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 (PyPI).

> **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_YOUR_USERNAME_HERE`. If your username is `me`, then the package would be `example_package_me`; this ensures that you have a unique package name that doesn't conflict with packages uploaded by other people following this tutorial. 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_YOUR_USERNAME_HERE/
        ├── __init__.py
        └── example.py
```

The directory containing the Python files should match the project name. This simplifies the configuration and is more obvious to users who install the package.

Creating the file `__init__.py` is recommended because the existence of an `__init__.py` file allows users to import the directory as a regular package, even if (as is the case in this tutorial) `__init__.py` is empty. [1]

`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

```
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
├── src/
│   └── example_package_YOUR_USERNAME_HERE/
│       ├── __init__.py
│       └── example.py
└── tests/
```

Creating a test directory

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

Choosing a build backend

Tools like pip and build do not actually convert your sources into a distribution package (like a wheel); that job is performed by a build backend. The build backend determines how your project will specify its configuration, including metadata (information about the project, for example, the name and tags that are displayed on PyPI) and input files. Build backends have different levels of functionality, such as whether they support building extension modules, and you should choose one that suits your needs and preferences.

You can choose from a number of backends; this tutorial uses Hatchling by default, but it will work identically with Setuptools, Flit, PDM, and others that support the `[project]` table for metadata.

Note

Some build backends are part of larger tools that provide a command-line interfac
project initialization and version management, as well as building, uploading, and installing packages. This tutorial uses single-purpose tools that work independently.

The `pyproject.toml` tells build frontend tools like pip and build which backend to use for your project. Below are some examples for common build backends, but check your backend's own documentation for more details.

Hatchling | setuptools | Flit | PDM

```toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

The `requires` key is a list of packages that are needed to build your package. The frontend should install them automatically when building your package. Frontends usually run builds in isolated environments, so omitting dependencies here may cause build-time errors. This should always include your backend's package, and might have other build-time dependencies.

The `build-backend` key is the name of the Python object that frontends will use to perform the build.

Both of these values will be provided by the documentation for your build backend, or generated by its command line interface. There should be no need for you to customize these settings.

Additional configuration of the build tool will either be in a `tool` section of the `pyproject.toml`, or in a special file defined by the build tool. For example, when using `setuptools` as your build backend, additional configuration may be added to a `setup.py` or `setup.cfg` file, and specifying `setuptools.build_meta` in your build allows the tools to locate and use these automatically.

Configuring metadata

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

```toml
[project]
name = "example_package_YOUR_USERNAME_HERE"
version = "0.0.1"
authors = [
  { name="Example Author", email="author@example.com" },
]
description = "A small example package"
readme = "README.md"
requires-python = ">=3.8"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

[project.urls]
Homepage = "https://github.com/pypa/sampleproject"
Issues = "https://github.com/pypa/sampleproject/issues"
```

- `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. **Be sure to update this with your username** for this tutorial, 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. (Some build backends allow it to be specified another way, such as from a file or Git tag.)
- `authors` is used to identify the author of the package; you specify a name and an email for each author. You can also list `maintainers` in the same format.
- `description` is a short, one-sentence summary of the package.
- `readme` is a path to a file containing a detailed description of the package. This is shown on the package detail page on PyPI. In this case, the description is loaded from `README.md` (which is a common pattern). There also is a more advanced table form described in the pyproject.toml guide.
- `requires-python` gives the versions of Python supported by your project. An installer like pip will look back through older versions of packages until it finds one that has a matching Python version.
- `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/.
- `urls` lets you list any number of extra links to show on PyPI. Generally this could be to the source, documentation, issue trackers, etc.

See the pyproject.toml guide for details on these and other fields that can be defined in the `[project]` table. Other common fields are `keywords` to improve discoverability and the

`dependencies` that are required to install your package.

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.
```

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/. Once you have chosen a license, open `LICENSE` and enter the license text. For example, if you had chosen the MIT license:

```
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.
```

Most build backends automatically include license files in packages. See your backend's documentation for more details.

Including other files

The files listed above will be included automatically in your source d... include additional files, see the documentation for your build backend.

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 distribution whereas the `.whl` file is a built distribution. Newer pip versions preferentially install built distributions, but will fall back to source distributions if needed. You should always upload a source distribution and provide built distributions 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

Finally, it's time to upload your package to the Python Package Index!

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 [illegible] 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: __token__
Uploading example_package_YOUR_USERNAME_HERE-0.0.1-py3-none-any.whl
100% ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8.2/8.2 kB • 00:01 • ?
Uploading example_package_YOUR_USERNAME_HERE-0.0.1.tar.gz
100% ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.8/6.8 kB • 00:00 • ?
```

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-YOU
```

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_H
Installing collected packages: example_package_YOUR_USERNAME_HERE
Successfully installed example_package_YOUR_USERNAME_HERE-0.0.1
```

Note

This example uses `--index-url` flag to specify TestPyPI instead of live PyPI. Additionally, it specifies `--no-deps`. Since TestPyPI doesn't have the same packages as the live PyPI, it's possible that attempting to install dependencies 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_YOUR_USERNAME_HERE import example
>>> example.add_one(2)
3
```

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, but you can't use an existing name.
- 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 ht
- 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 about advanced configuration for your chosen build backend: Hatchling, setuptools, Flit, PDM.
- Look at the guides on this site for more advanced practical information, or the discussions for explanations and background on specific topics.
- Consider packaging tools that provide a single command-line interface for project management and packaging, such as hatch, flit, pdm, and poetry.

Notes

[1] Technically, you can also create Python packages without an `__init__.py` file, but those are called namespace packages and considered an **advanced topic** (not covered in this tutorial). If you are only getting started with Python packaging, it is recommended to stick with *regular packages* and `__init__.py` (even if the file is empty).