Packaging and Distributing Python Projects

Maximilian Nöthe

Astroparticle Physics, TU Dortmund

ESCAPE
European Science Cluster of Astronomy &
Particle physics ESFRI research Infrastructures

Summer School – 2021-06-10

overview

Introduction

Using setuptools

Versions and Semantic Versioning

Choosing a License

Publishing Binary Wheels

A new Alternative: poetry

Conda Packages and conda-forge

Conclusions and Recommendations

Warning

Copying commands or code from PDF files is **dangerous**

Copy from the example files in the repository or type by hand.

Typing by hand is best for learning.

The Python Package Index

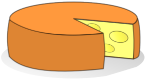
- Python packages are published on the Python Package Index (`https://pypi.org`)
- `pip install foo` will by default:
  1. Search for a package named `foo` on PyPI
  2. Download the best available distribution for your platform
  3. Install all dependencies of the package
  4. Install the package
- There is `https://test.pypi.org` for people to test their packaging code before publishing to “the real thing”.
- It is also possible to self-host a python package index

Source Distributions and Wheels

Source Distributions

- → `.zip` or `.tar.gz` archives of the project
- → Simplest solution to publish your package
- → If a package contains compiled components, these need to be built at installation time

Wheels



- → Standardized format for pre-built python packages
- → Simple for pure-python packages (no compiled components)
- → Platform-dependent wheels for packages with compiled components
  - → C-Extensions
  - → Cython-Code
  - → Wrappers for C or C++-Libraries
  - → …

Wheels

- → Platform dependent binary wheels must follow standards to be uploaded to PyPI
- → This is to ensure they run on many systems (not just on your computer)
- → Essentially:
  - → Compile using the oldest C-Standard Library a package wants to support
  - → Include all needed libraries in the wheel

More on how to actually build wheels for your own projects later.

Using setuptools

setuptools

- → `setuptools` is the most common solution for python packaging
- → Allows to declare package metadata, dependencies
- → Facilitates creation of files for distribution

Example Package Structure

```
eschool21_demo
├── eschool21_demo
│   ├── tests
│   │   ├── __init__.py
│   │   └── test_fibonacci.py
│   ├── fibonacci.py
│   └── __init__.py
├── LICENSE
├── pyproject.toml
├── README.md
├── setup.cfg
└── setup.py
```

Common convention: project directory equal or very similar to package name:

→ `numpy / numpy`

→ `PyTables / tables`

→ `python-dateutil / dateutil`

Example Package Structure

```
eschool21_demo
├── eschool21_demo
│   ├── tests
│   │   ├── __init__.py
│   │   └── test_fibonacci.py
│   ├── fibonacci.py
│   └── __init__.py
├── LICENSE
├── pyproject.toml
├── README.md
├── setup.cfg
└── setup.py
```

Files in the base directory for metadata / build configuration

README.md	Project description
LICENSE	Software license
pyproject.toml	Common configuration for python projects
setup.{py,cfg}	setuptools specific project files

pyproject.toml

- Defines *build-time* dependencies of a python package
- Uses the toml file format: `https://github.com/toml-lang/toml`
- Defined in PEP 517 and PEP 518
- Many other tools can also be configured through `pyproject.toml`, e.g. black, poetry, ….

Minimal pyproject.toml file for projects using setuptools

```toml
[build-system]
# required packages *at build time*
requires = ["setuptools", "wheel"]

# the function e.g. pip will call to build our project.
build-backend = "setuptools.build_meta"
```

setup.py and setup.cfg

→ All metadata concerning your package can be specified in a `setup.py` file

  → Using code for configuration is generally not a good idea

  → Projects can run arbitrary python code in `setup.py` to setup the project

⇒ For simple projects, only use the `setup.cfg`

→ Editable installs currently require a minimal `setup.py`:

```python
from setuptools import setup

# this is a workaround for an issue in pip that prevents editable installs
# with --user, see https://github.com/pypa/pip/issues/7953
import site, sys; site.ENABLE_USER_SITE = "--user" in sys.argv[1:]

setup()
```

setup.cfg

```
[metadata]
name = mypackage
version = 0.1.0
description = Example Package
license = MIT
# ... many more metadata options possible, see docs
long_description = file: README.md
long_description_content_type = text/markdown

classifiers =
  # see https://pypi.org/classifiers/ for more
  License :: OSI Approved :: MIT License

[options]
packages = find:  # automatically find python packages
python_requires = >=3.6
install_requires =
  astropy >= 4
```

Building the Project

- Install the `build` package (already available in the `eschool21` environment):

```
$ python -m pip install build
```

- Run the build module in the project directory

```
$ python -m build
```

- You will get both the sdist and the wheel in the `dist` directory:

```
$ ls -1 dist
eschool21_demo-0.1.0-py3-none-any.whl
eschool21_demo-0.1.0.tar.gz
```

