How to Package and Publish Your Python Codes

Junying (Alice) Fang

Check In


CHECK-IN
HOW TO PACKAGE AND PUBLISH YOUR PYTHON CODE
WEDNESDAY, JANUARY 15 AT 1:00PM
https://cglink.me/2gi/c19513971010152534
1 Open the My PrincetonU app.
2 Select a Hub
3 Click on QR Code scanner.
4 Scan this QR Code and you are checked-in!
Download on the App Store
GET IT ON Google Play

Download Workshop Codes

https://github.com/fjying/PythonPackageWorkshop

1. Open the Repo Link
2. Click the Green Code Button on the Top Right of the Page
3. Click Download Zip to Download the Repo
4. Zip the Downloaded Folder
5. Rename the Folder as PythonPackageWorkshop
6. Put the Folder on the appropriate place

Outline of Workshop

1. Virtual Environment of Packages
2. Build Python Package
3. Create an Automatic Test of Package Codes
4. Documentation Files for Package Publishing
5. Package Versioning
6. Publish Package
7. Create Package Citation

Virtual Environment of Packages

Virtual Environment

Suppose we just open the terminal and enter

pip install <package>,

the package would be installed on the base python environment by default.

What is the main concern with this?

Package Versions Conflict

For instance, when we use pip install pandas to install package pandas, the latest version of pandas would be installed. However, other packages may require the use of previous versions of pandas as some functions (such as DataFrame.append()) in the previous versions are no longer supported by the latest version.

Virtual Environment to Manage Different Versions of Packages

We could not install different versions of the same package in the same virtual environment.

The solution is to create **different virtual environments** so **different versions of the same package** could be installed **separately inside different environments.**

When we want to use the package of specific version, we could activate the environment related to that package version.

Virtual Environment for Software Development

Typically, for software development, each environment is created and managed for each project to prevent versions conflicts. The environment file could be shared, so other people could set up the exactly same environment to run project codes smoothly.

Specifically, for each Github Repo, we could create one virtual environment to manage packages specifically related to that Repo.

Use Python to Create Virtual Environment

mkdir ~/packageenvs

cd ~/packageenvs

python -m venv <env_name>

source <env_name>/bin/activate

deactivate

Use Conda to Create Virtual Environment

This requires the installation of Anaconda.

conda create -n <env_name>

conda activate <env_name>

conda deactivate

Set Up Virtual Environment of the Workshop

Conda:

```
conda create -n pythonpackageworkshop python=3.9

conda activate pythonpackageworkshop
```

Python:

```
mkdir ~/packageenvs

cd ~/packageenvs

python -m venv pythonpackageworkshop

source pythonpackageworkshop/bin/activate
```

How to Build a Python Package?

The Necessary Directory Structure for a Package

src directory: contain source codes

tests directory: tests codes

noxfile.py: main file to run test codes

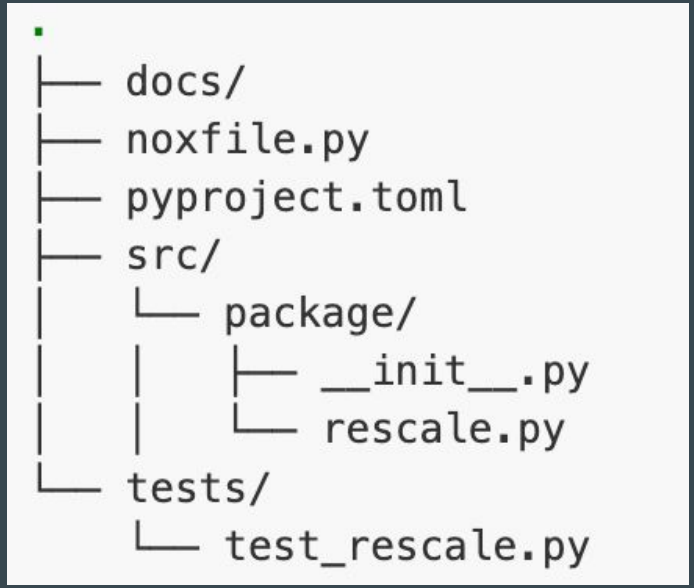
docs directory: package usage documentation

pyproject.toml: contains metadata to build

and install a package

__init__.py: Even if it is empty, it is necessary to maintain package and

subpackages structure.



