Packaging Python Projects

This tutorial walks you through how to package a simple Python project. It will show you how to add the necessary files and structure to create the package, how to build the package, and how to upload it to the Python Package Index.

Tip: If you have trouble running the commands in this tutorial, please copy the command and its output, then open an issue on the packaging-problems repository on GitHub. We'll do our best to help you!

Some of the commands require a newer version of pip, so start by making sure you have the latest version installed:

Unix/macOS Windows

```
python3 -m pip install --upgrade pip
```

A simple project

This tutorial uses a simple project named `example_package`. We recommend following this tutorial as-is using this project, before packaging your own project.

Create the following file structure locally:

```
packaging_tutorial/
└── src/
    └── example_package/
        ├── __init__.py
        └── example.py
```

`__init__.py` is required to import the directory as a package, and should be empty.

`example.py` is an example of a module within the package that could contain the logic (functions, classes, constants, etc.) of your package. Open that file and enter the following content:

```
def add_one(number):
    return number + 1
```

If you are unfamiliar with Python's modules and import packages, take a few minutes to read over the Python documentation for packages and modules.

Once you create this structure, you'll want to run all of the commands in this tutorial within the `packaging_tutorial` directory.

Creating the package files

You will now add files that are used to prepare the project for distribution. When you're done, the project structure will look like this:

```
packaging_tutorial/
├── LICENSE
├── pyproject.toml
├── README.md
├── setup.cfg
├── src/
│   └── example_package/
│       ├── __init__.py
│       └── example.py
└── tests/
```

`tests/` is a placeholder for test files. Leave it empty for now.

Creating pyproject.toml

`pyproject.toml` tells build tools (like pip and build) what is required to build your project. This tutorial uses setuptools, so open `pyproject.toml` and enter the following content:

```
[build-system]
requires = ["setuptools>=42"]
build-backend = "setuptools.build_meta"
```

`build-system.requires` gives a list of packages that are needed to build your package. Listing something here will *only* make it available during the build, not after it is installed.

`build-system.build-backend` is the name of Python object that will be used to perform the build. If you were to use a different build system, such as flit or poetry, those would go here, and the configuration details would be completely different than the setuptools configuration described below.

See **PEP 517** and **PEP 518** for background and details.

Configuring metadata

There are two types of metadata: static and dynamic.

- Static metadata (`setup.cfg`): guaranteed to be the same every time. This is simpler, easier to read, and avoids many common errors, like encoding errors.
- Dynamic metadata (`setup.py`): possibly non-deterministic. Any items that are dynamic or determined at install-time, as well as extension modules or extensions to setuptools, need to go into `setup.py`.

Static metadata (`setup.cfg`) should be preferred. Dynamic metadata (`setup.py`) should be used only as an escape hatch when absolutely necessary. `setup.py` used to be required, but can be omitted with newer versions of setuptools and pip.

`setup.cfg` **(static)** `setup.py` **(dynamic)**

`setup.cfg` is the configuration file for setuptools. It tells setuptools about your package (such as the name and version) as well as which code files to include. Eventually much of this configuration may be able to move to `pyproject.toml`.

Open `setup.cfg` and enter the following content. Change the `name` to include your username; this ensures that you have a unique package name and that your package doesn't conflict with packages uploaded by other people following this tutorial.

```
[metadata]
name = example-package-YOUR-USERNAME-HERE
version = 0.0.1
author = Example Author
author_email = author@example.com
description = A small example package
long_description = file: README.md
long_description_content_type = text/markdown
url = https://github.com/pypa/sampleproject
project_urls =
    Bug Tracker = https://github.com/pypa/sampleproject/issues
classifiers =
    Programming Language :: Python :: 3
    License :: OSI Approved :: MIT License
    Operating System :: OS Independent

[options]
package_dir =
    = src
packages = find:
```

```
[options.packages.find]
where = src
```

There are a variety of metadata and options supported here. This is in configparser format; do not place quotes around values. This example package uses a relatively minimal set of `metadata`:

