PyPi Package Example Documentation

Release 1.0.0

Paul Vincent Craven

Mar 02, 2020

Contents

Getting started creating a Python package is very simple. Your project only needs two files, and a directory.

Module vs. Package

A Python *module* is any file with Python code in it. A Python *package* has multiple Python modules. A package has a `__init__.py` file in a directory of Python modules.

Once you've got your package started, you can expand on it and add more features, testing, documentation, and more.

This documentation shows how to build a modern Python package. If you already have a package and want to know what each file does, see *Example Directory Tree*.

The source for this example is at:

https://pypi-package-example.readthedocs.io/

CHAPTER 1

Documentation

1.1 Core Files and Directories

There are only two required files, and one required directory:

- `/pypi_package_example/` This is the main project directory where all the Python source code goes. The directory should be named the same as your package name. Check the PyPi Package Index to make sure there isn't a conflict before picking your package name.
- `/pypi_package_example/__init__.py` This is the starting file for your project. It is run when you import your package. It should not do much processing, it should just load in all the functions and classes that you plan on using in your project.

1.1.1 Setup File

- `/setup.py` or `/setup.cgf`. This specifies how your project is to be built, and other meta information about the project. The `/setup.py` seems more common based on my limited experience, but in 2016 PEP 518 was provisionally accepted which specifies a different setup method, to be stored in a file called `setup.cfg`.

1.1.2 Setup File In Detail

When you run setup.py, you can get a full list of commands:

Listing 1: setup.py options

```
(venv) C:\pypi_package_example>python 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++ extensions (compile/link to build directory)
  build_clib       build C/C++ libraries used by Python extensions
  build_scripts    "build" scripts (copy and fixup #! line)
```

(continues on next page)

(continued from previous page)

```
  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:
  bdist_wheel        create a wheel distribution
  build_sphinx       Build Sphinx documentation
  flake8             Run Flake8 on modules registered in setup.py
  compile_catalog    compile message catalogs to binary MO files
  extract_messages   extract localizable strings from the project code
  init_catalog       create a new catalog based on a POT file
  update_catalog     update message catalogs from a POT file
  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
  upload_docs        Upload documentation to PyPI

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
```

The setup.py file itself can be pretty simple. As it is Python, you can keep adding onto it as your project gets more complex and you need more customization. See the setup.py documentation for an idea of what you can do with that file.

Listing 2: setup.py

```python
#!/usr/bin/env python

from os import path
from setuptools import setup

VERSION = "1.0.0"

if __name__ == "__main__":

```

(continues on next page)

(continued from previous page)

```
    # List of all the required packages.
    install_requires = [
        "arcade",
    ]

    # Grab the long description out of the README
    fname = path.join(path.dirname(path.abspath(__file__)), "README.rst")
    with open(fname, "r") as f:
        long_desc = f.read()

    # Call the setup function with our setup parameters.
    # This kicks off the build.
    setup(
        name="arcade",
        version=VERSION,
        description="Sample Python Package",
        long_description=long_desc,
        author="Paul Vincent Craven",
        author_email="paul.craven@simpson.edu",
        license="MIT",
        url="https://pypi-package-example.readthedocs.io/en/latest/",
        install_requires=install_requires,
        packages=["pypi_package_example"],
        python_requires=">=3.6",
        classifiers=[
            "Development Status :: 5 - Production/Stable",
            "Intended Audience :: Developers",
            "License :: OSI Approved :: MIT License",
            "Operating System :: OS Independent",
            "Programming Language :: Python",
            "Programming Language :: Python :: 3.6",
            "Programming Language :: Python :: 3.7",
            "Programming Language :: Python :: 3.8",
            "Programming Language :: Python :: Implementation :: CPython",
            "Topic :: Software Development :: Libraries :: Python Modules",
        ],
        test_suite="tests",
        package_data={"arcade": ["examples/images/*.png"]},
        project_urls={
            "Documentation": "https://pypi-package-example.readthedocs.io/en/latest/",
            "Issue Tracker": "https://github.com/pvcraven/pypi_package_example/issues
↪",
            "Source": "https://github.com/pvcraven/pypi_package_example",
        },
    )
```

1.2 Git

If you are using git version control, you need a list of files and directories for git to ignore. This are saved in the `.gitignore` file.

GitHub maintains a great list of sample `.gitignore` files in there collection of useful .gitignore templates.