Pyproject.toml

Contains Package Metadata for Building a Package and Publishing It

Two Categories of Metadata:

Informational (like Package Name, Author)

Functional (like Requirements to install all necessary python packages to run codes)

See toml file for the details

Build the First Python Package

1. Change Directory to the Repo Folder

cd <.../PythonPackageWorkshop>

2. Build and Install the Package named packageworkshop

pip install -e .

3. Run Test Codes below to See If Package Works

python

import numpy as np

from packageworkshop.rescale import rescale

rescale(np.linspace(0, 100, 5))

Automatic Test of Package Codes

Automatic Test

Test Function: tests/test_rescale.py

Test Session: noxfile.py

Terminal:

cd to the repo folder: <.../PythonPackageWorkshop>

nox

Package Files Documentation

README

A README is a plain text file that sits at the top level of the package and provides general information about the package.

A README should at least contain information below:

The name of the software package

A brief description of what the software does

Installation instructions

A brief usage example

The type of software license (with more information in a separate LICENSE file, described next)

(See README.md on codes repo and README on Github Page)

LICENSE

Before we use third-party software such as PyCharm, the software provider gives us the LICENSE to ensure that we could use the software appropriately.

Similarly, it is necessary to put LICENSE on the repo if the repo is public to ensure that end users use it legally based on the terms of usage in the license.

LICENSE is necessary to protect your copyright as a software developer.

LICENSE

Common Licenses Used by Research Software:

MIT License, BSD 3-Clause License, Apache License 2.0

Please do not write your own license!

Could create a license from Github Repo:

https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/adding-a-license-to-a-repository

CHANGELOG

In the CHANGELOG file, you should record major changes to the package made since the last released version. Then, when you decide to release a new version, you add a new section to the file above this list of changes.

There are 6 types of changes:

Added for new features,

Changed for changes in existing functionality,

Deprecated for soon-to-be removed features,

Removed for now-removed features,

Fixed for any bug fixes, and

Security in case of vulnerabilities.

Package Versioning

Three Types of Package Versioning

SemVer: Semantic Versioning

ZeroVer: Modified Semantic Versioning Starting with 0

CalVer: Calendar Based Versioning

SemVer: Semantic Versioning

SemVer: Three numbers in a Semantic Versioning; <major>.<minor>.<patch>

If the package changes in a breaking way, the major number must be incremented.

If the package has more features but the existing one still works fine, the minor number must be incremented.

If only bugs are fixed, the patch number must be incremented.

For instance:

Package version upgrades from 1.0.0 to 2.0.0: Users definitely need to check the latest version. Some new features are added. E.g. Fine-Tuning for ML model

Package version upgrades from 1.0.0 to 1.0.1: Users don't need to check the latest version. If something is broken on current use case, users may check the latest version to see if the use case could be fixed.

ZeroVer: Modified Semantic Versioning

Exactly the same as Semantic Versioning

Except that the lowest package version starts with 0.X.X.

The zero version (0.X.X) means that users could use this zero version stably, but there are huge parts of codes which would be rewritten and huge features would be added. This means that this package is still in a development phase.

CalVer: Calendar based versioning

This sets the version number based on the release date. There are several variations. Some projects literally place the date (two or four digit year followed by month then day) as a version number, and some project blend a little bit of SemVer in by making the second or third digit SemVer-like.

For instance:

22.04 version indicates that the package was released on April, 2022.

23.1.0 version indicates that the package was released on January, 2023. 0 indicates that this package version is a major version without any patches.

Semver versus Calver: Which one should we use?

For most packages, we should use Semver or ZeroVer.

If a package needs to talk to many external services regarding its version released date automatically, we should use Calver. For instance, several core Python packaging projects (like pip, packaging, and virtualenv) use CalVer.

Publish Package on TestPyPI or PyPI

Do you need to publish package to PyPI？

Not every package needs to go on PyPI. Users can pip install the package directly from codes on Github by following our steps before. Also, if the package is used for the front-end development such as building web application so web app instead of package codes would be used directly, such package does not need to be on PyPI.

Publish Package to TestPyPI

Before publishing any package officially to PyPI, we should always do a test drive by publishing it to TestPyPI. The steps of publishing packages are exactly the same for TestPyPI and PyPI.

