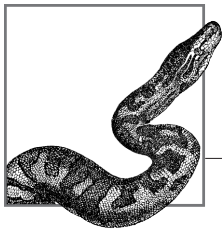

24

Packaging Programs and Extensions

This chapter assumes that 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 *tables*, 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

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

`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`, a modern standards-compliant Python build system, comparing it with the more traditional `setuptools` approach that many projects continue to rely on to distribute their software. Other PEP 517-compliant build tools worth mentioning—but not covered further—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.

For a more in-depth and advanced explanation of the material in this chapter, see the "Python Packaging User Guide", maintained by the PyPA, which offers sound advice and useful instruction to anyone who wants to make their Python software widely available.

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 two different versions of the same dependency, no single Python environment could support both. Nowadays, each virtual environment (see "Python Environments" in Chapter 7 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] Having multiple versions of Python installed on your machine is a different issue, which does not necessarily require virtual environments (although it does no harm to use them!). Rather, virtual environments are a must to maintain multiple projects using the same Python version but with different, possibly conflicting, dependencies.

When the Python Package Index (PyPI) was conceived in 2003, virtual environments were not 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`. However, requiring a third-party 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. Affirming the value of Python's open source licensing

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

policy, this was eventually merged back into the `setuptools` codebase, which is controlled today by the PyPA.

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 since the appearance of `distutils` that conform to PEPs 517 and 518, and in this chapter we'll look at some different ways to install additional functionality into a Python environment.

With the acceptance in 2012 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.

In 2013, 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, such as those mentioned in this chapter's introduction, that have rationalized the Python ecosystem, are the result of a lot of hard work by the PyPA, to whom Python's ruling Steering Council has delegated most matters relating to packaging and distribution.

The Build Process

Ultimately, all Python applications, libraries, and extensions start out as source code, organized according to the project's needs into a source tree. Building the project requires transforming the source tree into one or more distributions in wheel format. The so-called *build frontend* performs this task by calling *hooks* in a *build backend* specified as part of the package's metadata. The build backend does not necessarily need to implement all of the defined hooks: PEP 517 specifies which ones are mandatory. You can think of the build frontend as the coordinator, and the build backends as the component-specific build processes required to produce the necessary artifacts for distribution. Many tools also offer features to assist with distributing the wheels, once they're built.

Open source software often achieves the transfer by uploading distributions to PyPI, where they are available for download and installation by the general public. It's to the credit of the PyPA that, after such significant refactoring of Python's distribution system, the humble **`pip install`** *package* command still works so effectively.[3]

Let's take a look at the source tree of a typical pure Python package before it's made ready for distribution. The package's source code is in the *src/ch24* subdirectory, and the (`pytest`-based) tests are in the *tests* subdirectory, as shown in Example 24-1.

Example 24-1. Layout of a simple pure Python project

```
.
├── README.md
├── src
│   └── ch24
│       ├── __init__.py
│       ├── _version.py
│       ├── dotted_dict.py
│       └── object_dict.py
└── tests
    ├── __init__.py
    ├── test_dotted_dict.py
    ├── test_ch24.py
    ├── test_json_hooking.py
    └── test_object_dict.py
```

We'll examine how this can be packaged using both `poetry` and `setuptools` after we've described the structures used to transmit Python packages from one system to another.

Entry Points

Entry points are a way to identify particular pieces of your code (modules or functions, in most cases) by names that need not relate to the names they are given in the logic. They have two primary purposes. First, if a package includes command-line functionality, an entry point identifies each piece of functionality that should be run as the result of running that entry point (`setuptools` calls these *console scripts*, `poetry` simply calls them *scripts*). Second, you can use entry points to make your software extensible with plug-ins (see "Creating and Discovering Plugins" in the `setuptools` documentation).

3 The Python Software Foundation runs significant infrastructure to support the PyPA, its membership, and the Python development team. Donations to the PSF are always welcome.

Distribution Formats

As mentioned, Python projects are organized into a directory structure usually called the source tree. The very minimum you need to do to be able to distribute a project with standard Python tools is to build a *source distribution*. This contains some of the files from the source tree,[4] along with metadata that describes both the package and any additional artifacts required to install it. *Built distributions* are a little more complex, being specific to a particular architecture and version of Python.

Python recognizes two types of packages. *Pure packages* contain only Python code and associated data. *Extension packages*, on the other hand, include code in compiled languages that will require the attention of some non-Python tools to prepare it for installation in what is often referred to as a *build process*, specific to the architecture and environment of the target system and orchestrated by calls to the hooks in the package's build backend. Packages can be distributed as either source distributions or built distributions, both of which theoretically contain the artifacts required to install on a specific type of target.

Source distributions are compressed TAR archives (with file extension *.tar.gz*) that contain all the code of the package. For a pure package to become operational in the target environment, all you need to do is move the source distribution to the correct location (usually a virtual environment's *site-packages* directory). For an extension package, however, a source distribution requires the target system to already have installed all the necessary dependencies (language compilers and the like) in order to perform the build process.

The majority of computer users don't maintain a development environment, so built distributions are usually a handier solution for them. They are delivered as *wheel* files,[5] which are compressed zip files containing everything that needs to be added to a Python environment (of a specific version on a machine of a particular architecture) to render the package functional. Installing a built distribution of an extension package on a specific target is almost as simple as installing a pure Python package, since the build work is performed prior to distribution.

A built distribution's name contains *tags*, specified in PEP 425, to indicate (among other things):

- The Python implementation and version(s), e.g., *py27* for Python 2.7, *py3* for any Python 3
- A particular ABI,[6] e.g., *cp38* for CPython 3.8

4 For example, it isn't customary to package up tests as part of a distribution.

5 Pure packages can also be distributed as wheels.

6 The ABI, or application binary interface, is the machine-level interface Python offers to extensions.

- A specific platform, e.g., *win32* or *linux_i386*

PEP 427's naming convention suggests using the following format for wheel filenames:

```
{dist}-{version}(-{build_tag})?-{python_tag}-{abi_tag}-{platform_tag}.whl
```

A typical wheel for a built package might therefore be given a name such as *psycopg2-2.9.3-py310-cp310-macosx_12_0_arm64.whl*. This specifies version 2.9.3 of the `psycopg2` package, for CPython version 3.10 on the macOS platform with an ARM 64-bit processor. As you can see, built distribution wheels are quite specific!

Fortunately, much of this complexity is hidden from the average Python programmer, who simply uses `pip` or a similar tool to maintain multiple virtual environments. The tools select appropriate built distributions if they are available, and otherwise attempt to perform the build locally from a source distribution if possible. Unless all the required supporting software is installed on the target system, this attempt will fail.

poetry

The `poetry` utility is designed to offer all the facilities required to build and publish Python packages, having both frontend and backend build capabilities. Its primary interface is the command line, so operations can easily be scripted, and it is configured by the `[tool.poetry]` table in *pyproject.toml* (each section of the file is known as a "table"). `poetry` implements subcommands for the various tasks it can perform, the first one you generally need being `init`. Issuing the **`poetry`** command on its own gives you a list of the available subcommands.

