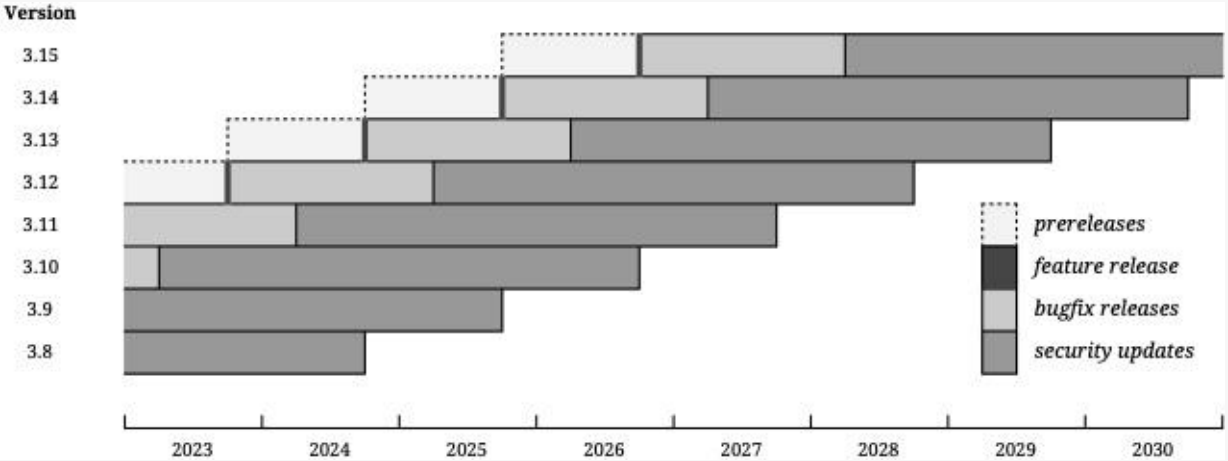


Figure 1-1. Timeline of Python Releases

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 tells the shell to remove duplicate entries from the search path.

❷ This inserts the new directory at the beginning of the search path.

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.

