> Note: This tutorial was created originally by Stephen Hudson and post at https://betterscientificsoftware.github.io/python-for-hpc/tutorials/python-pypi-packaging/. Modifications have been made here during my learning process.

1. Introduction

This is a quickstart guide to Python Packaging with a particular focus on the creation of a PyPI package, which will enable users to “pip install” the package. The document is broken down into sections so that readers may easily skips parts of the process they are already familiar with. All but the final section (Uploading to PyPI), can be undertaken as an exercise to understand Python packaging and test the process, without publishing a package on the formal PyPI distribution.

For a more detailed reference on package creation, see the official Python Packaging Authority (PyPA) website.

Note: PyPI should be pronounced “pie P I” to avoid confusion with pypy (a Python implementation).

2. What is pip?

pip is a package management system, specifically designed for installing Python packages from from the internet hosted Python Package Index (commonly known as PyPI). It is the most common way to install Python packages. E.g.

The package can now be imported in Python scripts. You may need to run as sudo if you have root privileges, or append `--user` to install under your home directory (often this will be under $HOME/.local).

Note: pip3 is used to install Python3 packages, however in some environments the command pip may point to pip3, just as python may point to Python3. You can use `which pip` to check this. For this document, examples will show the command simply as pip.

Tip: To download a specific version of a package:

```
pip install mpi4py==2.0.0
```

Tip: To find out what versions are available:

This is essentially trying to install a version that does not exist and causes pip to list available versions.

Tip: To see what version is currently installed:

Package information, including install location, can be obtained by running the Python interpreter:

```
$ python
Python 3.6.3 |Intel Corporation| (default, Oct 16 2017,
15:28:36) ....
>>> import mpi4py
>>> mpi4py.__path__
['/home/shudson/miniconda3/lib/python3.6/site-
packages/mpi4py']
>>> mpi4py.__version__
'2.0.0'
>>> mpi4py.__file__
'/home/shudson/miniconda3/lib/python3.6/site-
packages/mpi4py/__init__.py'
```

Installing pip is easy: https://pip.pypa.io/en/stable/installing

3. Creating a Python package

This article gives an overview of how to create an installable Python package.

> Note on Ambiguity: The term package can refer to an installable python *package* within a project (a directory containing an __init__.py file). It can also mean a *distribution package* which refers to the entire distributed part of the project (as in a source distribution - or “tarball”). Such a package may consist of multiple python package/sub-packages. In most cases the context should be sufficient to make the distinction.

A Python project will consist of a root directory with the name of the project. Somewhere inside this will be included a directory which will constitute the main installable package. Most often this has the same name as the project (This is not compulsory but makes things a bit simpler. Inside that package directory, alongside your python files, create a file called `__init__.py`. This file can be empty, and it denotes the directory as a python package. When you pip install, this directory will be installed and become importable.

A simple project may have this structure:

```
pyexample
├── LICENSE
├── pyexample
|   ├── __init__.py
|   ├── add.py
|   ├── subtract.py
|   └── times.py
|   └── divide.py
├── README.rst
└── setup.py
```

At the root directory, you will need a `setup.py` file, which will govern the installation of your package. The setuptools package is recommended for this (the in-built distutils is an older alternative).

```
pip install setuptools
```

The main requirement in `setup.py` is to call the setup routine, providing project information as keyword arguments. A lot of information can be provided, but the following is a minimalist example.

```
from setuptools import setup

setup(
    name='ppackexample',
    version='0.0.1',
    description='A example Python package',
    url='https://github.com/shuds13/ppackexample',
    author='Bin Yang',
    author_email='bin.yang@polymtl.ca',
    license='BSD 2-clause',
    packages=['ppackexample'],
    install_requires=['numpy>=0.5',
                      'scipy',
                      ],
    classifiers=[
        'Development Status :: 1 - Planning',
        'Intended Audience :: Science/Research',
        'License :: OSI Approved :: BSD License',
        'Operating System :: POSIX :: Linux',
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.4',
        'Programming Language :: Python :: 3.5',
        'Programming Language :: Python :: 3.7',
        'Programming Language :: Python :: 3.8',
        'Programming Language :: Python :: 3.9',
        'Programming Language :: Python :: 3.10',
    ],
)
```

> **Setup options**
>
> Further information on setup options can be found at: PyPA packaging instructions and yet more detailed and up to date information at: The setuptools command reference.

> Classifiers
>
> The classifiers are not functional, they are for documentation, and will be listed on the PYPI page, once uploaded. A complete list of classifiers is available at: PyPI classifiers list

Having created a setup.py, test the install with pip. In root dir:

```
pip install .
```

This is recommended in place of the default `python setup.py install` which uses easy_install. If you have an existing install, and want to ensure package and dependencies are updated use `--upgrade`

```
pip install --upgrade .
```

To uninstall (use package name):

```
pip uninstall ppackexample
```

> A reliable clean uninstall is one advantage of using setuptools over distutils.

It is worth noting that the version in your setup.py will not provide the package attribute __version__. A common place to provide this along with other meta-data for the package is inside the `__init__.py`. This is run whenever the module is imported. E.g: `__init__.py` may contain:

```
"""
ppackexample.

An example python library.
"""


__version__ = "0.0.1"
__author__ = 'Bin Yang'
__credits__ = 'Polytechnique Montreal & Wuhan University of 
Technology'
```

If you now pip install again and run the Python interpreter you should be able to access these variables:

```
$ python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC 
v.1914 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more 
information.
>>> import ppackexample
>>> ppackexample.__version__
'0.0.1'
>>> ppackexample.__author__
'Bin Yang'
>>> ppackexample.__credits__
'Polytechnique Montreal & Wuhan University of Technology'
```

This does create the problem of having two places holding the version, which must also match any release tags created (eg. in git). Various approaches exist for using a single version number. See https://packaging.python.org/guides/single-sourcing-package-version

If you wish to create sub-packages, these should ideally be directories inside the main package (Re-mapping from other locations is possible using the package_dir argument in setup but this can cause a problem with develop installs. The sub-packages also require an __init__.py in the directory.

4. Creating a source distribution

It is recommended that all Python projects provide a source distribution. PyPI has certain required meta-data that the setup.py should provide. **To quickly check if your project has this data use:**

```
python setup.py check
```

If nothing is reported your package is acceptable.

Create a source distribution. From your root directory:

```
python setup.py sdist
```

This creates a dist/ directory containing a compressed archive of the package (e.g. `<PACKAGE_NAME>-<VERSION>`.tar.gz in Linux). This file is your source distribution. If it does not automatically contain what you want, then you might consider using a MANIFEST file (see https://docs.python.org/distutils/sourcedist).

Note: A `<PACKAGE_NAME>`.egg-info directory will also be created in your root directory containing meta-data about your distribution. This can safely be deleted if it is not wanted (despite the extension, this is generated even though you have not built an egg format package).

5. Creating a wheel distribution

Optionally you may create a wheel distribution. This is a built distribution for the current platform. Wheels should be used in place of the older egg format. Bear in mind, any extensions will be built for the given platform and as such this must be consistent with any other project dependencies. Wheels will speed up installation if you have compiled code extensions as the build step is not required.

> If you do not have the wheel package you can pip install it.
>
> ```
> pip install wheel
> ```

There are different types of wheels. However, if your project is pure python and python2/3 compatible create a universal wheel:

```
python setup.py bdist_wheel --universal
```

If it is not python2/3 compatible or contains compiled extensions just use:

```
python setup.py bdist_wheel
```

The installable wheel will be created under the dist/ directory. A build directory will also be created with the built code. Further details for building wheels can be found here: https://packaging.python.org/tutorials/distributing-packages

6. Testing and Publishing package on PyPI

Distributing the package on PyPI will enable anyone on-line to pip install the package.

First you must set up an account on PyPI. If you are going to test your package on the PyPI test site you will need to set up an account there also. This is easy.

Create an account on PYPI: Go to: https://pypi.python.org and select Register. Follow instructions.

Create an account on testpypi: Go to: https://testpypi.python.org and select Register. Follow instructions.

You will also need a version number. Semantic versioning is recommended (see https://semver.org for details). The standard starting version for a project in development is 0.1.0.

The best approach to uploading to PyPI is to use twine.

```
pip install twine
```

IMPORTANT: First you can test your upload using the PyPI test site. It is highly recommended that you do this and test installing your package as below. NOTE: Once you upload a package to PYPI it is possible to remove it, but you cannot upload another package with the same version number – this breaks the version contract. It is therefore, especially prudent to test with testpypi first. Note that anything you put on testpypi should be considered disposable as the site regularly prunes content.

6.1 Uploading to testpypi

This section shows how to upload a source distribution of your package. Further documentation at: https://packaging.python.org/guides/using-testpypi.

> Note: This link includes the option of using a pypirc file to abbreviate some of the command lines below.

A source distribution provides everything needed to build/install the package on any supported platform. Test suites, documentation and supporting data can also be included.

You can now upload your package to testpypi as follows. Assuming your source distribution under *dist/* is called *ppackexample-0.0.1.tar.gz*:

```
twine upload --repository-url https://test.pypi.org/legacy/
dist/ppackexample-0.0.1.tar.gz
```

Now you need to provide your user name and password to complete the package uploading process. Alternatively, the following line will upload all your generated distrbutions under the dist/ directory. This may be used if you create wheels (see below) in addition to a source distribution.

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

You will be requested to give your username and password for your testpypi account also.

> Option: You have the option to digitally sign your package when you upload. You will need a gpg key set up to do this. It should be noted, however, that pip does not currently check gpg signatures when installing - this has to be done manually.
>
> To digitally sign using your gpg key (e.g. for package pyexample at version 0.1.0):
>
> ```
> gpg --detach-sign -a dist/pyexample-0.1.0.tar.gz
> ```
>
> A file `ppackexample-0.0.1.tar.gz.asc` will be created. Now upload:
>
> ```
> twine upload --repository-url
> https://test.pypi.org/legacy/ >dist/ppackexample-
> 0.0.1.tar.gz ppackexample-0.0.1.tar.gz.asc
> ```
>
> Note: `--detach-sign` means you are writing the signature into a separate file *.asc

The package should now be uploaded to: https://testpypi.python.org/pypi. Note how the info/classifiers you supplied in setup.py are shown on the page. You can now test pip install from the command line. E.g. To install package pyexample into your user install space:

```
pip install --index-url https://test.pypi.org/simple/
ppackexample --user
```

6.2 Uploading to PyPI

Once you are happy with the repository in testpyi, uploading to PYPI will be the same command line process, but without having to specify the url arguments. For example, the steps above would simply become:

- To upload all distributions created under dist/

  ```
  twine upload dist/*
  ```

- To upload the source distribution with a gpg signature:

  ```
  twine upload dist/ppackexample-0.0.1.tar.gz ppackexample-
  0.0.1.tar.gz.asc
  ```

You package should now be uploaded to: https://pypi.python.org/pypi

The package should pip install. E.g:

```
pip install ppackexample --user
```

> It is also recommended that you use virtual environments to test installing dependencies from scratch and for trying out different python versions. Check required flags to ensure your environment is isolated. E.g. For Virtualenv use the flag `--no-site-packages`. For Conda, set the environment variable `export PYTHONNOUSERSITE=1` before activating you environment. Packages that are explicitly linked through PYTHONPATH will still be found however.

6.3 Downloading tarball without install

To test downloading a source distribution (no install) with dependencies:

Or just the package without dependencies:

```
pip download --no-deps ppackexample
```

Downloading the source distribution is a good way to check that it includes what you want by default. If not, then consider adding a MANIFEST file, which instructs setuptools what to include in the source distribution.

If this was helpful, please leave a star on this project github page.