This section is not intended to provide an exhaustive feature list for `poetry`, whose documentation and command-line help are both concise and informative. Instead, it walks you through creating a simple project to explain the major options of `poetry`'s most often used subcommands.

Preparing a Project for poetry

Once you've installed `poetry`, the command **`poetry new`** *`project_name`* creates a skeletal project structure in the directory *project_name*. Change into the project's root directory (ours is called *ch24*), then issue the **`poetry init`** command. This will guide you through creating a *pyproject.toml* file, along the lines shown in Example 24-2. Note that you can specify a range of compatible Python versions, as shown here, in addition to the one that's suggested (the version running `poetry`).

Example 24-2. A dialog with the `poetry init` command (user entries in bold)

```
$ cd ch24
$ poetry init
This command will guide you through creating your pyproject.toml config.
```

```
Package name [ch24]: ↩
Version [0.1.0]: ↩
Description []:  Copy of hu code for demonstration purposes
Author [Steve Holden <steve@example.com>, n to skip]: ↩
License []:  MIT
Compatible Python versions [^3.10]:  ^3.8 || ^3.9 || ^3.10

Would you like to define your main dependencies
    interactively? (yes/no) [yes] no
Would you like to define your development
    dependencies interactively? (yes/no) [yes] no
Generated file

[tool.poetry]
name = "ch24"
version = "0.1.0"
description = "Copy of hu code for demonstration purposes"
authors = ["Steve Holden <steve@example.com>"]
license = "MIT"

[tool.poetry.dependencies]
python = "^3.8 || ^3.9 || ^3.10"

[tool.poetry.dev-dependencies]

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"

Do you confirm generation? (yes/no) [yes] yes
```

As you can see from the listing, `poetry` generated a *pyproject.toml* file with two tables:

- The `[tool.poetry]` table establishes the basic project metadata. Its `dependencies` subtable contains only the supported Python versions, and its `dev-dependencies` subtable is initially empty, though you can enter dependencies one by one interactively in the dialog should you so choose.
- The `[build-system]` table confirms `poetry` as the build backend and affirms the requirement to install it for build purposes.

Note that `poetry` separates the build and development dependencies (only needed by developers) from the dependencies of the software being packaged.

In this project the actual source code lives in the *src/ch24/* subdirectory. Telling the build chain where the source representing the package can be found requires adding a further key to the `[tool.poetry]` table called `packages`, as shown here:

```
packages = [
    { include = "ch24", from = "src" },
]
```

The `packages` key references a list, so you can include multiple packages, although this simple example includes only one.

Establishing and Managing Virtual Environments

`poetry` allows you to switch between virtual environments, which it can manage for you with its **`env`** subcommand. Executing the following command, for example, causes `poetry` to create and populate a Python 3.10 environment which then becomes the default for `poetry` commands:

```
$ poetry env use 3.10
Creating virtualenv ch24-0iMRbWbT-py3.10 in /Users/sholden/Library/
Caches/pypoetry/virtualenvs
Using virtualenv: /Users/sholden/Library/Caches/pypoetry/virtualenvs/
ch24-0iMRbWbT-py3.10
```

Should you need to add dependencies for your project, use the **`poetry add`** *`package_name`* command. For dependencies only required during development, add the **`-D`** option:

```
$ poetry add -D pytest
Using version ^7.1.2 for pytest

Updating dependencies
Resolving dependencies... (1.0s)
Writing lock file

Package operations: 8 installs, 0 updates, 0 removals

  • Installing pyparsing (3.0.9)
  • Installing attrs (22.1.0)
  • Installing iniconfig (1.1.1)
  • Installing packaging (21.3)
  • Installing pluggy (1.0.0)
  • Installing py (1.11.0)
  • Installing tomli (2.0.1)
  • Installing pytest (7.1.2)
```

These changes immediately affect the current environment. Remove packages with **`poetry remove`** *`package_name`*. To test consistency with Python 3.9, you could use **`poetry use 3.9`** to create another environment; **`poetry install`** will then add known dependencies to the new environment. **`poetry env list`** will show you the existing environments, identifying the currently active one:

```
(v39) $ poetry env list
ch24-0iMRbWbT-py3.10
ch24-0iMRbWbT-py3.9 (Activated)
```

poetry May Interact Poorly with an Active Virtual Environment

When you issue `poetry` commands with a virtual environment already active, especially in complex testing frameworks like `tox`, there can be confusion about exactly which version of which environment is active for commands run from the command line. Make sure to deactivate any active virtualenv before using `poetry` commands.

A very useful command for developers is **`poetry shell`**, which starts a subshell with the current `poetry` virtualenv already activated. Note the change of Python interpreter after issuing the command:

```
$ which python
/usr/bin/python
$ poetry shell
Spawning shell within /Users/sholden/Library/Caches/pypoetry/
virtualenvs/ch24-0iMRbWbT-py3.9
$ . /Users/sholden/Library/Caches/pypoetry/virtualenvs/ch24-
0iMRbWbT-py3.9/bin/activate
(ch24-0iMRbWbT-py3.9) $ which python
/Users/sholden/Library/Caches/pypoetry/virtualenvs/ch24-0iMRbWbT-py3.9/
bin/python
```

Terminating the shell with **`exit`** or **`^D`** (**`^Z`** on Windows) gets you back to your original environment. To run a single command with the virtualenv active, use **`poetry run`** *`command args`*.

Handling Entry Points in poetry

Identify entry points to `poetry` in the `[tool.poetry.scripts]` table in *pyproject.toml*. Each key in the table will cause an executable of that name to be installed in the */bin* directory of the virtual environment it's installed into. Each value is a string, which should be the name of a module, followed by a colon and the name of a callable object within that module. You can also add a `[tool.poetry.plugins]` table whose subtables associate keys with objects referenced in the same way as the scripts.

Building Your Project

With at least one virtual environment available, you can now ask `poetry` to perform the build backend function with the **`poetry build`** command. `poetry` will install any necessary dependencies (though our simple package has none) before running the build:

```
$ poetry build
Building ch24 (0.1.0)
  - Building sdist
  - Built ch24-0.1.0.tar.gz
  - Building wheel
```

```
  - Built ch24-0.1.0-py3-none-any.whl
$ tree dist
dist
├── ch24-0.1.0-py3-none-any.whl
└── ch24-0.1.0.tar.gz
```

By default, `poetry` builds both a source distribution and a wheel in the project's *dist* subdirectory. Since this is a pure Python package, the wheel contains only metadata and the necessary source files. Note the suffix to the wheel file, *-py3-none-any.whl*, which indicates that it's a Python 3 package with no API requirements and should run on any architecture. This is typical for pure packages. Listing the TAR shows the contents:

```
$ tar -tzf dist/ch24-0.1.0.tar.gz
ch24-0.1.0/pyproject.toml
ch24-0.1.0/src/ch24/__init__.py
ch24-0.1.0/src/ch24/_version.py
ch24-0.1.0/src/ch24/dotted_dict.py
ch24-0.1.0/src/ch24/object_dict.py
ch24-0.1.0/setup.py
ch24-0.1.0/PKG-INFO
```

Notice that the source distribution contains a *setup.py* file; its contents are discussed further in "setuptools" on page 13. The `unzip` utility lists the wheel file's contents:

```
$ unzip -l dist/ch24-0.1.0-py3-none-any.whl
Archive:  dist/ch24-0.1.0-py3-none-any.whl
  Length      Date    Time    Name
---------  ---------- -----   ----
      122  01-01-1980 00:00   ch24/__init__.py
      145  01-01-1980 00:00   ch24/_version.py
     3477  01-01-1980 00:00   ch24/dotted_dict.py
     2403  01-01-1980 00:00   ch24/object_dict.py
       83  01-01-2016 00:00   ch24-0.1.0.dist-info/WHEEL
      496  01-01-2016 00:00   ch24-0.1.0.dist-info/METADATA
      484  01-01-2016 00:00   ch24-0.1.0.dist-info/RECORD
---------                     -------
     7210                     7 files
```

Installing Locally

Once `poetry` has built your distributions, it's easy to install them in local environments, either from the source distribution or the wheel. Note that installing from source causes `pip` to create another wheel, which it then installs!

```
$ poetry run pip install ch24-0.1.0.tar.gz
Processing ./ch24-0.1.0.tar.gz
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: ch24
  Building wheel for ch24 (pyproject.toml) ... done
```

```
  Created wheel for ch24: filename=ch24-0.1.0-py3-none-any.whl
size=3744 sha256=5ce2c779d68e5c430391cde42ed591a27fc225b29cf70afdf6a29f
2325fa9fbe
  Stored in directory:
/Users/sholden/Library/Caches/pip/wheels/80/ef/fe/465653c2686aa24beae75
39bd8946215d95356c1e6256ee5de
Successfully built ch24
Installing collected packages: ch24
Successfully installed ch24-0.1.0
```

Installing from the wheel is simpler, however, precisely because the work to create it has already been done:

```
(ch24env) $ pip install dist/ch24-0.1.0-py3-none-any.whl
Processing ./dist/ch24-0.1.0-py3-none-any.whl
Installing collected packages: ch24
Successfully installed ch24-0.1.0
```

Keep a Package's Name the Same as Its Import Name

Unfortunately, there is no requirement for the project name in *pyproject.toml* to be the same as the name of the package directory. The `BeautifulSoup` package (covered in Chapter 22), for example, is the cause of some consternation among developers who, having run **`pip install BeautifulSoup`**, are then perplexed to find that their Python code has to **`import`** `bs4`. We suggest you make things as simple as possible by keeping both names the same.

Distributing Your Project with poetry

Because `poetry` aims to meet as many of the needs of developers as possible, it also allows you to publish your packages to the PyPI repository. It's beyond the scope of this chapter to describe setting up the authentication to provide credentials automatically, but when provided with the credentials for your account, the command **`poetry publish --username`** *`user`* **`--password`** *`pw`* will push a new version of `ch24` to PyPI. Alternatively, you can use the `twine` package to upload it, as described in "Registering and Uploading to a Repository" on page 28, later in this chapter.

Once your package is up on PyPI, any Python user on the internet can download and install it.

Other Commands

Some other useful `poetry` commands are listed in Table 24-1.

Table 24-1. Useful `poetry` commands

`cache`	Manages caching of downloaded resources
`config`	Lets you create and edit `poetry` configuration item values
`export`	Exports the lock file (specifying dependencies and their versions) in a variety of formats
`lock`	Creates a lock file from the currently installed dependencies
`search`	Searches for packages on remote repositories
`version`	Manages the version of the distribution

setuptools

`setuptools` is a rich and flexible set of tools for packaging Python programs and extensions for distribution; while it relies on the standard library's `distutils`, it offers more flexibility and gives insurance against the forthcoming retirement of that package. It has been the primary mechanism for packaging and distribution for the past decade, and is currently migrating toward the standards of PEP 517 and PEP 518. A full description of driving `setuptools` using *pyproject.toml* can be found in the documentation, and this is the preferred way to distribute new projects (or update older ones). `setuptools` is automatically installed as a part of any virtual environment created with the **`python -m venv`** *`path`* command.

This Section Describes Legacy Methods

While the process described in the remainder of this section may be useful in maintaining legacy projects, we don't recommend that you use this approach for new projects. You will find a more modern take on using `setuptools` together with *pyproject.toml* in the user guide (see "Building and Distributing Packages with Setuptools").

The Source Tree and Its Root

A source tree is the set of files to package into a single archive file for distribution purposes, and may include metadata as well as program sources. It can contain Python packages and/or other Python modules (as covered in Chapter 7), as well as, optionally, Python scripts, C-coded (and other) extensions, data files, and auxiliary files with metadata.

You usually place all the files of the source tree in a directory known as the *source root*, and in subdirectories of the source root. Mostly, you can arrange the subtree of files and directories to suit your needs. However, as covered in "Packages" in Chapter 7, a Python package must reside in its own directory (unless you are creating *namespace packages*, also discussed in Chapter 7), and each package's directory must

contain a file named *__init__.py* (and subdirectories with *__init__.py* files for the package's subpackages, if any) as well as other modules that belong to that package.

`setuptools` expects each distribution to include a *setup.py* script, and optionally a *README* file (preferably in reStructuredText format); it may also contain a *requirements.txt*, a *MANIFEST.in*, and a *setup.cfg*. These files are all covered in the following sections.

Testing Your Package During Development

If you wish to test your package while developing it, you can install it locally with **`pip install -e`** *`project_source`*. The **`-e`** flag stands for "editable" (aka development mode), because editing the source will affect the virtual environment in which the package is installed. The *`project_source`* is frequently a local project directory, or a local or remote source repository reference. Complete details are provided in the online docs.

The setup.py Script

The distribution root directory must contain a Python script, by convention named *setup.py*. The *setup.py* script can, in theory, contain arbitrary Python code. However, in practice it usually boils down to some variation of this:

```
from setuptools import setup, find_packages
# Optional configuration steps
setup( <many named arguments can go here> )
# Optional post-install steps
```

You should also import `Extension` if your *setup.py* deals with a non-pure distribution.

The call to `setup` analyzes the command line used to run the *setup.py* script, which will be of the form:

```
python setup.py [options] command [options]
```

The available commands can be found using the **`--help-commands`** command-line option:

```
$ python /tmp/setup.py --help-commands
Standard commands:
  build             build everything needed to install
  build_py          "build" pure Python modules (copy to build
                    directory)
  build_ext         build C/C++ and Cython extensions (compile/link to
                    build directory)
  build_clib        build C/C++ libraries used by Python extensions
  build_scripts     "build" scripts (copy and fixup #! line)
  clean             clean up temporary files from 'build' command
  install           install everything from build directory
  install_lib       install all Python modules (extensions and pure
```

```
                    Python)
  install_headers   install C/C++ header files
  install_scripts   install scripts (Python or otherwise)
  install_data      install data files
  sdist             create a source distribution (tarball, zip file,
                    etc.)
  register          register the distribution with the Python package
                    index
  bdist             create a built (binary) distribution
  bdist_dumb        create a "dumb" built distribution
  bdist_rpm         create an RPM distribution
  bdist_wininst     create an executable installer for MS Windows
  check             perform some checks on the package
  upload            upload binary package to PyPI

Extra commands:
  alias             define a shortcut to invoke one or more commands
  bdist_egg         create an "egg" distribution
  develop           install package in 'development mode'
  dist_info         create a .dist-info directory
  easy_install      Find/get/install Python packages
  egg_info          create a distribution's .egg-info directory
  install_egg_info  Install an .egg-info directory for the package
  rotate            delete older distributions, keeping N newest files
  saveopts          save supplied options to setup.cfg or other config
                    file
  setopt            set an option in setup.cfg or another config file
  test              run unit tests after in-place build (deprecated)
  upload_docs       Upload documentation to sites other than PyPi such
                    as devpi

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd --help
```

Mostly we'll be considering the installation case (**`python setup.py install ...`**).