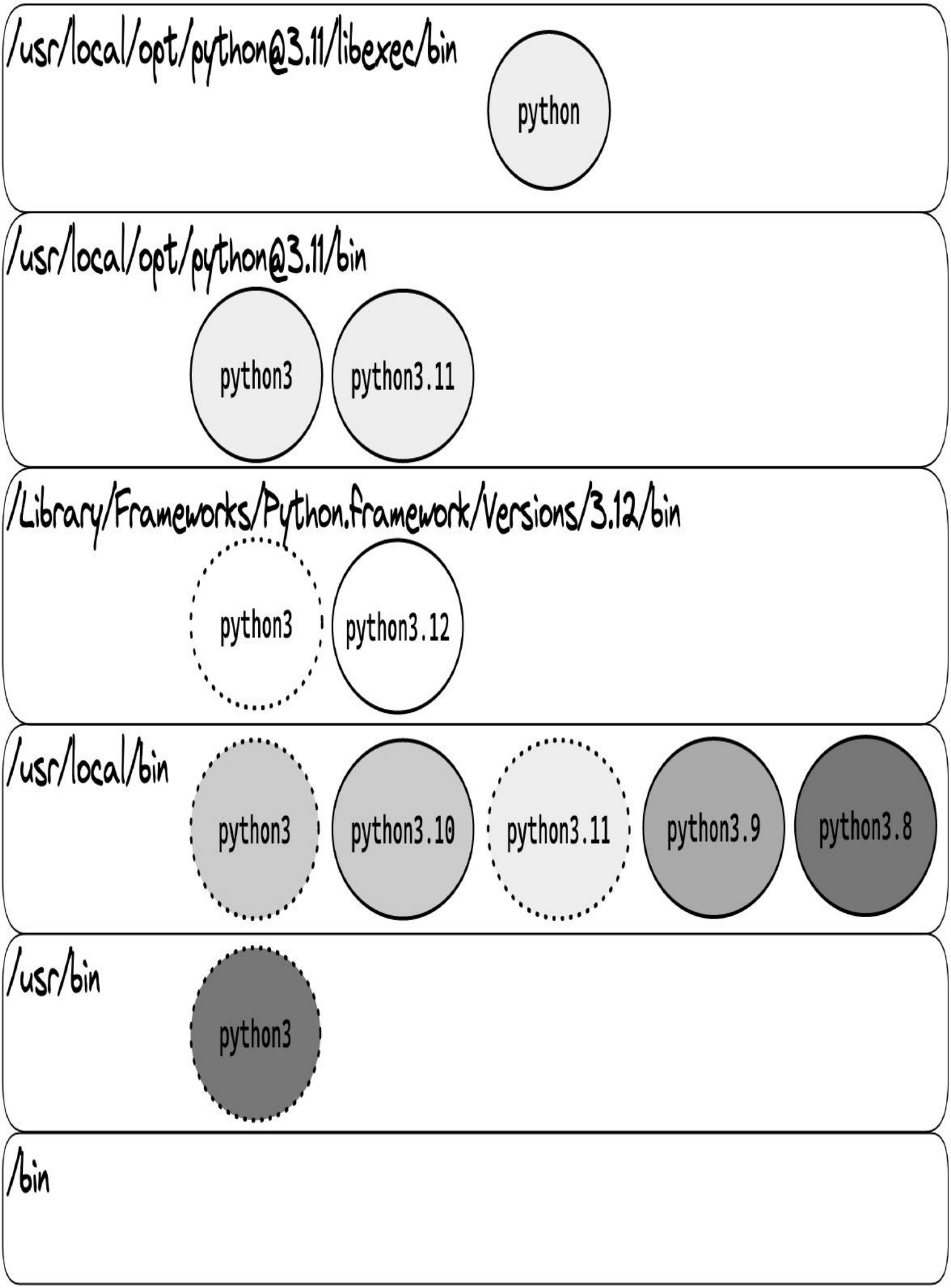

/usr/local/opt/python@3.11/libexec/bin
python
/usr/local/opt/python@3.11/bin
python3
python3.11
/Library/Frameworks/Python.framework/Versions/3.12/bin
python3
python3.12
/usr/local/bin
python3
python3.10
python3.11
python3.9
python3.8
/usr/bin
python3
/bin

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, 16:14:13) [...] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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, 16:36:42) [...] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

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 [Link to Come]); 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-properties-common
$ sudo add-apt-repository ppa:deadsnakes/ppa && sudo apt update
```

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/python3.10
 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 `python`*`X.Y`* commands; in other words, invoking `py -`*`X.Y`* is equivalent to running `python`*`X.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="$HOME/.pyenv/versions/3.x.y/bin:$PATH"
```