- `name` is the *distribution name* of your package. This can be any name as long as it only contains letters, numbers, `_` , and `-`. It also must not already be taken on pypi.org. **Be sure to update this with your username,** as this ensures you won't try to upload a package with the same name as one which already exists.
- `version` is the package version. See **PEP 440** for more details on versions. You can use `file:` or `attr:` directives to read from a file or package attribute.
- `author` and `author_email` are used to identify the author of the package.
- `description` is a short, one-sentence summary of the package.
- `long_description` is a detailed description of the package. This is shown on the package detail page on the Python Package Index. In this case, the long description is loaded from `README.md` (which is a common pattern) using the `file:` directive.
- `long_description_content_type` tells the index what type of markup is used for the long description. In this case, it's Markdown.
- `url` is the URL for the homepage of the project. For many projects, this will just be a link to GitHub, GitLab, Bitbucket, or similar code hosting service.
- `project_urls` lets you list any number of extra links to show on PyPI. Generally this could be to documentation, issue trackers, etc.
- `classifiers` gives the index and pip some additional metadata about your package. In this case, the package is only compatible with Python 3, is licensed under the MIT license, and is OS-independent. You should always include at least which version(s) of Python your package works on, which license your package is available under, and which operating systems your package will work on. For a complete list of classifiers, see https://pypi.org/classifiers/.

In the `options` category, we have controls for setuptools itself:

- `package_dir` is a mapping of package names and directories. An empty package name represents the "root package" — the directory in the project that contains all Python source files for the package — so in this case the `src` directory is designated the root package.
- `packages` is a list of all Python import packages that should be included in the distribution package. Instead of listing each package manually, we can use the `find:` directive to automatically discover all packages and subpackages and `options.packages.find` to specify the `package_dir` to use. In this case, the list of packages will be `example_package` as that's the only package present.
- `python_requires` gives the versions of Python supported by your project. Installers like pip will look back through older versions of packages until it finds one that has a matching Python version.

There are many more than the ones mentioned here. See Packaging and distributing projects for more details.

Creating README.md

Open `README.md` and enter the following content. You can customize this if you'd like.

```
# Example Package

This is a simple example package. You can use
[Github-flavored Markdown](https://guides.github.com/features/mastering-markdown/)
to write your content.
```

Because our configuration loads `README.md` to provide a `long_description`, `README.md` must be included along with your code when you generate a source distribution. Newer versions of setuptools will do this automatically.

Creating a LICENSE

It's important for every package uploaded to the Python Package Index to include a license. This tells users who install your package the terms under which they can use your package. For help picking a license, see https://choosealicense.com/.

```
Copyright (c) 2018 The Python Packaging Authority

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

Including other files

The files listed above will be included automatically in your source distribution. If you want to control what goes in this explicitly, see Including files in source distributions with MANIFEST.in.

The final built distribution will have the Python files in the discovered or listed Python packages. If you want to control what goes here, such as to add data files, see Including Data Files from the setuptools docs.

Generating distribution archives

The next step is to generate distribution packages for the package. These are archives that are uploaded to the Python Package Index and can be installed by pip.

Make sure you have the latest version of PyPA's build installed:

Unix/macOS | Windows

```
python3 -m pip install --upgrade build
```

Tip: If you have trouble installing these, see the Installing Packages tutorial.

Now run this command from the same directory where `pyproject.toml` is located:

Unix/macOS | Windows

```
python3 -m build
```

This command should output a lot of text and once completed should generate two files in the `dist` directory:

```
dist/
  example-package-YOUR-USERNAME-HERE-0.0.1-py3-none-any.whl
  example-package-YOUR-USERNAME-HERE-0.0.1.tar.gz
```

The `tar.gz` file is a source archive whereas the `.whl` file is a built distribution. Newer pip versions preferentially install built distributions, but will fall back to source archives if needed. You should always upload a source archive and provide built archives for the platforms your project is compatible with. In this case, our example package is compatible with Python on any platform so only one built distribution is needed.

Uploading the distribution archives

The first thing you'll need to do is register an account on TestPyPI, which is a separate instance of the package index intended for testing and experimentation. It's great for things like this tutorial where we don't necessarily want to upload to the real index. To register an account, go to https://test.pypi.org/account/register/ and complete the steps on that page. You will also need to verify your email address before you're able to upload any packages. For more details, see Using TestPyPI.

To securely upload your project, you'll need a PyPI API token. Create one at https://test.pypi.org/manage/account/#api-tokens, setting the "Scope" to "Entire account". **Don't close the page until you have copied and saved the token — you won't see that token again.**

Now that you are registered, you can use twine to upload the distribution packages. You'll need to install Twine:

Unix/macOS Windows

```
python3 -m pip install --upgrade twine
```

Once installed, run Twine to upload all of the archives under `dist`:

Unix/macOS Windows

```
python3 -m twine upload --repository testpypi dist/*
```

You will be prompted for a username and password. For the username, use `__token__`. For the password, use the token value, including the `pypi-` prefix.

After the command completes, you should see output similar to this:

```
Uploading distributions to https://test.pypi.org/legacy/
Enter your username: [your username]
Enter your password:
Uploading example-package-YOUR-USERNAME-HERE-0.0.1-py3-none-any.whl
100%|████████████████████████████████| 4.65k/4.65k [00:01<00:00, 2.88kB/s]
Uploading example-package-YOUR-USERNAME-HERE-0.0.1.tar.gz
100%|████████████████████████████████| 4.25k/4.25k [00:01<00:00, 3.05kB/s]
```

Once uploaded your package should be viewable on TestPyPI, for example, https://test.pypi.org/project/example-package-YOUR-USERNAME-HERE

Installing your newly uploaded package

You can use pip to install your package and verify that it works. Create a virtual environment and install your package from TestPyPI:

Unix/macOS Windows

```
python3 -m pip install --index-url https://test.pypi.org/simple/ --no-deps example-package-YOUR-USERNAME-HERE
```

Make sure to specify your username in the package name!

pip should install the package from TestPyPI and the output should look something like this:

```
Collecting example-package-YOUR-USERNAME-HERE
  Downloading https://test-files.pythonhosted.org/packages/.../example-package-YOUR-USERNAME-HERE-0.0.1-py3-n
Installing collected packages: example-package-YOUR-USERNAME-HERE
Successfully installed example-package-YOUR-USERNAME-HERE-0.0.1
```

may fail or install something unexpected. While our example package doesn't have any dependencies, it's a good practice to avoid installing dependencies when using TestPyPI.

You can test that it was installed correctly by importing the package. Make sure you're still in your virtual environment, then run Python:

Unix/macOS Windows

```
python3
```

and import the package:

```
>>> from example_package import example
>>> example.add_one(2)
3
```

Note that the import package is `example_package` regardless of what `name` you gave your distribution package in `setup.cfg` or `setup.py` (in this case, `example-package-YOUR-USERNAME-HERE`).

Next steps

Congratulations, you've packaged and distributed a Python project! ✨ 📦 ✨

Keep in mind that this tutorial showed you how to upload your package to Test PyPI, which isn't a permanent storage. The Test system occasionally deletes packages and accounts. It is best to use TestPyPI for testing and experiments like this tutorial.

When you are ready to upload a real package to the Python Package Index you can do much the same as you did in this tutorial, but with these important differences:

- Choose a memorable and unique name for your package. You don't have to append your username as you did in the tutorial.
- Register an account on https://pypi.org - note that these are two separate servers and the login details from the test server are not shared with the main server.
- Use `twine upload dist/*` to upload your package and enter your credentials for the account you registered on the real PyPI. Now that you're uploading the package in production, you don't need to specify `--repository`; the package will upload to https://pypi.org/ by default.
- Install your package from the real PyPI using `python3 -m pip install [your-package]`.

At this point if you want to read more on packaging Python libraries here are some things you can do:

- Read more about using setuptools to package libraries in Packaging and distributing projects.
- Read about Packaging binary extensions.
- Consider alternatives to setuptools such as flit, hatch, and poetry.

---

[1] Some legacy Python environments may not have `setuptools` pre-installed, and the operators of those environments may still be requiring users to install packages by running `setup.py install` commands, rather than providing an installer like `pip` that automatically installs required build dependencies. These environments will not be able to use many published packages until the environment is updated to provide an up to date Python package installation client (e.g. by running `python -m ensurepip`).