Configuring the setup

You can provide configuration parameters to `setup` in three different ways:

- The simplest option is to provide named arguments to your *setup.py*'s call to `setuptools.setup`. The examples we use here show the configuration parameters as they would appear when provided as setup arguments.
- You can offer your users better installation flexibility by instead providing a *setup.cfg* file that provides the same information—this separation of function from configuration data makes it simpler for users to edit the necessary data

for custom installations.[7] More modern tools will also look in *pyproject.toml*, to which configuration data should, in the long term, migrate.

- Finally, you can add configuration options to the command line you use to execute the `setup` function.

It is perfectly acceptable to include other logic in your *setup.py* before the call to `setup`. For example, a `long_description` string may be populated from its own separate file, as in:

```
with open('./README.rst') as f:
    long_description = f.read()
```

Configuration parameters fall primarily into three groups: metadata about the distribution, information about which files are in the distribution, and information about dependencies. An example of modern practice is the *setup.py* for the Django web framework, shown in Example 24-3.

Example 24-3. Django framework setup.py file

```
import os
import site
import sys
from distutils.sysconfig import get_python_lib

from setuptools import setup

# Allow editable install into user site directory.
# See https://github.com/pypa/pip/issues/7953.
site.ENABLE_USER_SITE = "--user" in sys.argv[1:]

# Warn if we are installing over top of an existing installation. This can
# cause issues where files that were deleted from a more recent Django are
# still present in site-packages. See #18115.
overlay_warning = False
if "install" in sys.argv:
    lib_paths = [get_python_lib()]
    if lib_paths[0].startswith("/usr/lib/"):
  # We have to try also with an explicit prefix
  # of /usr/local in order to
  # catch Debian's custom user site-packages directory.
        lib_paths.append(get_python_lib(prefix="/usr/local"))
    for lib_path in lib_paths:
        existing_path = os.path.abspath(os.path.join(lib_path, "django"))
        if os.path.exists(existing_path):
# We note the need for the warning here,
```

7 Options for specific setup commands can also be included. The **`--test-suite=NAME`** command-line option to the **`test`** command would appear as `test_suite_name = NAME` in *setup.py*'s `[test]` section.

```
# but present it after the
# command is run, so it's more likely to be seen.
            overlay_warning = True
            break

setup()

if overlay_warning:
    sys.stderr.write(
        """
========
WARNING!
========
You have just installed Django over top of an existing
installation, without removing it first. Because of this,
your install may now include extraneous files from a
previous version that have since been removed from
Django. This is known to cause a variety of problems. You
should manually remove the
%(existing_path)s
directory and re-install Django.
"""
        % {"existing_path": existing_path}
    )
```

Most of the logic is concerned with detecting the situation where someone installs Django on top of an existing installation. The actual functionality is provided by a simple call to `setup`, with no arguments provided. The heavy lifting of configuration is performed by the *setup.cfg* file, shown in Example 24-4 (though those settings could still be overridden by command-line arguments).

Example 24-4. Django framework setup.cfg file

```
[metadata]
name = Django
version = attr: django.__version__
url = https://www.djangoproject.com/
author = Django Software Foundation
author_email = foundation@djangoproject.com
description = A high-level Python web framework that encourages rapid
development and clean, pragmatic design.
long_description = file: README.rst
license = BSD-3-Clause
classifiers =
    Development Status :: 2 - Pre-Alpha
    Environment :: Web Environment
    Framework :: Django
    Intended Audience :: Developers
    License :: OSI Approved :: BSD License
    Operating System :: OS Independent
    Programming Language :: Python
```

```
    Programming Language :: Python :: 3
    Programming Language :: Python :: 3 :: Only
    Programming Language :: Python :: 3.8
    Programming Language :: Python :: 3.9
    Programming Language :: Python :: 3.10
    Topic :: Internet :: WWW/HTTP
    Topic :: Internet :: WWW/HTTP :: Dynamic Content
    Topic :: Internet :: WWW/HTTP :: WSGI
    Topic :: Software Development :: Libraries :: Application Frameworks
    Topic :: Software Development :: Libraries :: Python Modules
project_urls =
    Documentation = https://docs.djangoproject.com/
    Release notes = https://docs.djangoproject.com/en/stable/releases/
    Funding = https://www.djangoproject.com/fundraising/
    Source = https://github.com/django/django
    Tracker = https://code.djangoproject.com/
[options]
python_requires = >=3.8
packages = find:
include_package_data = true
zip_safe = false
install_requires =
    asgiref >= 3.5.2
    backports.zoneinfo; python_version<"3.9"
    sqlparse >= 0.2.2
    tzdata; sys_platform == 'win32'
[options.entry_points]
console_scripts =
    django-admin = django.core.management:execute_from_command_line
[options.extras_require]
argon2 = argon2-cffi >= 19.1.0
bcrypt = bcrypt
[bdist_rpm]
doc_files = docs extras AUTHORS INSTALL LICENSE README.rst
install_script = scripts/rpm-install.sh
[flake8]
exclude = build,.git,.tox,./tests/.env
extend-ignore = E203
max-line-length = 88
per-file-ignores =
    django/core/cache/backends/filebased.py:W601
    django/core/cache/backends/base.py:W601
    django/core/cache/backends/redis.py:W601
    tests/cache/tests.py:W601
[isort]
profile = black
default_section = THIRDPARTY
known_first_party = django
```

You will observe that the `django-admin` command is specified as a console entry point; it runs the `execute_from_command_line` function from `django.core.management`. The project also comes with a *pyproject.toml* file (shown in Example 24-5) that specifies `setuptools` as the build backend. As mentioned previously, in the long term you should move all configuration data to *pyproject.toml*.

Example 24-5. Django framework pyproject.toml file

```
[build-system]
requires = ['setuptools>=40.8.0']
build-backend = 'setuptools.build_meta'
[tool.black]
target-version = ['py38']
force-exclude = 'tests/test_runner_apps/tagged/tests_syntax_error.py'
```

For a fuller appreciation of the significant work that goes into releasing a new version of a large Python project, you may find the Django project's release documentation of interest.