My `.gitignore` for Python typically looks like this: .gitignore.

Note: Please teach your students not to check in SSH keys! Also make sure they don't check in the results of a build. Go over a typical .gitignore so they understand why things should be, and should not be checked in.

1.3 GitHub and GitHub Community

If you host your files on GitHub, they have recommended supporting files for documentation and building a community.

1.3.1 Readme

`/README.md` or `/README.rst` This file is shown at the bottom of your GitHub page. It (along with most other files) can be in Markdown or Restructured Text format. Look at the bottom of the pypi_package_example GitHub site for this website's Read Me. You can click on the README.rst to see this file, and hit the 'Raw' button to see the source.

Badges - Badges are small graphics you can stick on your site to visually list information about your project. The README is a good spot for them. See this project's README.rst. For more info see *Badges*.

1.3.2 Code of Conduct

- `/CODE_OF_CONDUCT.md` Public projects should have a code of conduct. Add it before you need one, not after. See GitHub's Code of Conduct suggestions, and the Contributor Covenant for open source projects.

1.3.3 Contributing

- `/CONTRIBUTING.md` Encourage contributions to your project by telling interested developers how to get started.

1.3.4 License

- `/license.md` Post a license for how people can use the software. See Choose a License for help in selecting one. If other people contribute to your project, they do so under the license you publish. Changing the license on your project should involve the buy-in of all contributors. This isn't easy with a popular open-source project, so choose wisely the first time.

1.3.5 Bug Report Template

- `/.github/ISSUE_TEMPLATE/bug_report.md` When people report a bug, GitHub will use the contents of this file as a template. This file, and similar ones like `pull_request_template.md` have a few different places where GitHub will look for them. I like putting them in `.github` rather than pollute the root directory. You can also have multiple templates, depending on if it is a bug, feature request, etc.

1.3.6 Pull Request Template

- `/.github/PULL_REQUEST_TEMPLATE/pull_request_template.md` When developers create a pull request a bug, GitHub will use the contents of this file as a template.

1.4 Building and Deployment

1.4.1 Requirements

- `/requirements.txt` This should be a simple list of every package required to *develop* your project. The packages required to *run* the project go in setup.py. This file makes it easy for automatic setup of virtual environment, and automated builds.
- Python programs often use a virtual environment in a folder (usually named venv). Python has a lot of other tools like pipenv and more trying to solve this same problem. Last year, PEP 582 was approved that will use a `__pypackages__` directory that, if it exists, will be used instead of global packages.

1.4.2 Setup

- `/setup.py` This is one of the two required files. You can use the setup file to build the project. For more info, refer back to *Core Files and Directories*.

1.4.3 Make File

- `/make.bat`, `make.sh`, `make.py` There are so many different commands for building, testing, and deployment, I like having a "make" file with a instructions to make the process easier.

1.4.4 Build Directory

- `/build/` This is automatically created by setup.py when you build.

1.4.5 Distribution Directory

- `/dist/` This is automatically created by setup.py when we build wheels.

1.4.6 Additional Build Info

- The command to build your project is `python setup.py build`
- bdist / wheels - If you have the wheel package installed, you can create a one-file distribution of your project. If the project is pure Python, that wheel can work on any platform. If you've got platform-specific libraries, you can make wheels for each platform. See Python's packaging projects for more info. The command to create a wheel is `python setup.py bdist_wheel`. This only works if you have the wheel package installed.
- Manifest: https://packaging.python.org/guides/using-manifest-in/
- Twine - Once your project is packaged in a wheel, you can upload it to the PyPi repository for other people to use. This is done with the twine module. It is simple as:

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

- AWS - If you deploy on your own server, Amazon Web Services has a great Python-based command-line interface as part of the awscli package.

1.5 Package Documentation Files

Documenting your project is important if you want anyone else to use it. Documentation is done using a markup language that is then converted into HTML for your website.

To convert from a markup language to HTML we use a tool called Sphinx. Sphinx is a popular tool for creating web documentation for Python projects. It is part of a larger group of tools known as *Static Site Generators*. You can see a list of top static site generators at StaticGen.