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-optimizations --with-lto'
```

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.

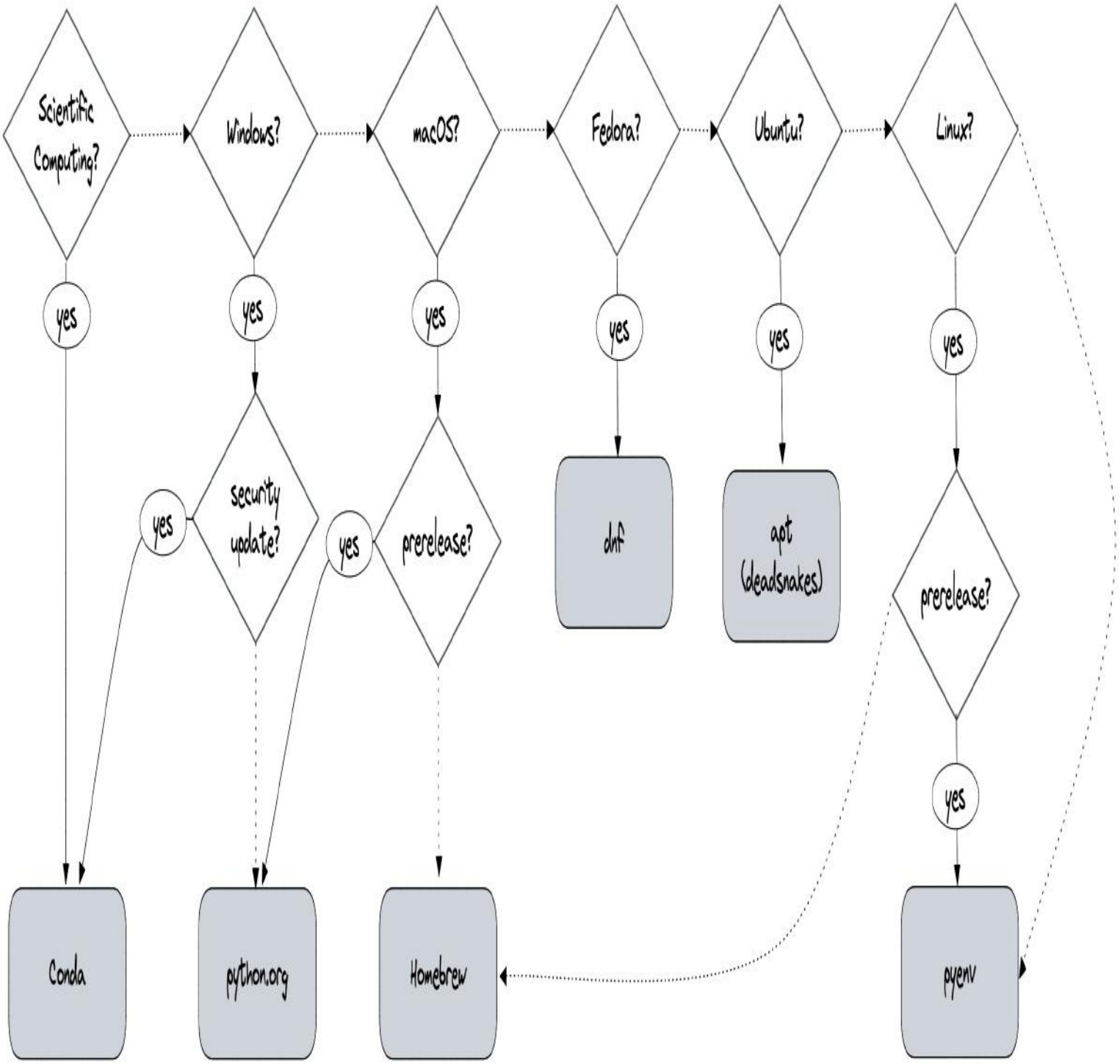


Figure 1-3. Choosing an installation method for Python

---

1 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.

2 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.)

3 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.

4 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.

5 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*.

6 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

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 second 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.

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.

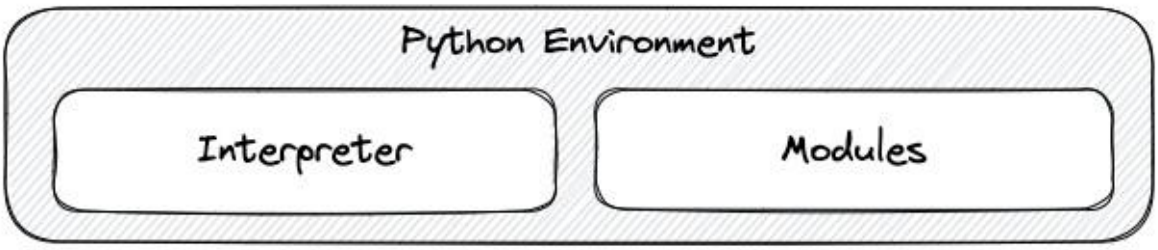


Figure 2-1. The essential components of a Python environment

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.

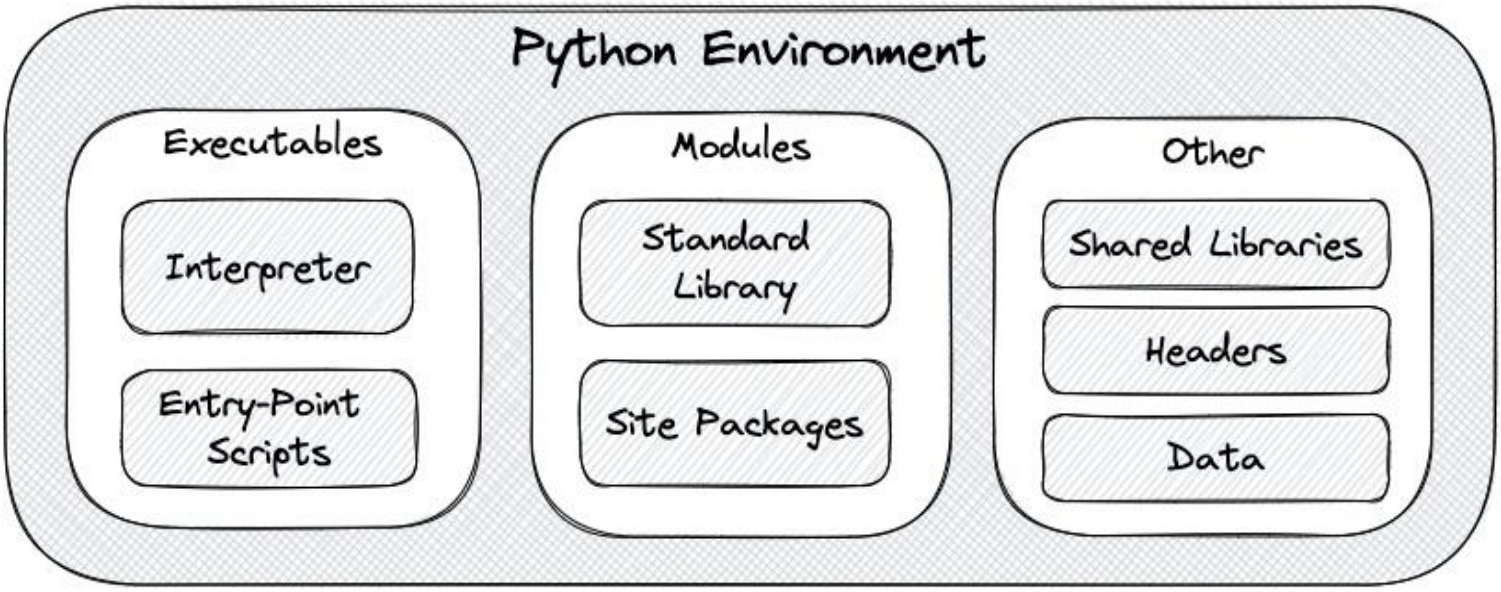


Figure 2-2. Python environments consist of an interpreter, the standard library, site packages, entry-point scripts, shared libraries, and more.

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