Metadata about the distribution

It's important to provide metadata about the distribution by supplying some of the configuration parameters shown in Table 24-2. The value you associate with each argument name you supply is a string, intended mostly to be human-readable; the specifications about the string's format are advisory. The explanations and recommendations about the metadata fields in the following table are also nonnormative and correspond only to common, not universal, conventions. References to "this distribution" in these descriptions can be taken to refer to the material included in the distribution rather than to the packaging of the distribution. The following are the most commonly used metadata arguments; the PyPA's "Core Metadata Specification" gives full details.

Table 24-2. Common metadata arguments

`author`	The name(s) of the author(s) of material included in this distribution. Always provide this information: authors deserve credit for their work.
`author_email`	The email address(es) of the author(s) named in the argument `author`. You should usually provide this information.
`classifiers`	A list of Trove strings to classify your package; each string must be one of those listed in "Trove Classifiers" on PyPI.
`description`	A concise description of this distribution, preferably fitting within one line of 80 characters or less.
`keywords`	A list of strings that somebody looking for the functionality provided by this distribution would be likely to search for. Provide this information as a comma-separated list of keywords, so that users can find your package on PyPI or using other search engines.

`license`	The licensing terms of this distribution, in a concise form that typically points to the full license text, which is included as another distributed file or available at a certain URL.
`long_description`	A long description of this distribution, typically as provided in the *README* file (which is preferably in *.rst* format).
`maintainer`	The name(s) of the current maintainer(s) of this distribution. Provide this information when the maintainer is different from the author.
`maintainer_email`	The email address(es) of the maintainer(s) named in the argument `maintainer`. Provide this information only when you supply the `maintainer` argument and the maintainer is willing to receive email about this work.
`name`	The name of this distribution as a valid Python identifier (see PEP 426 for exact criteria). If you plan to upload your project to PyPI, this name must not conflict with any other project already in the PyPI database.
`platforms`	A list of platforms on which this distribution is known to work. Provide this information when you have reason to believe the distribution may not work everywhere. This information should be reasonably concise, so the field often references information at a URL or in another distributed file.
`url`	A URL at which more information can be found about this distribution, or **None** if no such URL exists.
`version`	The version of this distribution, normally structured as *`major.minor`* or even more finely. See PEP 440 for recommended versioning schemes.

Distribution contents

A distribution can contain a mix of Python source files, C-coded extensions, and data files. `setup` accepts optional named arguments that detail which files to put in the distribution. Whenever you specify file paths, the paths must be relative to the distribution root directory and use `/` as the path separator. `setuptools` adapts the location and separator appropriately when it installs the distribution. Wheels, in particular, do not support absolute paths: all paths are relative to the top-level directory of your package.

Distinguish Between Package and Filesystem Paths

The configuration parameters `packages` and `py_modules` do not list file paths, but rather Python packages and modules, respectively. Therefore, in the values of these named arguments, don't use path separators or file extensions. If you list subpackage names in `packages`, use Python module syntax instead (i.e., *`top_package.sub_package`*).

Python source files

By default, `setup` looks for Python modules (listed in the value of the configuration parameter `py_modules`) in the distribution root directory, and for Python packages

(listed in the value of the configuration parameter `packages`) as subdirectories of the distribution root directory.

Table 24-3 lists the `setup` configuration parameters you will use most frequently to detail which Python source files are part of the distribution.

Table 24-3. Common `setup` configuration parameters

`entry_points`	The value should be a `dict` holding one or more *groups*; each group consists of a list of *`name=value`* strings, where *`name`* is an identifier and *`value`* is a reference to a Python module or function. These references take the form of the (possibly dotted) name of a module or package, optionally followed by a colon and a (possibly dotted) reference to a function within it. The most common use of `entry_points` is to create executable scripts with the `console_scripts` and `gui_scripts` groups, but see also "Creating and Discovering Plugins" for further information on its use.
`packages`	The value should be a list of packages, which you can provide yourself or generate with a call to the `find_packages` function from `setuptools`, which can automatically locate and include packages and subpackages in your distribution root directory. For each package name string *p* in the list, `setup` expects to find a subdirectory *p* in the source root directory and includes in the distribution the file *p/__init__.py*, which must be present, as well as any other file *p/*.py* (i.e., all the modules of package *p*). `setup` does not search for subpackages of *p*: unless you use `find_packages`, you must explicitly list all subpackages, as well as top-level packages, in the value of the named argument `packages`. We recommend using `find_packages` to avoid having to update `packages` (and potentially miss a package) as your distribution grows.
`py_modules`	The value should be a list of module name strings. For each module name string *m* in the list, `setup` expects to find the file *m.py* in the distribution root directory and includes *m.py* in the distribution. Use `py_modules` instead of `find_packages` when you have a very simple package with only a few modules and no subdirectories.

Handling entry points in setup

The `entry_points` parameter tells the installer which plug-ins, services, or scripts to register for the application, and instructs it to create the appropriate platform-specific executables in the environment's */bin* directory. The primary `entry_points` *`group`* arguments used are `console_scripts` (which replaces the deprecated `scripts`) and `gui_scripts`. Other plug-ins and services (e.g., parsers) are also supported, but we do not cover them further in this book; see the "Python Packaging User Guide" for more detailed information.

When `pip` installs a package, it registers each entry point *`name`* and creates an appropriate executable (including an *.exe* launcher on Windows), which you can then run by simply entering **`name`** at the terminal prompt, rather than, for example, having to type **`python -m mymodule`**.

Scripts are generally Python source files that are meant to be run as main programs (see "The Main Program" in Chapter 7), usually from the command line. Each script file should have as its first line a shebang line—that is, a line starting with `#!`

and containing the substring `python`. In addition, each script should end with the following code block:

```
if __name_ == '__main__' :
    mainfunc()
```

To have `pip` install your script as an executable, list the script in `entry_points` under `console_scripts` (or `gui_scripts`, as appropriate). In addition to, or instead of, the main function of your script, you can use `entry_points` to register other functions as script interfaces. Here's what `entry_points` with both `console_scripts` and `gui_scripts` defined might look like:

```
entry_points={
    'console_scripts': ['example=example:mainfunc',
                        'otherfunc=example:anotherfunc',
                       ],
    'gui_scripts': ['mygui=mygui.gui_main:run',
                   ],
    },
```

After installation, just type **`example`** at the terminal prompt to execute *`mainfunc`* in the module *`example`*. If you type **`otherfunc`**, the system executes *`anotherfunc`*, also in the module *`example`*. Installing in the (usually virtual) environment means the environment's */bin* subdirectory will contain an item for each entry point, thereby guaranteeing the entry point(s) will be available whenever the environment is active.

Including packages and modules

Your code may be complex enough that it needs to be split into submodules and even subpackages. You can use the `packages` and `py_modules` arguments to `setup` for this purpose. `packages` can either be `find_packages()` or a list of package names, each corresponding to a subdirectory in the project root, and `py_modules` should be a list of modules, whose Python files should again appear in the package root directory:

```
packages=find_packages(),
modules=['mod_one', 'mod_two'],
```

For the latest advice on package and module discovery, see "Package Discovery and Namespace Packages" in the `setuptools` documentation.

Data and other files

To put files of any kind in the distribution, supply the named arguments listed in Table 24-4. In most cases, you'll want to use `package_data` to list your data files. The named argument `data_files` is used for listing files that you want to install to directories *outside* your package; however, we do not recommend you use it, due to complicated and inconsistent behavior, as described here.