Markdown (.md) and RestructuredText (.rst) - Static sites are normally written using either markdown or restructured text. Markdown is more popular in the whole eco-system of markup. RestructuredText is more popular for Python documentation. Restructured Text also allows inclusion of external files, which is GREAT for maintaining code samples. See examples of this at:

http://arcade.academy/examples

To get started with Sphinx, there's a sphinx-quickstart command that can build out some of the files to get started. Personally, I find it easier to start with an old project and copy/modify from there.

1.5.1 Building API Docs From Code

Sphinx can pull documentation stored in comments from Python files. If the Python code looks like this:

Listing 3: Documented Python Code Listing

```python
def my_function():
    """
    Sample function that prints Hi
    """
    print("Hi")


def my_addition_function(a: int, b: int):
    """
        Sample function that adds two integers.

        Our documentation starts off with a one-line sentence. We can follow it up
        with multiple lines that give a more complete description.

        :param int a: First number to add.
        :param int b: Second number to add.

        :return: The two numbers added together.
        :rtype: int
        :raises: None

        Then we can follow it up with an example:

        :Example:

            >>> result = my_addition_function(10, 15)
            >>> print(result)

        .. note::
            This is just a silly example. Don't really use this function to add
            numbers.

```

(continues on next page)

(continued from previous page)

```
    """
    return a + b
```

Then, in the restructured text file add code like the following:

```
.. automodule:: pypi_package_example
    :members:
    :undoc-members:
    :inherited-members:
    :show-inheritance:
```

And finally get output that looks like this: *API*

Note how it also links to the source of the code!

1.5.2 Read The Docs

Do you have an open-source project and don't want to spend a lot of time hosting a website and keeping everything up-to-date? ReadTheDocs will take any GitHub project and automatically build a website for you using Sphinx. They will inject some ads into it to help pay for it.

ReadTheDocs supports custom URLs. They also support webhooks that will auto-build the documentation every time you push a new version to GitHub.

1.5.3 Files and Directories for Documentation

- `/doc/` Put your documentation in this directory
- `/doc/index.rst` This is your main landing page
- `/doc/_static` This directory will be included in your main project. I use it for a custom css file.
- `/doc/images` It is a good idea to keep images separate from content.
- `/pypi_package_example/examples` If you want example code, I suggest putting it in a subdirectory to your project. Don't put it in the doc directory. This makes it easy to run your example with a command like:

```
python -m pypi_package_example/examples/my_example
```

1.6 Badges

Badges are a fun, visual way of showing information about your projects. Here are some badges for this project:

You can get the source to copy/paste into your own project from ShieldsIO. This website has TONS of badges to look through and see what fits your project.

1.7 Testing

1.7.1 PyTest

There are several testing frameworks that exist for helping with unit tests. One of the most popular for Python is PyTest. PyTest makes it easy to write and run unit tests.

Typically I create a directory called "tests" for all of the unit tests. I put the "tests" directory in my root folder, but if you want to run tests as part of the package once it has been installed, then you'll need to include it as a subdirectory in the folder with the source code.

Files that contain tests should start with `test_` and contain functions that start the same way:

`/tests/test_*.py`

An example test file:

Listing 4: test_my_addition_function.py

```python
import pypi_package_example


def test_my_addition_function():
    assert pypi_package_example.my_addition_function(5, 10) == 15
    assert pypi_package_example.my_addition_function(15, 10) == 25
    assert pypi_package_example.my_addition_function(-10, 10) == 0
```

If you are using PyCharm you can right-click on the tests folder and run the tests easily from within the IDE.

You can run PyTest from the command-line by just typing in `pytest` on the root folder. If that doesn't work:

- Make sure PyTest is listed in `requirements.txt` and installed.
- Create an empty file called `conftest.py` in the root of your project folder.

1.7.2 Code Coverage

If you'd like to make sure that your unit tests cover all (or most) of your code, you can add the pytest-cov package. Then you can run PyTest with the `--cov` parameter to see what percent of the project your tests cover:

`pytest --cov=pypi_package_example tests/`

You'll get putput like this:

```
(venv) S:\Webserver\pypi_package_example>pytest --cov=pypi_package_examp
le tests/
========================= test session starts =========================
platform win32 -- Python 3.7.4, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: S:\Webserver\pypi_package_example
plugins: cov-2.8.1
collected 2 items

tests\test_my_addition_function.py .                              [ 50%]
tests\test_my_function.py .                                       [100%]

----------- coverage: platform win32, python 3.7.4-final-0 -----------
Name                              Stmts   Miss  Cover
-----------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
pypi_package_example\__init__.py        4      0   100%


========================= 2 passed in 0.09s ==========================
```

This does not guarantee that your tests are good tests, just help you identify what parts of the code are at least run once as part of the tests.