```
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=lambda d: d.name):
    print(f"{distribution.name:30} {distribution.version}")
```

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
```

[illegible] shebang.

❹

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/Versions/3.x*[a]
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*.
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, expand=False)
        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/python3.x/site-packages*[a]
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
```

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 RemoteSigned -Scope CurrentUser
```

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` subcommand. 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.

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://example.com
```

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.
`__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.

3 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.

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

5 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.

6 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.

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

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

Chapter 3. Python Packages

> **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 third 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.

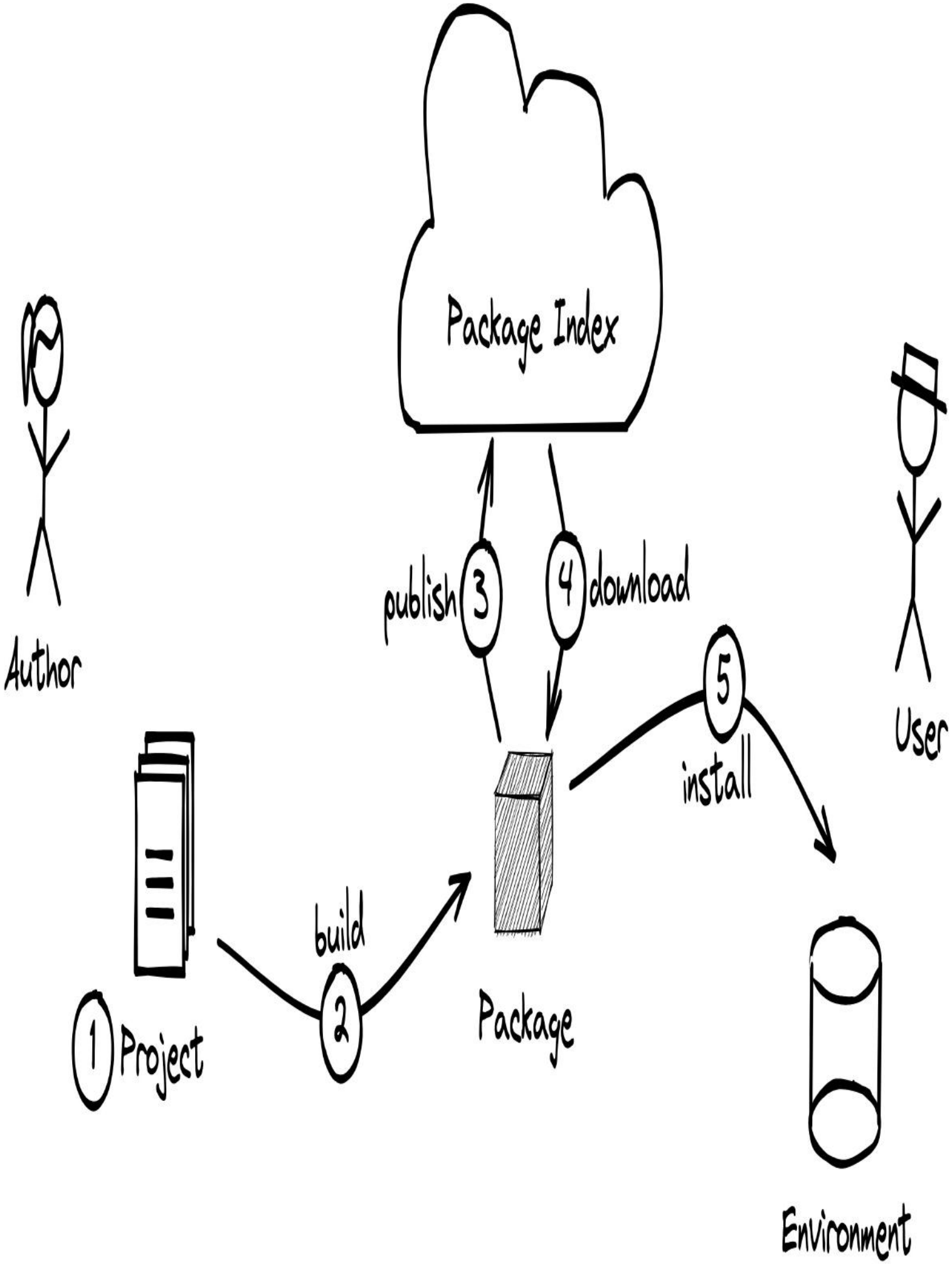
In this chapter you'll learn how to package your Python projects for distribution. A *package* is a single file containing an archive of your code along with metadata that describes it, like the project name and version. You can install this file into a Python environment using Pip, the Python package 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.


Package Index
Author
User
publish
3
4
download
5
install
build
2
1
Project
Package
Environment

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/page/random/summary" ❶

def main():
    with urllib.request.urlopen(API_URL) as response: ❷
        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 central Germany
is a storage pond near the town of Clausthal-Zellerfeld in the
county of Goslar in Lower Saxony. It is one of the Upper Harz Ponds
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 in a Nutshell

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.

Example 3-2 shows how to package the script from "An Example Application" with the bare minimum of project metadata. Place the script and this 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"
```

> **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.