Table 24-4. Named arguments for adding files to setup

`data_files`	`data_files=[(`*`target_directory, list_of_files`*`), ...]` The value of this argument is a list of pairs. Each pair's first item is a string that names a *target directory* (i.e., a directory where `setuptools` places data files when installing the distribution); the second item is the list of file path strings for files to put in the target directory. At installation time, installing from a wheel places each target directory as a subdirectory of Python's `sys.prefix` for a pure distribution, or of Python's `sys.exec_prefix` for a non-pure distribution. Installing from the source distribution with `pip` uses `setuptools` to place target directories relative to *site_packages*, but installing without `pip` and with `distutils` has the same behavior as with a wheel. Because of such inconsistencies, we do not recommend you use `data_files`.
`package_data`	`package_data={`*`k`*`: `*`list_of_globs, ...`*`}` The value of this argument is a `dict`. Each key is a string that names a *package* in which to find the data files; the corresponding value is a list of glob patterns for files to include. The patterns may include subdirectories (using relative paths separated by a forward slash, `/`, even on Windows). An empty package string, `''`, recursively includes all files in any subdirectory that matches the pattern—for example, `'': ['*.txt']` includes all *.txt* files anywhere in the top-level directory or subdirectories. At installation time, `setuptools` places each file in appropriate subdirectories relative to `site_packages`.

C-coded extensions

To put C-coded extensions in the distribution, supply the following named argument:

`ext_modules`	`ext_modules=[ <`*`list of instances of class Extension`*`> ]`
	All the details about each extension are supplied as arguments when instantiating the `setuptools.Extension` class. `Extension`'s constructor accepts two mandatory arguments and many optional named arguments. The simplest possible example looks something like this:
	`ext_modules=[Extension('x',sources=['x.c'])]`

The `Extension` class's constructor has the signature:

`Extension`	**`class`** `Extension(`*`name, sources, **kwds`*`)` *`name`* is the module name string for the C-coded extension. *`name`* may include dots to indicate that the extension module resides within a package. *`sources`* is the list of C source files that must be compiled and linked in order to build the extension. Each item of *`sources`* is a string that gives a source file's path relative to the distribution root directory, complete with the file extension *.c*. *`kwds`* lets you pass other optional named arguments to `Extension`, as covered later in this section.

The `Extension` class also supports additional file extensions besides *.c*, indicating other languages you may use to code Python extensions. On platforms having a

C++ compiler, the file extension *.cpp* indicates C++ source files. Other file extensions that may be supported, depending on the platform and on various add-ons to `setuptools`, include *.f* for Fortran, *.i* for SWIG, and *.pyx* for Cython files. See "Extending Python Without Python's C API" in Chapter 25 for information about using different languages to extend Python.

In most cases, your extension needs no further information besides the mandatory arguments *`name`* and *`sources`*. Note that you need to list any *.h* headers in your *MANIFEST.in* file. `setuptools` does all that is needed to make the Python headers directory and the Python library available for your extension's compilation and linking, and provides whatever compiler or linker flags or options are needed to build extensions on a given platform.

When additional information is required to compile and link your extension correctly, you can supply such information via the named arguments of the class `Extension`. However, when you plan to distribute your extensions to other platforms, you should examine whether you really need to provide build information via these arguments; it is sometimes possible to avoid this by careful coding at the C level. Using these arguments may potentially interfere with the cross-platform portability of your distribution. In particular, whenever you specify file or directory paths as the values of such arguments, the paths should be relative to the distribution root directory.

Table 24-5 lists the named arguments that you may pass when calling `Extension`.

Table 24-5. Named arguments of the `Extension` class

`define_macros`	`define_macros=[(`*`macro_name`*`,`*`macro_value`*`) ...]` Each of the items *`macro_name`* and *`macro_value`* is a string, respectively the name and value of a C preprocessor macro definition, equivalent in effect to the C preprocessor directive `#define` *`macro_name macro_value`*. *`macro_value`* can also be **None**, to get the same effect as the C preprocessor directive `#define` *`macro_name`*.
`extra_compile_args`	`extra_compile_arg=[<`*`list of compile_arg strings`*`>]` Each of the strings listed as the value of `extra_compile_args` is placed among the command-line arguments for each invocation of the C compiler.
`extra_link_args`	`extra_link_args=[<`*`list of link_arg strings`*`>]` Each of the strings listed as the value of `extra_link_args` is placed among the command-line arguments for the linker.
`extra_objects`	`extra_objects=[<`*`list of object_name strings`*`>]` Each of the strings listed as the value of `extra_objects` names an object file to link in. Do not specify the file extension as part of the object name: `distutils` adds the platform-appropriate file extension (*.o* on Unix-like platforms, *.obj* on Windows) to help you preserve cross-platform portability.
`include_dirs`	`include_dirs=[<`*`list of directory_path strings`*`>]` Each of the strings listed as the value of `include_dirs` identifies a directory to supply to the compiler as one where header files are found.

`libraries`	`libraries=[<list of library_name strings>]` Each of the strings listed as the value of `libraries` names a library to link in. Do not specify the file extension or any prefix as part of the library name: `distutils`, in cooperation with the linker, adds the platform-appropriate file extension and prefix (*.a* and the prefix *lib* on Unix-like platforms, *.lib* on Windows) to help you preserve cross-platform portability.
`library_dirs`	`library_dirs=[<list of directory_path strings>]` Each of the strings listed as the value of `library_dirs` identifies a directory to supply to the linker as one where library files are found.
`runtime_library_dirs`	`runtime_library_dirs=[<list of directory_path strings>]` Each of the strings listed as the value of `runtime_library_dirs` identifies a directory where dynamically loaded libraries are found at runtime.
`undef_macros`	`undef_macros=[<list of macro_name strings>]` Each of the strings *macro_name* listed as the value of `undef_macros` is the name of a C preprocessor macro definition, equivalent in effect to the C preprocessor directive `#undef` *macro_name*.

Dependencies and requirements

You may optionally list dependencies with named arguments in *setup.py*, as described in Table 24-6, or in a requirements file (described in the following section).

Table 24-6. Named arguments for listing dependencies in setup.py

`extras_require`	`extras_require={'extra_name':['pkgname'], ...}` Takes a `dict` of recommended additional dependencies: each key `'extra_name'` has as its value a list of packages that should be installed when the extra is selected at install time. `pip` does not automatically install these dependencies; the user opts into installing extra *rec* at installation time of a main package *mpk* with **`pip install`** *mpk*`[`*rec*`]`.
`install_requires`	`install_requires=['pkg',['pkg2>=n.n']]` Takes a list of package names as strings, with specific version requirements *n.n* optionally provided (see PEP 440 for details on how version specifiers are handled). Provide the broadest version requirements possible for your program. Use this named argument to supply dependencies that are the minimum necessary for your program to run. `pip` automatically calls **`pip install`** *pkg* on each argument *pkg* to install the dependency.

The requirements.txt File

You may optionally provide a *requirements.txt* file containing **`pip install`** specifications, one per line. This is useful to re-create a particular environment for installation, or to force the use of certain versions of dependencies.

When the user enters the following at a command prompt, `pip` installs all items listed in the file; however, installation is *not* guaranteed to be in any particular order:

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

install_requires Versus extras_require Versus requirements.txt?

`pip` only automatically discovers and installs those dependencies it finds listed in `install_requires`. `extras_require` entries must be specified at installation time (as described in the previous section), unless they are independently required by another package (e.g., when your package is one of a number being installed together). Packages in *requirements.txt* must be manually installed by the user. For further information and detailed usage instructions, see the `pip` documentation.

The MANIFEST.in File

When you package your source distribution, `setuptools` by default inserts the following files in the distribution:

- All Python (*.py*) and C source files explicitly listed in `packages` or found by `find_packages` in *setup.py*
- Files listed in `package_data` and `data_files` in *setup.py*
- Scripts or plug-ins defined in `entry_points` in *setup.py*
- Test files located at *test/test*.py* under the distribution root directory, unless excluded in `find_packages`
- The *README*, *README.md*, *README.rst*, or *README.txt* files (if any), as well as *setup.cfg* (if present) and *setup.py*

To add other files in the source distribution, place in its root directory a *manifest template* file named *MANIFEST.in*, whose lines are rules, applied sequentially, about files to add (`include`) or subtract (`prune`) from the list of files to place in the distribution. See the `distutils` documentation for more details. If you have any C extensions in your project (listed in *setup.py* named argument `ext_modules`), the path to any *.h* header files must also be listed in *MANIFEST.in* to ensure the headers are included, with a line like *`pkg_name`*`/include`, providing a relative path to the directory containing the headers.

Distributing Your Package

Once you have your *setup.py* (and other files) in order, to distribute your package:

1. Create the distribution as a wheel or other archive format.
2. Register your package, if necessary, to a repository.
3. Upload your package to a repository.

Create the Distribution

In the past, a packaged source distribution (*sdist*) made with **`python setup.py sdist`** was the most useful file you could produce with `distutils`. You'll still want to create an sdist when you are distributing packages with C extensions for flavors of Linux; and when you absolutely require absolute paths (rather than relative paths) for installation of certain files (listed in the `data_files` argument to *setup.py*), you need to use an sdist. (See the discussion on `data_files` in the "Python Packaging User Guide.") But when you are packaging pure Python, or platform-dependent C extensions for macOS or Windows, you can make life easier for most users by also creating built distributions as wheels.

Building wheels

For a non-pure distribution, making built forms available may be more than just an issue of convenience. An extension distribution, by definition, includes code that is not pure Python—generally, C code, though other languages can be integrated.[8] Unless you supply a built form, users need to have the appropriate C compiler installed in order to build and install your distribution. Installing a source distribution may be difficult for end users who are not experienced programmers. We therefore recommend you provide both an sdist, in *.tar.gz* format, and a wheel (or several, for non-pure packages). For that, you need to have the necessary C compiler installed. Non-pure wheels work only on other computers with the same platform (e.g., macOS, Windows) and architecture (e.g., 32-bit, 64-bit) they were built for.

Creating wheels

To create a wheel, in many cases, all you need to run is a single command. For a pure (Python-only) distribution, just type the following from your distribution's top-level directory:

```
$ python setup.py bdist_wheel
```

This creates a wheel that can be installed on any platform. Non-pure wheels can be created for macOS or for Windows, but only for the platform being used to

8 NumPy, for example, incorporates code written in FORTRAN.

create them. Running **`python setup.py bdist_wheel`** automatically detects the extension(s) and creates the appropriate wheel. One benefit of creating a wheel is that your users are able to **`pip install`** the package, regardless of whether they have a C compiler themselves, as long as they're running on the same platform as you used to create the wheel.

Non-Pure Linux Wheels

Unfortunately, distributing non-pure Linux wheels isn't anywhere near as simple as distributing pure ones, due to variations among Linux distributions, as described in PEP 600. PyPI does not accept non-pure Linux wheels unless they are tagged *manylinux*. The constraints of the systems used to create these cross-distribution wheels are described in the PEP; check out the `manylinux` *README* for the gory details.

Once you run **`python setup.py bdist_wheel`**, you will have a wheel named something like *mypkg-0.1-py2.py3-none-any.whl* in a (new, if you've run *setup.py* for the first time) directory called *dist/*. For more information on wheel naming and tagging conventions, see the now-venerable PEP 425.

Creating an sdist

To create a source distribution for your project, type the following in the top level of your package directory:

```
$ python setup.py sdist
```

This creates a *.tar.gz* file (on Windows, a *.zip*, by default; add **`--formats=gztar`** to make it produce the *.tar.gz* archive format instead). You may then upload the *.tar.gz* file to PyPI, or otherwise distribute it. Your users will have to download it, unpack it, and install it with, typically, **`python setup.py install`**. More information on source distributions is available in the online docs.

PyPI Prefers .tar.gz Files

Do not attempt to upload both a *.zip* and a *.tar.gz* file to PyPI; you'll get an error. Instead, stick with *.tar.gz* for most use cases.

Registering and Uploading to a Repository

Once you've created a wheel or an sdist, you may choose to upload it to a repository for easy distribution to your users. You can upload to a local repository, such as a company's private repository, or to a public repository such as PyPI. In the past, *setup.py* was used to build and immediately upload; however, due to issues with security, this is no longer recommended. Instead, you should use a third-party module such as `twine` (or `flit` for extremely simple packages, as covered in the

`flit` docs). There are plans to eventually merge `twine` into `pip`: check the "Python Packaging User Guide" for updated information.

Using `twine` is fairly straightforward. Run **`pip install twine`** to download it. You'll need to create a *~/.pypirc* file (which provides repository information), and then it's a simple command to register or upload your package.

The ~/.pypirc file

`twine` recommends that you have a *~/.pypirc* file (that is, a file named *.pypirc*, residing in your home directory) that lists information about the repositories you want to upload to, including your username and password. Since *.pypirc* is typically stored in cleartext, you should set the file's permissions to `600` (on Unix-like systems); use a tool such as keyring, set environment variables, or omit the password so you're prompted for it each time you run `twine`. The *~/.pypirc* file should look something like this:

```
[distutils]
index-servers=
    testpypi
    pypi
    myrepo
[testpypi]
repository=https://testpypi.python.org/pypi
username=your_username
password=your_password

[pypi]
repository=https://pypi.python.org/pypi
username=your_username
password=your_password

[myrepo]
repository=https://otherurl/myrepo
username=your_username
password=your_password
```

Registering and uploading to PyPI

If you've never used PyPI, first create a user account with your desired username and password, online at PyPI.) You should also create a TestPyPi user account to practice uploading packages to a temporary repository before uploading them to the public repository.

Registering your package before upload is not supported on PyPI. If you need to register on another repository, follow that repository's instructions.