If you want an even nicer display, the Coveralls website allows you to display and navigate your code coverage statistics. This is easy to add, by linking the Coveralls website to your GitHub account, turning on the project, and updating the `/.travis.yml` file to send over the data.

1.7.3 Pre-Commit

If you want to make sure that everything is in order before you commit, the Pre-commit module will allow you to run tests (andy anything else you'd like) whenever you try to commit with git. This helps encourage code quality.

1.8 Continuous Integration

There are many services that will automatically build your project on multiple platforms and run unit tests, code formatting tests, and code coverage tests.

While you can create your own in-house build machine, there are companies that already have it set up. Some of them include:

- Appveyor
- Jenkins
- TravisCI

1.8.1 Travis CI

In this example, we use TravisCI to do our builds. There is a YAML configuration file for TravisCI in the main file:

- .travis.yml

Here is the link so you can see the TravisCI for pypi_package_example build history on TravisCI.

Using Coveralls, you can see the code coverage of our tests:

- https://coveralls.io/github/pvcraven/pypi_package_example

You can add cool badges to your docs for these:

1.9 Getting Code Formatted Correctly

Code should follow a consistent standard to help readability. For Python this standard is defined in the style-guide called PEP-8.

If you use PyCharm, you can use there hints. These appear as a yellow underline around the offending code, and on the right margin. You can also scan an entire prject with the *Code... Inspect Code* menu option.

To scan for PEP-8 compliance on the command-line, you can use the flake8 module.

The Black will fix many issues for you automatically, rather than just telling you about them.

Using the pre-commit module along with flake8 and Black will make sure that code meets standards before allowing it to be committed.

flake8 can also be added as part of the Continuous Integration and cause a broken build if standards aren't met.

1.10 Versioning

1.11 Tips

You can install a package and use it from your development directory by `pip install -e .`

Get things set up with CookieCutter.

1.12 Example Directory Tree

- .github → *GitHub and GitHub Community*
  - ISSUE_TEMPLATE
    - bug_report.md → *Bug Report Template*
  - PULL_REQUEST_TEMPLATE
    - pull_request_template.md → *Pull Request Template*
- doc → *Package Documentation Files*
  - _static
    - css
  - build
    - html
  - conf.py
  - index.rst
  - make.bat

- pypi_package_example
  - __pypackages__ $\longrightarrow$ See PEP 582
  - __init__.py
- tests $\longrightarrow$ *Testing*
- venv
- .gitignore $\longrightarrow$ *Git*
- .pre-commit-config.yaml $\longrightarrow$ *Pre-Commit*
- .travis.yml $\longrightarrow$ *Travis CI*
- CODE_OF_CONDUCT.md $\longrightarrow$ *Code of Conduct*
- conftest.py $\longrightarrow$ *Testing*
- CONTRIBUTING.md $\longrightarrow$ *Contributing*
- license.rst $\longrightarrow$ *License*
- make.bat $\longrightarrow$ *Make File*
- MANIFEST.in $\longrightarrow$ Use this to specify what should be in a source distribution
- README.rst $\longrightarrow$ *Readme*
- requirements.txt $\longrightarrow$ *Requirements*
- setup.py $\longrightarrow$ *Setup File*

CHAPTER 2

Application Programming Interface

2.1 API

`pypi_package_example.`**`my_addition_function`**`(`*a: int*, *b: int*`)`

Sample function that adds two integers.

Our documentation starts off with a one-line sentence. We can follow it up with multiple lines that give a more complete description.

Parameters

- **a** (*int*) – First number to add.
- **b** (*int*) – Second number to add.

Returns The two numbers added together.

Return type int

Raises None

Then we can follow it up with an example:

Example

```
>>> result = my_addition_function(10, 15)
>>> print(result)
```

Note: This is just a silly example. Don’t really use this function to add numbers.

`pypi_package_example.`**`my_function`**`()`

Sample function that prints Hi

CHAPTER 3

Notes

The official Python Packaging documentation:

https://packaging.python.org/

The official sample project:

https://github.com/pypa/sampleproject

Python Module Index

Index