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-wikipedia-article
  Building wheel for random-wikipedia-article (pyproject.toml) ... done
  Created wheel for random-wikipedia-article: …
  Stored in directory: …
Successfully built random-wikipedia-article
Installing collected packages: random-wikipedia-article
Successfully installed random-wikipedia-article-0.1
```

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 project.toml file with an entry-point script

```
[project]
name = "random-wikipedia-article"
version = "0.1"
```

```
[project.scripts]
random-wikipedia-article = "random_wikipedia_article:main"

[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, installed using Python 3.10.8
  These apps are now globally available
    - random-wikipedia-article
done!
```

You can now invoke the script directly:

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

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... (hatchling)
* Getting build dependencies for sdist...
* Building sdist...
* Building wheel from sdist
* Creating venv isolated environment...
* Installing packages in isolated environment... (hatchling)
* Getting build dependencies for wheel...
* Building wheel...
Successfully built random_wikipedia_article-0.1.tar.gz
 and random_wikipedia_article-0.1-py2.py3-none-any.whl
```

`Build` places the packages in the *dist* directory of your project. Let's conclude this little tour of Python packaging by publishing them 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 dist/*
Uploading distributions to https://test.pypi.org/legacy/
Enter your username: __token__
Enter your password:
Uploading random_wikipedia_article-0.1-py2.py3-none-any.whl
Uploading random_wikipedia_article-0.1.tar.gz
```

```
View at:
https://test.pypi.org/project/random-wikipedia-article/0.1/
```

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/simple random-wikipedia-article
  installed package random-wikipedia-article 0.1, installed using Python 3.10.8
  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 [Link to Come].

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:

```
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:

```
[project]
name = "foo"
```

```
version = "0.1"
```