Two Ways to Do It:

(1) Manual

(2) Github Action

Set Up TestPyPI Account and Make an API Token

1. Register account at https://test.pypi.org/

2. Verify your email address (check your account settings)

3. Make an API Token:

In your account settings, go to the API tokens section and select "Add API token"

3. Add API Token to Local Computer so we don't need to enter username and password whenever we want to publish a package to TestPyPI

vim ~/.pypirc

Inside the file:

[testpypi]

username = __token__

password = <api_token_generated_in_the_previous_step>

Manually Publish Package to TestPyPI

Note that the package name needs to be unique across TestPyPI

Need to change packagename in the pyproject.toml first if you want to publish a package on your own

1. Install All Necessary Packages for Publishing

python -m pip install build twine

2. Build Your Own Package

python -m build

3. Publish Package to Testpypi

twine upload -r testpypi dist/*

See Published Package and Install It from TestPyPI

See Package on TestPyPI Package

https://test.pypi.org/project/packageworkshop/

Install Package from TestPyPI

pip install -i https://test.pypi.org/simple/packageworkshop

Manually Publish a Package to PyPI

Exactly the same as publishing it to TestPyPI

Need to make sure your package is finalized before publishing it to PyPI

python -m build

twine upload dist/*

(instead of twine upload -r testpypi dist/*)

Publish New Version of Package to TestPyPI/PyPI

- Remove Previously Built Files on dist Folder
  - rm -r dist
- Change Version
  - Change version in the pyproject.toml if it is fixed
  - Change version in the version global variable if it is not fixed
- Rebuild Package
  - python -m build
- Upload Package to TestPyPI/PyPI
  - twine upload -r testpypi dist/*
  - twine upload -r dist/*
  - twine upload speakerchangedetect-0.0.1-py3-none-any.whl
  - Options: --verbose, --config-file /path/to/missing/file
- View Release History on TestPyPI/PyPI
  - https://test.pypi.org/project/packageworkshop/#history

Advanced: Use Github CI to Publish Python Packages

Use Github Action to Automate the Process of Publishing package:

Whenever we make a new release on Github, the corresponding release would be published to TestPyPI or PyPI automatically.

1. Need to fill out the form on the following link:

https://test.pypi.org/manage/account/publishing/

2. Create a YAML file for Github Actions to automate the workflow
PythonPackageWorkshop/.github/workflows/publishtopypi.yml

(See Continuous Integration/Delivery with Github Workflows Workshop)

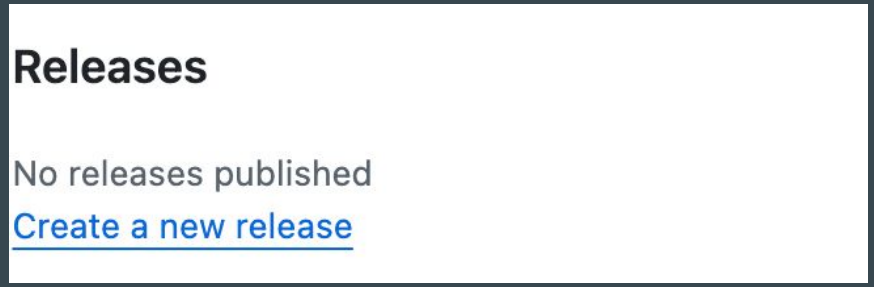
Release New Version on Github

Release New Version on Github

Publishing a package to PyPI is not always necessary. Github, instead of PyPI could also be a platform for sharing packages. Github also provides features to release multiple versions of the package and manage each version.

On the right sidebar of Github Repo, look for the **Releases** Section and click Create a new release.

(Set as the latest release Option)



Make Package Citable

Zenodo for a Software Citation

1. Log into Zenodo by using Sign In with Github:

https://zenodo.org/

2. Under Username on the top right, Click Github Section

3. Follow the instructions: Zenodo would automatically update the citation page when the new version is released on Github

4. Add a DOI badge to README

On Zenodo, click on the badge which opens up the markdown of badge.

Put the markdown at the top of README

Could go to the citation page by clicking on the Badge

ZENODO Citation Page of Software

https://zenodo.org/records/10436875

Citation of software in different styles is in the right corner.

Create CITATION.cff

CITATION.cff file on the Github Repo helps people to know how to cite your package.

Should not create CITATION.cff file manually

Generate CITATION.cff file using the below link:

https://citation-file-format.github.io/cff-initializer-javascript/#/

See CITATION.cff file on the repo for details

Push CITATION.cff file to Github

After pushing the CITATION.cff file to Github, people could easily cite the package by visiting the github repo: