Chapter 24. Packaging Programs and Extensions

In this chapter, abridged for print publication, we describe the packaging ecosystem's development. We provide additional material in the **online version of this chapter***, available in the GitHub repository for this book. Among other topics (see* **"Online Material"** *for a complete list), in the online version we describe* `poetry`*, a modern standards-compliant Python build system, and compare it with the more traditional* `setuptools` *approach.*

Suppose you have some Python code that you need to deliver to other people and groups. It works on your machine, but now you have the added complication of making it work for other people. This involves packaging your code in a suitable format and making it available to its intended audience.

The quality and diversity of the Python packaging ecosystem have greatly improved since the last edition, and its documentation is both better organized and much more complete. These improvements are based on careful work to specify a Python source tree format independent of any specific build system in **PEP 517**, "A Build-System Independent Format for Source Trees," and the minimum build system requirements in **PEP 518**, "Specifying Minimum Build System Requirements for Python Projects." The "Rationale" section of the latter document concisely describes why these changes were required, the most significant being removal of the need to run the *setup.py* file to discover (presumably by observing tracebacks) the build's requirements.

The major purpose of PEP 517 is to specify the format of build definitions in a file called *pyproject.toml*. The file is organized into sections called *ta-*

bles, each with a header comprising the table's name in brackets, much like a config file. Each table contains values for various parameters, consisting of a name, an equals sign, and a value. **3.11+** Python includes the `tomllib` module for extracting these definitions, with `load` and `loads` methods similar to those in the `json` module.[1]

Although more and more tools in the Python ecosystem are using these modern standards, you should still expect to continue to encounter the more traditional `setuptools`-based build system (which is itself **transitioning** to the *pyproject.toml* base recommended in PEP 517). For an excellent survey of packaging tools available, see the **list** maintained by the Python Packaging Authority (PyPA).

To explain packaging, we first describe its development, then we discuss `poetry` and `setuptools`. Other PEP 517-compliant build tools worth mentioning include `flit` and `hatch`, and you should expect their number to grow as interoperability continues to improve. For distributing relatively simple pure Python packages, we also introduce the standard library module `zipapp`, and we complete the chapter with a short section explaining how to access data bundled as part of a package.

What We Don't Cover in This Chapter

Apart from the PyPA-sanctioned methods, there are many other possible ways of distributing Python code—far too many to cover in a single chapter. We do not cover the following packaging and distribution topics, which may well be of interest to those wishing to distribute Python code:

- Using **conda**
- Using **Docker**
- Various methods of creating binary executable files from Python code, such as the following (these tools can be tricky to set up for complex projects, but they repay the effort by widening the potential audience for an application):
  - — **PyInstaller**, which takes a Python application and bundles all the required dependencies (including the Python interpreter and

necessary extension libraries) into a single executable program that can be distributed as a standalone application. Versions exist for Windows, macOS, and Linux, though each architecture can only produce its own executable.

- — **PyOxidizer**, the main tool in a utility set of the same name, which not only allows the creation of standalone executables but can also create Windows and macOS installers and other artifacts.
- — **cx_Freeze**, which creates a folder containing a Python interpreter, extension libraries, and a ZIP file of the Python code. You can convert this into either a Windows installer or a macOS disk image.

A Brief History of Python Packaging

Before the advent of virtual environments, maintaining multiple Python projects and avoiding conflicts between their different dependency requirements was a complex business involving careful management of `sys.path` and the `PYTHONPATH` environment variable. If different projects required the same dependency in two different versions, no single Python environment could support both. Nowadays, each virtual environment (see **"Python Environments"** for a refresher on this topic) has its own *site_packages* directory into which third-party and local packages and modules can be installed in a number of convenient ways, making it largely unnecessary to think about the mechanism.[2]

When the Python Package Index was conceived in 2003, no such features were available, and there was no uniform way to package and distribute Python code. Developers had to carefully adapt their environment to each different project they worked on. This changed with the development of the `distutils` standard library package, soon leveraged by the third-party `setuptools` package and its `easy_install` utility. The now-obsolete platform-independent *egg* packaging format was the first definition of a single-file format for Python package distribution, allowing easy download and installation of eggs from network sources. Installing a package used a *setup.py* component, whose execution would integrate the package's code into an existing Python environment using the features of `setuptools`. Requiring a third-party (i.e., not part of the standard distribution) module such as `setuptools` was clearly not a fully satisfactory solution.

In parallel with these developments came the creation of the `virtualenv` package, vastly simplifying project management for the average Python programmer by offering clean separation between the Python environments used by different projects. Shortly after this, the `pip` utility, again largely based on the ideas behind `setuptools`, was introduced. Using source trees rather than eggs as its distribution format, `pip` could not only install packages but uninstall them as well. It could also list the contents of a virtual environment and accept a versioned list of the project's dependencies, by convention in a file named *requirements.txt*.

`setuptools` development was somewhat idiosyncratic and not responsive to community needs, so a fork named `distribute` was created as a drop-in replacement (it installed under the `setuptools` name), to allow development work to proceed along more collaborative lines. This was eventually merged back into the `setuptools` codebase, which is nowadays controlled by the PyPA: the ability to do this affirmed the value of Python's open source licensing policy.

-3.11 The `distutils` package was originally designed as a standard library component to help with installing extension modules (particularly those written in compiled languages, covered in **Chapter 25**). Although it currently remains in the standard library, it has been deprecated and is scheduled for removal from version 3.12, when it will likely be incorporated into `setuptools`. A number of other tools have emerged that conform to PEPs 517 and 518. In this chapter we'll look at different ways to install additional functionality into a Python environment.

With the acceptance of **PEP 425**, "Compatibility Tags for Built Distributions," and **PEP 427**, "The Wheel Binary Package Format," Python finally had a specification for a binary distribution format (the *wheel*, whose definition has **since been updated**) that would allow the distribution of compiled extension packages for different architectures, falling back to installing from source when no appropriate binary wheel is available.

PEP 453, "Explicit Bootstrapping of pip in Python Installations," determined that the `pip` utility should become the preferred way to install packages in Python, and established a process whereby it could be updated independently of Python to allow new deployment features to be delivered without waiting for new language releases.

These developments and many others that have rationalized the Python ecosystem are due to a lot of hard work by the PyPA, to whom Python's ruling "Steering Council" has delegated most matters relating to packaging and distribution. For a more in-depth and advanced explanation of the material in this chapter, see the **"Python Packaging User Guide"**, which offers sound advice and useful instruction to anyone who wants to make their Python software widely available.

Online Material

As mentioned at the start of the chapter, the **online version of this chapter** contains additional material. The topics covered are:

- The build process
- Entry points
- Distribution formats
- `poetry`
- `setuptools`
- Distributing your package
- `zipapp`
- Accessing data included with your code

1 Users of older versions can install the library from PyPI with **`pip install toml`**.

2 Be aware that some packages are less than friendly to virtual environments. Happily, these are few and far between.