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

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

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

```
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.

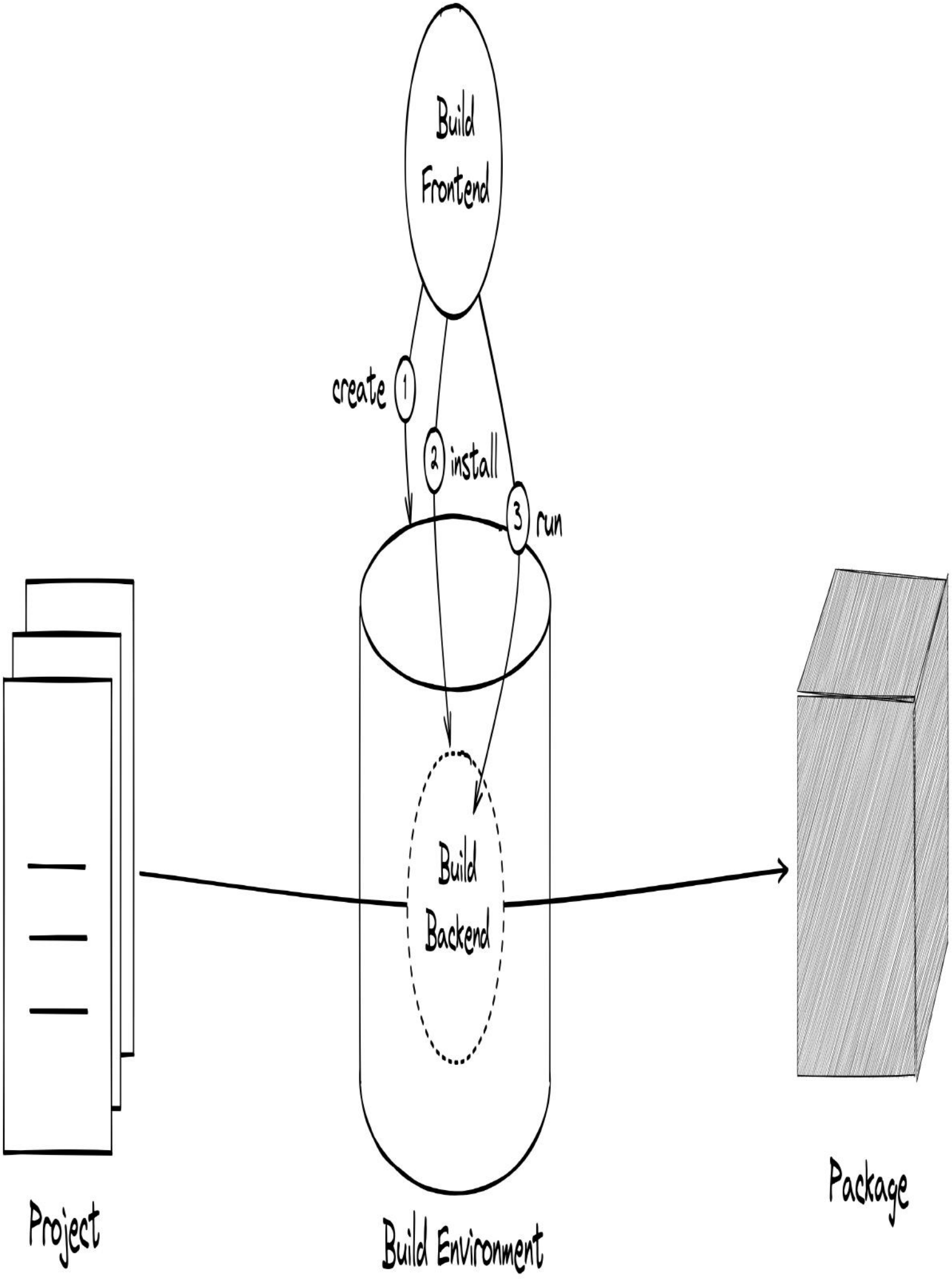

Build
Frontend
create
1
2
install
3
run
Build
Backend
Project
Build Environment
Package

Figure 3-2. Build Frontend and Build Backend

In Example 3-2, I've opted for `hatchling` as the build backend. It comes with Hatch, a modern and standards-compliant Python project manager. 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.whl'
>>>
$ 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.