Upload to (Test-)PyPI

- → Create an Account at (Test-)PyPI
- → Install `twine` (already available in the `eschool21` environment)

```
$ python -m pip install twine
```

- → Run the upload (here to test.pypi.org):

```
$ twine upload --repository testpypi dist/*
```

- → Go to your uploaded project and check everything is ok

Upload to (Test-)PyPI

→ Create an Account at (Test-)PyPI

→ Install `twine` (already available in the `eschool21` environment)

```
$ python -m pip install twine
```

→ Run the upload (here to test.pypi.org):

```
$ twine upload --repository testpypi dist/*
```

→ Go to your uploaded project and check everything is ok

For security reasons, PyPI does not allow replacing uploaded files. You have to upload a new *version*.

Entry Points

→ Console script entry points define scripts that get installed so they can be run from the command line

setup.cfg

```
[options.entry_points]
console_scripts =
  fibonacci = eschool21_demo.__main__:main
```

Including Data

→ To include non-code files into the source distribution and wheels you need to add

setup.cfg

```
[options]
include_package_data = True
```

→ And define these additional files in an additional file `MANIFEST.in`

Versions and Semantic Versioning

Versioning your Projects

→ PEP 440 prescribes a versioning scheme for all python projects:

```
[N!]N(.N)*[{a|b|rc}N][.postN][.devN]
```

N! Version epoch, extremely rare, needed only when switching the versioning scheme

N(.N)* Version identifier as arbitrarily many numbers separated by a dot

aN|bN|rcN Pre-releases (alpha, beta, release candidate) for testing

.postN Post releases, no changes to actual code, but e. g. better docs / fixed build system

.devN are development releases (N can be used e.g. to specify the number of commits since the last released)

→ By default, pip will not consider pre- and dev-releases

→ Versions are sortable

Version examples

Versions in sorted order

```
1.0.9
1.1.0.dev10
1.1.0a1
1.1.0a2
1.1.0b1
1.1.0rc1
1.1.0
1.1.0.post1
1.2.0
```

Semantic Versioning

- → See `https://semver.org`
- → SemVer uses a three part version like this:

`MAJOR.MINOR.PATCH`

- → Projects must increment:
  1. `MAJOR` version when you make incompatible API changes,
  2. `MINOR` version when you add functionality in a backwards compatible manner
  3. `PATCH` version when you make backwards compatible bug fixes
- → This makes depending on specific versions much easier

Caveats:
Many python projects do not strictly follow SemVer (e. g. numpy)
Many projects make breaking changes in MINOR updates until reaching 1.0.0

Specifying Versions of Dependencies

- → One of the most important things for packages is defining the compatible versions of the depedencies.
- → Projects can require smaller, larger, exactly equal and “compatible” versions
- → Projects can exclude versions
- → Versions may contain wildcards
- → Also defined in PEP 440

Depedency defintions

```
pandas                  # no requirement on the version
pandas >=1.0            # at least 1.0
pandas >=1.0,<2.0.0a0   # at least 1.0 but smaller than 2.0.0a0
pandas ==1.*            # Any 1.x version
pandas ~=1.1            # Any 1.x version >=1.1
pandas ~=1.1.2          # Any 1.1.x version >=1.1.2
pandas >=1.1,!=1.1.1    # Exclude 1.1.1 (had a bug?)
```

Extras

- You can use `extras` to define sets of optional dependencies
- Usefull especially for test and docs dependencies
- Consider providing an `all` extra to make it simple
- Extras can be requested in `[]`

```
$ pip install "package[extra1,extra2]"
```

setup.cfg

```
[options.extras_require]
tests =
    pytest
    pytest-cov
docs =
    sphinx
all =
    %(tests)s
    %(docs)s
```

Avoiding duplicated version definitions

- A common problem is that version information is needed at multiple locations
  - The git tag
  - The package version (e. g. in `setup.py` or `CMakeLists.txt`)
  - Accessible version in the code (e. g. `eschool21_demo.__version__`)
- Not having this information duplicated avoids errors
- Setuptools supports reading this from the code
- Tools like `setuptools_scm` can extract version information from git tags, but is a bit complicated to setup correctly

Defining the version in code

eschool21_demo/__init__.py

```python
from .fibonacci import fibonacci

__version__ = '0.1.0.post1'
__all__ = ['fibonacci']
```

setup.cfg

```ini
[metadata]
version = attr: eschool21_demo.__version__
```

This also works when `__init__.py` already imports dependencies, since setuptools is not actually importing the variable but parses the code.

Choosing a License

Software Licenses

- → Disclaimer: I am a Physicist, not a Lawyer
- → Software licenses have two main purposes
    1. Define what other people are allowed to do with your software
    2. Free the authors from liability / waving warranties
- → There are several "standard" free and open source licenses, endorsed by the *Open Source Initiative*: `https://opensource.org/licenses`
- → These licenses range from
    - → very short to very long
    - → very restrictive to very permissive
- → "Free as in freedom, not as in free beer."

