# Python Packaging

John Strickler

Version 1.0, February 2025

# Table of Contents

# Chapter 1: Packaging

## Objectives

- Create a pyproject.toml file
- Understand the types of wheels
- Generate an installable wheel
- Configure dependencies
- Configure executable scripts
- Distribute and deploy packages

# Packaging overview

- Bundling project for distribution
- Uses build tools
- Needs metadata
- Extremely flexible

Packaging a project for distribution does not have to be complex. However, the tools are very flexible, and the amount of configuration can be overwhelming at first.

It boils down to these steps:

**Create a virtual environment**

While not absolutely necessary, creating a virtual environment for your project makes life easier, especially when it comes to dependency management.

**Create a project layout**

The arrangement of files and subfolders in the project folder. This can be *src* (recommended) or *flat*.

**Specify metadata**

Using `pyproject.toml`, specify metadata for your project. This metadata tells the build tools how and where to build and package your project.

**Build the project**

Use the build tools to create a *wheel* file. This wheel file can be distributed to developers.

**Install the wheel file**

Anyone who wants to install your project can use `pip` to install the project from the wheel file you built.

**Upload the project to PyPI (Optional)**

To share a project with everyone, upload it to the **PyPI** online repository.

# Terminology

Here are some terms used in Python packaging. They will be explained in more detail in the following pages.

**build backend**

A module that does the actual creation of installable files (wheels). E.g., `setuptools` (>=61), `poetry-core`, `hatchling`, `pdm-backend`, `flit-core`.

**build frontend**

A user interface for a build backend. E.g., `pip`, `build`, `poetry`, `hatch`, `pdm`, `flit`

**cookiecutter**

A tool to generate the files and folders needed for a project.

**dependency**

A package needed by the current package.

**editable install**

An installation that is really a link to the development folder, so changes to the code are reflected whenever the package or module is imported.

**package**

Can refer to either a **distribution package** or an **import package**.

**distribution package**

A collection of code (usually a folder) to be bundled into a reusable (installable) "artifact" AKA *wheel* file. A distribution package can be used to install modules, import packages, scripts, or any combination of those items.

```
pip install distribution-package
```

**import package**

An installable module, usually implemented as a folder that contains one or more module files.

```
import import_package
```

**PEP**

Python Enhancement Proposal — a document that describes some aspect of Python. Similar to RFCs in the Internet world.

**pip**

The standard tool to install a Python package.

**script**

An executable Python script that is installed in the `scripts` (Windows) or `bin` (Mac/Linux) folder of your

Python installation

**toml**

A file format similar to INI that is used for describing projects.

**wheel**

A file that contains everything needed to install a package

[1] Not to be confused with "package", which is a folder used to organize modules. (Well, maybe to be confused a little).

# Project layout

A typical project has several parts: source code, documentation, tests, and metadata. These can be laid out in different ways, but most people either do a *flat* layout or a *src* layout.

A *flat* layout has the code in the top level of the project, and a *src* layout has code in a separate folder named `src`.

In the long run, the *src* layout seems to be the most readable, and that makes it the best practice.

Metadata goes in the `pyproject.toml` file.

Unit tests go in a folder named `tests`. Documentation, using a tool such as **Sphinx**, goes in a folder named `docs`.

You can add any other files or folders necessary for your project.

## Typical layout

```
temperature
├── README.md
├── docs
│   ├── Makefile
│   ├── make.bat
│   └── source
│       ├── _static
│       ├── _templates
│       ├── conf.py
│       └── index.rst
├── pyproject.toml
├── src
│   └── temperature.py
└── tests
    └── test_temperature.py
```

the name of the project folder can be anything, but is typically the name of of the module or package you are creating.

The layouts on the following pages are not the only possibilities. You can combine them in whatever way works for your project.

# Sample Project layouts

## Module

```
MODULE_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml
 ├──── src
 │     └──── mymodule.py
 └──── tests
```

Code in mymodule.py will run as follows:

```
# import module (in a script)
# __name__ set to "mymodule"
import mymodule
from mymodule import MyClass, myfunction

# execute module (from command line)
# __name__ set to "__main__"
python mymodule.py
python -m mymodule
```

## Module with callable scripts

```
MODULE_SCRIPTS_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml  # script names mapped to mymodule:function
 ├──── src
 │     └──── mymodule.py  # functions called by scripts
|        |──── function1()
|        |──── _function1_wrapper()
 └──── tests
```

In `pyproject.toml`

```
[project.scripts]
myscript='mymodule:_function1_wrapper'
```

Module can be run or imported normally as above, plus scripts defined in `pyproject.toml` can run

directly from the command line

```
myscript
```

## Package

```
PACKAGE_PROJECT
├──── README.md
├──── docs
├──── pyproject.toml
├──── src
│    └──── mypackage
│        ├──── mymodule1.py
│        └──── mymodule2.py
└──── tests
```

```
from mypackage import mymodule1
from mypackage.mymodule1 import MyClass, myfunction
```

## Package with subpackages

```
PACKAGE_PROJECT
├──── README.md
├──── docs
├──── pyproject.toml
├──── src
│    └──── mypackage
│        └──── mysubpackage1
│            ├──── mymodule1.py
│            └──── mymodule2.py
│        └──── mysubpackage2
│            ├──── mymodule3.py
│            └──── mymodule4.py
└──── tests
```

```
from mypackage.subpackage1 import mymodule1
from mypackage.subpackage2.mymodule3 import MyClass, myfunction
```

## Package callable with `python -m packagename`

```
PACKAGE_APP_PROJECT
├──── README.md
├──── docs
├──── pyproject.toml
├──── src
│    └──── mypackage
│        ├──── __main__.py  # module executed by  python  -m packagename
│        ├──── mymodule1.py    # code to support main module
│        └──── mymodule2.py    # code to support main module
└──── tests
```

```
python -m mypackage
```

# python -m *NAME*

- python -m module
  - executes entire module
- python -m package
  - executes module `__main__.py` in package
- includes `if __name__ == "__main__"` section

The command `python -m NAME` is designed to execute a module or package without knowing its exact location. It uses the module search mechanism to find and load the module. This searches the folders in `sys.path`.

If NAME is a package, it executes the module named `__main__.py` in the top level of the package.

If NAME is a module, it executes the entire module.

In both cases, the code in the `if __name__ == "__main__"` block (if any) is executed. If a module is imported, that code is not executed.

A list of standard modules that have a command line interface via `python -m` is here: https://docs.python.org/3/library/cmdline.html

# Invoking Python

Assume the following code layout for the examples in the table

```
spam.py
ham
 ├──── __main__.py
 └──── toast.py
```

| Invocation | Description | Example |
|---|---|---|
| *FILE and FOLDER specified as pathnames* | | |
| `python FILE` | Run all code in `FILE` | `python spam.py`<br>`python ham/toast.py` |
| `python FOLDER` | If `__main__.py` exists in `FOLDER`, run all code in `__main__.py`; otherwise, raise error | `python ham` |
| *MODULE and PACKAGE found using sys.path* | | |
| `python -m MODULE` | Run all code in `MODULE` | `python -m spam` |
| `python -m PACKAGE` | Run all code in `PACKAGE.__init__.py`<br>Run all code in `__main__` | `python -m ham` |
| `python -m PACKAGE.MODULE` | Run all code in `PACKAGE.__init__.py`<br>Run all code in `PACKAGE.MODULE` | `python -m ham.toast` |

# Cookiecutter

- Creates standard layout
- Developed for Django
- Very flexible

**cookiecutter** is a utility written by Audrey and Roy Greenfeld to make it easy to replicate a standard setup for Django. However, it can be used create a layout for any type of project.

The `cookiecutter` command prompts you for information, then creates the project folder.

It uses a cookiecutter *template*, which is a folder, to create the new project. There are many templates on **github** to choose from, and you can easily create your own.

Syntax is

```
cookiecutter template-folder
```

The script copies the template layout (all folders and files) to a new folder which is the "slug" (short name) of your project. It inserts your project name in the appropriate places. It will do this in both file names and file contents.

There are two **cookiecutter** templates provided in the SETUP folder of the student files to generate the layouts on the previous pages:

- `cookiecutter-python-module`
- `cookiecutter-python-package`

The project layouts will be generated based on answers to the cookiecutter questions.

cookiecutter home page: https://github.com/audreyr/cookiecutter
cookiecutter docs: https://cookiecutter.readthedocs.io

Feel free to copy the cookiecutter templates and modify them for your own projects.

Another useful tool for generating Python projects is **PyScaffold**. Details at https://pyscaffold.org/en/stable/index.html.

**cookiecutter-python-package/cookiecutter.json**

```
{
    "package_name": "Package Name (can have spaces)",
    "package_slug": "{{ cookiecutter.package_name.lower().replace(' ','').replace('-
','_') }}",
    "package_description": "Short Description of the Package",
    "module_slug": "{{ cookiecutter.package_slug }}",
    "has_scripts": "n",
    "has_main": "n",
    "author_name": "Author Name",
    "author_email": "noone@nowhere.com",
    "author_url": "Author URL",
    "copyright_year": "2024",
    "readme_format": ["md", "rst"]
}
```

***tree cookiecutter-python-package***

```
/Users/jstrick/curr/courses/python/common/setup/cookiecutter-python-package
├──── cookiecutter.json
├──── hooks
│    ├──── post_gen_project.py
│    └──── pre_gen_project.py
└──── {{cookiecutter.package_slug}}
    ├──── README.{{cookiecutter.readme_format}}
    ├──── docs
    │    ├──── Makefile
    │    ├──── make.bat
    │    └──── source
    │        ├──── _static
    │        ├──── _templates
    │        ├──── conf.py
    │        └──── index.rst
    ├──── pyproject.toml
    ├──── src
    │    └──── {{cookiecutter.package_slug}}
    │        ├──── __init__.py
    │        ├──── __main__.py
    │        └──── {{cookiecutter.module_slug}}.py
    └──── tests
        ├──── __init__.py
        └──── test_{{cookiecutter.module_slug}}.py

9 directories, 14 files
```

***cookiecutter cookiecutter-python-package***

```
[1/11] package_name (Package Name (can have spaces)): Log Processor
[2/11] package_slug (logprocessor): logproc
[3/11] package_description (Short Description of the Package): Process Log Files
[4/11] module_slug (logproc):
[5/11] has_scripts (n): y
[6/11] has_main (n): y
[7/11] author_name (Author Name): Sabrina Q. Programmer
[8/11] author_email (noone@nowhere.com): sabrinaq@gmail.com
[9/11] author_url (Author URL): https://www.sabrinaq.com
[10/11] copyright_year (2024):
[11/11] Select readme_format
1 - md
2 - rst
Choose from [1/2] (1): 1
```

***tree logproc***

```
logproc
├──── README.md
├──── docs
│    ├──── Makefile
│    ├──── make.bat
│    └──── source
│        ├──── _static
│        ├──── _templates
│        ├──── conf.py
│        └──── index.rst
├──── pyproject.toml
├──── src
│    ├──── logproc
│        ├──── __init__.py
│        ├──── __main__.py
│        └──── logproc.py
└──── tests
    ├──── __init__.py
    └──── test_logproc.py
```

# Defining project metadata

- Create `pyproject.toml`
- Use `build` to build the package

The **modern** way to package a Python project is using the `pyproject.toml` config file. The specifications that support this are specified in **PEP 518** and **PEP 621**.

The **TOML** format is similar to `.ini` files, but adds some features.

The first part of the file is always required. It tells the `build` program what tools to use.

```
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"
```

Put all the project metadata that the build system will need to package and install your project after the `[build-system]` section.

```
[project]
name = "logproc"
version = "1.0.0"
authors = [
    { name="Author Name", email="sabrinaq@gmail.com" },
]
description = "Short Description of the Package"
readme = "README.rst"
requires-python = ">=3.0"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

dependencies = [
    'requests[security] < 3',
]
```

TOML value types arrays are similar to Python `list` and TOML tables (including inline) are similar to `dict`.

The rest of the file after the `[build-system]` section is not needed if you are also using `setup.cfg` and `setup.py`. However, best practice is to *not* use those legacy files, as all the data needed for building the package, installing it, and uploading it to **PyPI** can be contained in `pyproject.toml`, and can then be used by nearly any build *backend*.

# Packages with scripts

- Provide utility scripts
- Run from command line
- Installed in `⋯/scripts` or `⋯/bin`
- Add config to `pyproject.toml`

It is easy to add one or more command-line scripts to your project. These scripts are created in the `scripts` (Windows) or `bin` (other OS) folders of your Python installation. While they require Python to be installed, they are run like any other command.

The scripts are based on functions in the module. Since the scripts are run from the CLI, they are not called with normal parameters. Instead, the functions access `sys.argv` for arguments, like any standalone Python script.

```
def _c2f_cli():
    """
    CLI utility script
    Called from command line as 'c2f'
    """
    cel = float(sys.argv[1])
    return c2f(cel)


def _f2c_cli():
    """
    CLI utility script
    Called from command line as 'f2c'
    """
    fahr = float(sys.argv[1])
    return f2c(fahr)
```

To configure scripts, add a section like the following to `pyproject.toml`. The names on the left are the installed script names. The values on the right are the module name and the function name, separated by a colon.

```
[project.scripts]
c2f = 'temperature:_c2f_cli'
f2c = 'temperature:_f2c_cli'
```

See the project `temperature_scripts` in EXAMPLES for details.

# Editable installs

- Use `pip install -e package`
- Puts a link in library folder
- Allows testing as though module is installed

When using a `src` (or other name) folder for your codebase and `tests` for your test scripts, the tests need to find your package. While you could put the path to the `src` folder in PYTHONPATH, the best practice is to do an *editable install*.

This is an install that uses the path to your development folder. It achieves this by using a virtual environment. Then you can run your tests after making changes to your code, without having to reinstall the package.

From the top level folder of the project, type the following (you do not have to build the distribution for this step).

```
pip install -e .
```

Now the project is installed and is available to import or run like any other installed module.

# Running unit tests

- Use editable install
- Just use `pytest` or `pytest -v`

To run the tests that you have created in the `tests` folder, just run `pytest` or `pytest -v` (verbose) in the top level folder of the project. Because the project was installed with an editable install, tests can import the module or package normally.

## Example

```
$ pytest -v
============================================================ test session starts
============================================================
platform darwin -- Python 3.9.17, pytest-7.1.2, pluggy-1.0.0 -- /Users/jstrick/opt/miniconda3/bin/python
cachedir: .pytest_cache
PyQt5 5.15.7 -- Qt runtime 5.15.2 -- Qt compiled 5.15.2
hypothesis profile 'default' ->
database=DirectoryBasedExampleDatabase('/Users/jstrick/curr/courses/python/common/examples/temperature/.hypothesis/examples')
rootdir: /Users/jstrick/curr/courses/python/common/examples/temperature
plugins: anyio-3.6.1, qt-4.1.0, remotedata-0.3.3, assert-utils-0.3.1, lambda-2.1.0, astropy-header-0.2.1, fixture-order-
0.1.4, common-subject-1.0.6, mock-3.8.2, typeguard-2.13.3, astropy-0.10.0, filter-subpackage-0.1.1, hypothesis-6.54.3,
openfiles-0.5.0, django-4.5.2, doctestplus-0.12.0, cov-3.0.0, arraydiff-0.5.0
collected 8 items

tests/test_temperature.py::test_c2f[100-212] PASSED
[ 12%]
tests/test_temperature.py::test_c2f[0-32] PASSED
[ 25%]
tests/test_temperature.py::test_c2f[37-98.6] PASSED
[ 37%]
tests/test_temperature.py::test_c2f[-40--40] PASSED
[ 50%]
tests/test_temperature.py::test_f2c[212-100] PASSED
[ 62%]
tests/test_temperature.py::test_f2c[32-0] PASSED
[ 75%]
tests/test_temperature.py::test_f2c[98.6-37] PASSED
[ 87%]
tests/test_temperature.py::test_f2c[-40--40] PASSED
[100%]

============================================================ 8 passed in 0.20s
============================================================
```

# Wheels

- 3 kinds of wheels
  - Universal wheels (pure Python; python 2 *and* 3 compatible
  - Pure Python wheels (pure Python; Python 2 *or* 3 compatible
  - Platform wheels (Platform-specific; binary)

A wheel is prebuilt distribution. Wheels can be installed with pip.

A *Universal wheel* is a pure Python package (no extensions) that can be installed on either Python 2 or Python 3. It has to have been carefully written that way.

A *Pure Python wheel* is a pure Python package that is specific to one version of Python (either 2 or 3). It can only be installed by a matching version of pip.

A *Platform wheel* is a package that has extensions, and thus is platform-specific.

Build systems automatically create the correct wheel type.

# Building distributions

- `python -m build`
- Creates `dist` folder
- Binary distribution
  - `package-version.whl`
- Source distribution
  - `package-version.tar.gz`

To build the project, use

```
python -m build
```

This will create the wheel file (binary distribution) and a gzipped tar file (source distribution) in a folder named `dist`.

***python -m build***

```
* Creating virtualenv isolated environment...
* Installing packages in isolated environment... (setuptools>=61.0)
* Getting build dependencies for sdist...
running egg_info
writing src/temperature.egg-info/PKG-INFO
writing dependency_links to src/temperature.egg-info/dependency_links.txt
writing top-level names to src/temperature.egg-info/top_level.txt
reading manifest file 'src/temperature.egg-info/SOURCES.txt'
writing manifest file 'src/temperature.egg-info/SOURCES.txt'
* Building sdist...
```

*... about 70 lines of output ...*

```
running install_scripts
creating build/bdist.macosx-10.9-universal2/wheel/temperature-1.0.0.dist-info/WHEEL
creating '/Users/jstrick/curr/courses/python/common/examples/temperature/dist/.tmp-
538d2_ts/temperature-1.0.0-py3-none-any.whl' and adding 'build/bdist.macosx-10.9-
universal2/wheel' to it
adding 'temperature.py'
adding 'temperature-1.0.0.dist-info/METADATA'
adding 'temperature-1.0.0.dist-info/WHEEL'
adding 'temperature-1.0.0.dist-info/top_level.txt'
adding 'temperature-1.0.0.dist-info/RECORD'
removing build/bdist.macosx-10.9-universal2/wheel
Successfully built temperature-1.0.0.tar.gz and temperature-1.0.0-py3-none-any.whl
```

# Installing a package

- Use `pip`
  - many options
  - can install just for user

A wheel makes installing packages simple. You can just use

```
pip install package.whl
```

This will install the package in the standard location for the current version of Python.

If you do not have permission to install modules in the standard location, you can do a user install, which installs modules under your home folder.

```
pip install --user package.whl
```

# For more information

**Python Packaging User Guide**

https://packaging.python.org/en/latest/

**Distributing Python Modules**

https://docs.python.org/3/distributing/index.html

**setuptools Quickstart**

https://setuptools.pypa.io/en/latest/userguide/quickstart.html

**Thoughts on the Python packaging ecosystem**

https://pradyunsg.me/blog/2023/01/21/thoughts-on-python-packaging/

**THE BASICS OF PYTHON PACKAGING IN EARLY 2023**

https://drivendata.co/blog/python-packaging-2023

**Structuring Your Project (from The Hitchhiker's Guide to Python)**

https://docs.python-guide.org/writing/structure/

# Chapter 1 Exercises

## Exercise 1-1 (carddeck/*)

Step 1

Create a distributable module named `carddeck` from the `carddeck.py` and `card.py` modules in the root folder of the student files.

HINT: To do it the easy way, use the `cookiecutter-python-module` template. Add the two source files to the `src` folder. (remove any existing sample Python scripts in `src`).

To do it the "hard" way, create the project layout by hand, create the `pyproject.toml` file, etc.

Step 2

Build a distribution (wheel file).

Step 3

Install the wheel file with `pip`. (The cookiecutter template automatically does an editable install)

Step 4

Then import the new module and create an instance of the `CardDeck` class. Shuffle the cards, and deal out all 52 cards.

# Index