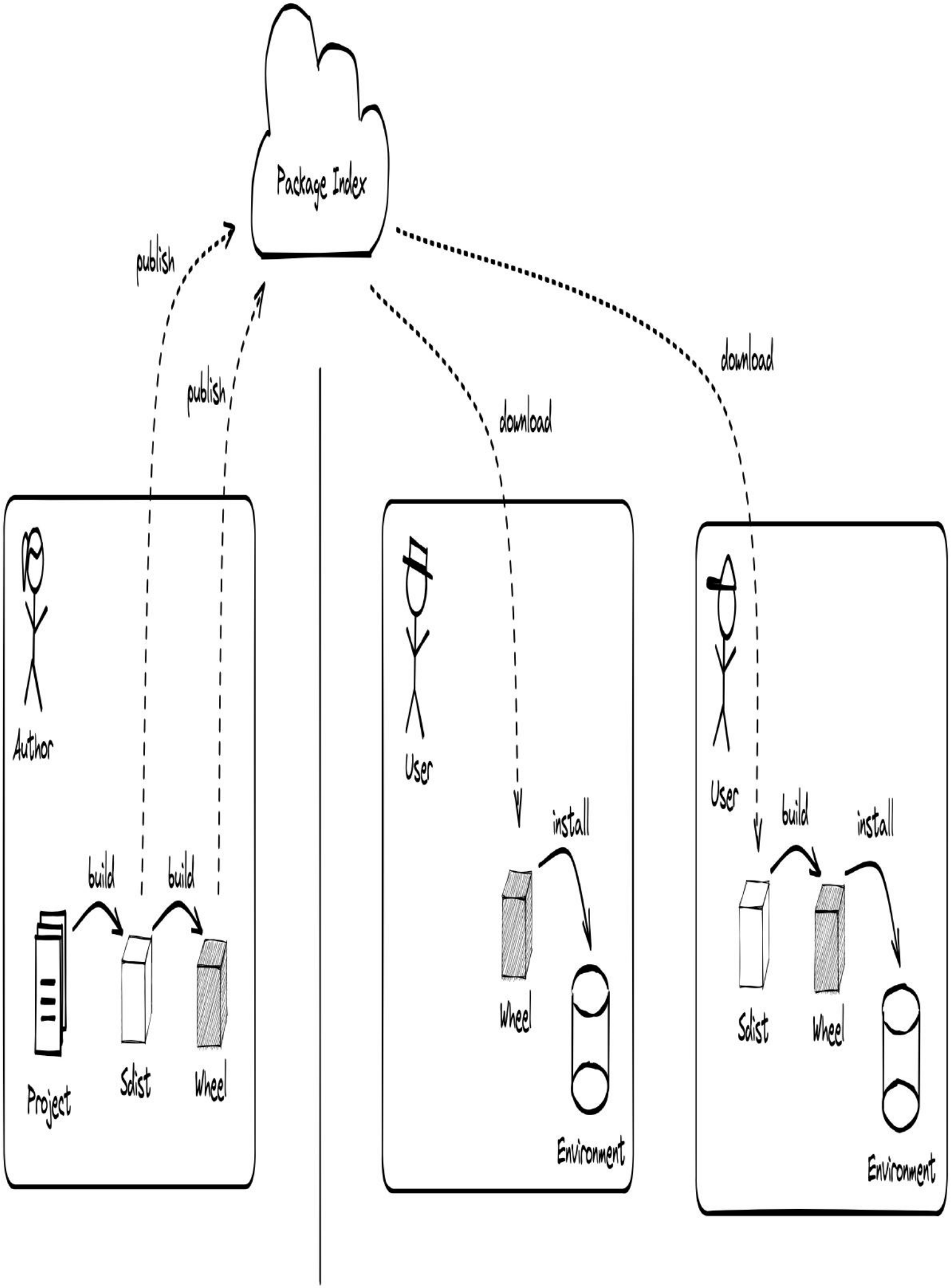

Package Index
publish
publish
download
download
Author
build
build
Project
Sdist
Wheel
User
install
Wheel
Environment
User
build
install
Sdist
Wheel
Environment

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
`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 Wikipedia articles"
keywords = ["wikipedia"]
readme = "README.md"
license = { text = "MIT" }
authors = [{ name = "Your Name", email = "you@example.com" }]
classifiers = ["Topic :: Games/Entertainment :: Fortune Cookies"]
```

```
urls = { Homepage = "https://yourname.dev/projects/random-wikipedia-article" }
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.

```
[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	`1.0.0dev1`
Post-release	Corrects a minor error outside of the code	`1.0.0post1`

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.

```
[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 tells Hatch to look for the `version` field dynamically.

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:

```
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 3-6).

Example 3-6. 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_article:main"
```

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_article:gui_main"
```

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 [Link to Come].)

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_article"
```

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 3-7.

Example 3-7. Test fixture with a random Wikipedia article

```
import json
import urllib.request

import pytest

API_URL = "https://en.wikipedia.org/api/rest_v1/page/random/summary"

@pytest.fixture
def random_wikipedia_article():
    with urllib.request.urlopen(API_URL) as response:
        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 3-8). Pytest recognizes that the function argument is a test fixture and invokes the test function with the return value of the fixture.

Example 3-8. A test function that uses the random article fixture

```
# test_wikipedia_viewer.py
def test_wikipedia_viewer(random_wikipedia_article):
    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, pluggy-1.0.0
rootdir: ...
plugins: random-wikipedia-article-0.1
collected 1 item

test_wikipedia_viewer.py F                                               [100%]
```

```
================================== FAILURES ===================================
_____________________________ test_wikipedia_viewer _____________________________

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

test_wikipedia_viewer.py:4: AssertionError
---------------------------- Captured stdout call ----------------------------
Halgerda stricklandi
Halgerda stricklandi is a species of sea slug, a dorid nudibranch, a shell-less
marine gastropod mollusk in the family Discodoridae.
========================== short test summary info ===========================
FAILED test_wikipedia_viewer.py::test_wikipedia_viewer - assert False
============================== 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.

```toml
[project]
authors = [{ name = "Your Name", email = "you@example.com" }]
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.

```toml
[project]
description = "Display extracts from random Wikipedia articles"
```

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/plain" }
```

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.

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

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

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

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`
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-wikipedia-article"
Source = "https://github.com/yourname/random-wikipedia-article"
Issues = "https://github.com/yourname/random-wikipedia-article/issues"
Documentation = "https://readthedocs.io/random-wikipedia-article"
```

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 License"]
```

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.

```
[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]

```
[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 [Link to Come]).

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.

2 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`.

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

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

5 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).

6 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.

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