The MIT License

MIT License

Copyright (c) 2021 ESCAPE

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.

GPL and LGPL

- → Allows redistribution, modification, running, …
- → Requires that source code is always published for binary distributions
- → Requires that derivative works are licensed under the same or a compatible license (copyleft)
- → If linking linking libraries constitutes a derivative work is contentious
- → The LGPL explicitly allows proprietary software to link LGPL licensed libraries

Scientific Software

- Opinion: The scientific method requires all code and data to be accessible
  - Reproducibility
  - Peer review
- This is most often not the case, but starting to get traction
  - Journals requiring release of software and data alongside publications
  - General trend towards more open development / open source scientific software
  - “Replication Crisis”

Publishing Binary Wheels

Example Project using cython and numpy

- Demo project in `school21/packaging/eschool21_cython_demo`
- Using cython and numpy’s cython API to build a C-extension
- We need Cython and numpy at build time
- We need to compile against the oldest supported numpy version

pyproject.toml

```toml
[build-system]
requires = ["setuptools", "wheel", "Cython", "oldest-supported-numpy"]
build-backend = "setuptools.build_meta"
```

Publishing Binary Wheels

Windows

With python >= 3.5, things got a lot easier

1. Install Visual Studio C/C++ Compilers and Windows SDK
2. Install all versions of python you want to support
3. Build the wheel for each version of python
4. Upload to PyPI

Publishing Binary Wheels

macOS

Decide the oldest supported macOS version (10.9 is most common)

1. Install compilers using `xcode install --select`
2. Install all versions of python you want to support (e.g. using pyenv or brew)
3. `export MACOSX_DEPLOYMENT_TARGET=10.9`
4. Build the wheel for each version of python
5. Upload to PyPI

Publishing Binary Wheels

Linux

- → The many different Linux distributions and versions make things a bit harder
- → Standardized wheels to make sure they run on “manylinux” distributions
- → Essentially you have to compile with the oldest glibc you want to support
- → e. g. `manylinux2014` is based on CentOS 7, glibc 2.17

The python packaging authority (PyPA) provides docker containers for each of these standards, which is the best way of building these, see `https://github.com/pypa/python-manylinux-demo`.

Publishing Binary Wheels – Including dependencies

- → Binary wheels are only allowed to link externally against basic system libraries defined in PEP 513 / PEP 599.
- → All other libraries must be included in the wheel
- → `auditwheel` (🐧) and `delocate` () take care of this
- → No off-the-shelve solution for ⊞
- → `cibuildwheel` offers CI build configurations for wheels for all platforms

Inside the manylinux docker container

```
$ auditwheel repair --plat manylinux2014_x86_64 <wheel> -w <outputdir>
```

A new Alternative: poetry

Poetry

poetry aims to provide a complete solution for dependency management and packaging

- + Configured completely in `pyproject.toml`
- + Automatic creation of virtual environments with all dependencies
- + Exact versions of all dependencies (including transitive) using a “lock file”
- + Building and publishing packages
- + Initial setup of new projects
- − No support for binary extensions / wheels yet

live demo

Conda Packages and conda-forge

Conda Packages

- → Conda packages for python packages should always start from a buildable package using the tools introduced before
- → Then, we only need to define build- and runtime depedencies as well as metadata in a yaml file `meta.yaml`

```
$ conda build <path to recipe directory>
$ anaconda upload <path to package>
```

- → `conda-forge` provides CI infrastructure to automatically build conda packages for open source projects

Conclusions and Recommendations

Conclusions and Recommendations

- → Always make sure your code is a valid python package and declares its dependencies
- → Publishing sdists / any-wheels to PyPI is free and easy
  → your code is just a `pip install` away
- → Choose a permissive FOSS License for scientific software (MIT or BSD 3-Clause)
- → When you expect your users to rely on conda, also publish conda packages
- → conda-forge is highly recommended, as it automatizes this process greatly
- → When using compiled python extensions, consider publishing wheels and/or conda packages
  → Much easier installation for your users, but may be a bit complicated to setup
- → Consider using poetry instead of setuptools for applications / libraries without compiled components
- → The python packaging landscape has improved greatly in the last couple of years, check carefully if guides you find are still up-to-date

Further Reading

PyPA User Guide	`https://packaging.python.org`
setuptools docs	`https://setuptools.readthedocs.io/`
poetry	`https://python-poetry.org/`
conda-forge	`https://conda-forge.org/`
conda build docs	`https://docs.conda.io/projects/conda-build`
cibuildwheel	`https://github.com/pypa/cibuildwheel`
auditwheel	`https://github.com/pypa/auditwheel`
delocate	`https://github.com/matthew-brett/delocate`
manylinux demo	`https://github.com/pypa/python-manylinux-demo`