Upload your package with the following command, issued from the root directory of your project (where the *dist/* subdirectory should be):

```
$ twine upload dist/*
```

`twine` finds the latest version of your distribution and uploads it to the repo specified in your *.pypirc* file (alternatively, you may provide the repository URL on the command line with the flag **`--repository-url`**). At this point, your users will be able to use **`python -m pip install`** *yourpkg* **`--user`** (or just **`python -m pip install`** in an active virtual environment—it's wiser to avoid a non-`venv` **`pip install`**, which affects the whole Python installation for all users and may require privileges the average user doesn't have) to install your package. Another acceptable variant is to mention the virtual environment explicitly as a command-line argument: **`python -m pip install`** *yourpkg* **`--user`** *venv*. (To list packages installed in the currently active virtual env or globally, use **`python -m pip list -l`**).

The PyPA continues to improve Python packaging and distribution. Please join in the effort by contributing to the codebase or documentation, posting bug reports, or joining in the discussions on packaging!

zipapp: Cheap and Cheerful Distribution

Python's ability to import modules from ZIP files has been deployed to great advantage in the `zipapp` module. This interesting and relatively little-known module lets you create a single file containing all the necessary pure Python code for an application that can then be directly run by the Python interpreter on the same or another computer.

This technique has several limitations, the most important of which is the inability to support compiled languages. For users who can live with those limitations, however, `zipapp` offers a very convenient mechanism for less formal distribution of Python programs.

`zipapp` can be driven from the command line using the **`python -m zipapp`** command. For a package `myapp` in a *src/myapp* directory, the command:

```
$ python -m zipapp src/myapp -m "myapp:main"
```

will create a *myapp.pyz* file containing the application's code. You can then run this file with:

```
$ python myapp.pyz
```

The module also offers an API that lets you create applications programmatically (the systems that run your *.pyz* file must already have a compatible Python interpreter installed for this to work). If your needs are simple enough to be satisfied by `zipapp`, you may well find that you don't need anything else.

Accessing Data Included with Your Code

Sometimes it's helpful to be able to distribute data that your code needs—anything from default configuration files to full-scale relational databases—along with your application or library. The techniques described in "The MANIFEST.in File" on page 26 and "Data and other files" on page 22 normally suffice to include your

data as a part of the delivered artifact, but other mechanisms may be available depending on your build backend. The remaining question is how to access such data at runtime.

In the past, many developers used `distutils.pkg_resources` to handle this task, or relied on computing the filestore path of the library and generating filestore paths relative to that. More recently, a far more elegant solution has emerged in the form of the `importlib.resources` subpackage. By using Python's import machinery, the library efficiently locates data installed as a part of the *installed* package's home directory, providing paths to imported resources that you can modify yourself. Not only that, but because the library's interface uses Python's import mechanism, you can read data artifacts distributed as part of packages in compressed ZIP files.

Some of the more significant functions in the module are listed in Table 24-7. Wherever a package is required, you can use either a module object (which should be an imported package) or a string containing the dotted name of an importable package, which will be imported. Note particularly that *resource names cannot contain any / path separators*.

Table 24-7. Functions of the `importlib.resources` module

`as_file`	**`as_file`**(*`traversable`*) **3.9+** *`traversable`* is a traversable object representing a file, as returned by a call to `importlib.resource.files`, for example. Returns a context manager (for use in a **`with`** statement) that provides a `pathlib.Path` object. Use when you need an actual file rather than an `importlib.abc.Traversable` interface.
`files`	`t = files(`*`package`*`)` **3.9+** `files` takes either a package or a string naming an importable package as an argument and returns an object that implements the `importlib.abc.Traversable` interface, which is a subset of that for `pathlib.Path` (see "The pathlib module" in Chapter 11).[a]

[a] For prior versions, the `files` method and other features are available as a backport by installing the `importlib-resources` package from PyPI, imported in your Python code as `import_resources`.

We'll assume for this example that when your package is installed, it appears somewhere in your filesystem with the directory tree shown here:

```
.../site-packages/ch24
├── __init__.py
├── _version.py
├── config.toml
├── data
│   └── myfile.dat
│   └── zipfile.zip
├── subpackage
│   └── __init__.py
│   └── data_file.dat
```

```
├── dotted_dict.py
├── object_dict.py
```

You can use path computations to work out the addresses of any file in your package and traverse its directory structure as necessary, allowing access to resources not inside subpackages:

```
>>> from importlib.resources import files, contents
>>> files('ch24')

PosixPath('.../site_packages/ch24')  # Shortened for publication

>>> with (files('ch24') / 'data' / 'myfile.dat').open() as f:
...     f.read()
...

'The quick brown fox jumps over the lazy dog.\n'
```

Note that the `contents` function does not recurse into subpackages or subdirectories:

```
>>> list(contents("ch24"))

['_version.py', '__init__.py', '__pycache__', 'object_dict.py',
'dotted_dict.py', 'subpackage', 'config.toml', 'data']

>>> list(contents("ch24.subpackage"))

['__init__.py', 'data_file.dat', '__pycache__']
```

Given a path object such as that returned by `files`, you can use its `joinpath` method, or the concatenation operator (`/`), to construct a path for objects underneath it in the directory tree.[9] This technique will fail, however, if the module is imported from a ZIP file because the objects therein are not filestore objects. You can access them as files, should you need to, using the `as_file` function from `importlib.resources`, which takes a path object and returns a context manager that also provides the `pathlib.Path` object.

When the context is activated, the function will extract the indicated object from the ZIP file and copy it temporarily into the filestore, removing the file when the context is destroyed. Suppose a ZIP file at */tmp/zipper.zip* has the contents shown in the following listing:

```
$ unzip -l zipper.zip
Archive:  zipper.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  10-20-2022 09:23   zipper/
       54  10-20-2022 09:23   zipper/module.py
        0  10-20-2022 09:24   zipper/package/
```

9 Beware here: `joinpath` implements no protections against directory traversal errors, which can unintentionally expose unanticipated parts of the filesystem.

```
        65  10-20-2022 09:24   zipper/package/__init__.py
         0  10-20-2022 12:16   zipper/package/data/
        53  10-20-2022 12:16   zipper/package/data/example.txt
---------                      -------
       172                      6 files
```

Here's how you might access the data file, even though it lives inside a ZIP file. You will notice the `Path` object has two components: the ZIP file path and the path of the desired artifact within the ZIP archive. While it's possible to open the file path for writing, there is little point in doing so, as those changes will not be reflected in the ZIP file:

```
>>> from importlib.resources import files, as_file
>>> import sys
>>> sys.path.insert(0, "/tmp/zipper.zip")
>>> import zipper.package
>>> path = files(zipper.package).joinpath("data/example.txt")
>>> path

Path('/tmp/zipper.zip', 'zipper/package/data/example.txt')

>>> with as_file(path) as fpath:
...     with open(fpath) as f:
...         print(f.name)
...         print(f.read())
...

/var/folders/29/j7_gxmh13zz6b21b8fr9bd_80000gn/T/tmp5hub946rexample.txt
This is the data in zipper/package/data/example.txt.
```

If you check the filestore after executing this example, you will find that the temporary file created by the call to `as_file` no longer exists.

The latest `importlib.resources` contains considerably more functionality than is used in the simple examples we offer here, and its documentation warrants further study. If you simply wish to gain access to files distributed with your package, these recipes will likely suffice.