Packaging in Python: Tools and Formats

Martin Thoma ⁝ 19-25 minutes ⁝ 11/7/2020

---

16 solutions to 9 problems — which ones do you know?

Created by the author

A virtual environment is an isolated Python environment. It has it's own installed site-packages which can be different from the systems site-packages. Don't worry, we will go into more detail later.

After reading this article, you will understand what the following tools are and which problems they solve: pip, pyenv, venv, virtualenv, pipx, pipenv, pip-tools, setup.py, requirements.txt, requirementst.in, Pipfile, Pipfile.lock, twine, poetry, flint, and hatch.

Package Types

For this article, you need to distinguish two types of (packaged) code:

- Libraries are imported by other libraries or applications. Libraries do not run on their own; they are always run by an application. Examples for libraries in Python are Numpy, SciPy, Pandas, Flask, Django, click,
- Applications are executed. Examples for applications in Python are awscli, Jupyter (the notebooks), any website created with Flask or Django.

You can further distinguish those, e.g. libraries and frameworks. Or command line applications, applications with graphical user interfaces, services, and many more. But for this article, we only need to distinguish between libraries and applications.

Please note that some applications also contain code that can be imported or some libraries have a part of the functionality shipped as an application. In those cases, you can either use them as a library (including their code in your project) or as an application (just executing them). You are in command.

Problem 1: Different Python Version Required

We have Python 3.6 installed, but the application requires Python 3.8. We cannot upgrade our systems Python version, e.g. because we're missing administrator privileges or because other things would break.

Solution: pyenv

Pyenv allows you to install any Python version you want. You can also easily switch between Python environments with pyenv :

```
$ python --version
Python 3.6.0$ pyenv global 3.8.6$ python --version
Python 3.8.6$ pip --version
pip 20.2.1 from /home/math/.pyenv/versions/3.8.6/lib/python3.8/site-
packages/pip (python 3.8)
```

For more information, read my article A Beginner's Guide to Python Development. For detailed installation instructions, go directly to the official pyenv website.

Problem 2: Package and Distribution building

You typically don't only use bare Python. As developers, we stand on the shoulders of giants — the whole ecosystem of freely available software. In the beginning of Python, people just copied files. A Python file, when imported, is also called a module. If we have multiple Python files in one folder with an __init__.py , they can import each other. This folder is then called a package. Packages can contain other packages — subfolders which also have an __init__.py and are then called sub-packages.

Copying files and folders is inconvenient. If the author of that code makes an update, I might need to update dozens of files. I need to know that there is an update in the first place. I might need to install hundreds of dependencies as well. Doing that by copy-and-paste would be hell.

We need a more convenient way to distribute the packages.

Solution: Source Distributions

A packaging system needs three core components:

- Package Format: The simplest format in Python is called a source distribution. It is essentially a ZIP file that has a certain structure. One essential part of this file is the possibility to specify dependencies of the package. It should also contain other metadata, such as the name of the package, the author, and license information.
- Package manager: A program that installs packages. pip installs packages in Python.
- Software repository: A central place where package managers can look for packages. In the Python ecosystem, pypi.org is THE public one. I'm not even aware of other public ones. You can create private ones, of course.

As mentioned, we need a way to specify metadata and dependencies. This is done with the setup.py file. It typically looks like this:

```
from setuptools import setupsetup(
    name="my_awesome_package",
    version="0.1.0",
```

```
    install_requires=["requests", "click"]
)
```

There are many more version specifiers you can use, for example:

```
numpy>3.0.0  # 3.0.1 is acceptable, but not 3.0.0
numpy~=3.1   # 3.1 or later, but not version 4.0 or later.
numpy~=3.1.2 # 3.1.2 or later, but not version 3.2.0 or later.
```

In order to create the source distribution, we run

```
$ python setup.py sdist
```

I don't like the setup.py file so much, because it is code. For metadata, I prefer to use a configuration file. Setuptools allows to use a setup.cfg file. You still need a setup.py, but it can be reduced to:

```
from setuptools import setupsetup()
```

And then you have the setup.cfg file as follows. There is documentation about the setup.cfg format.

```
[metadata]
name = my_awesome_packageauthor = Martin Thoma
author_email = info@martin-thoma.de
maintainer = Martin Thoma
maintainer_email = info@martin-thoma.de# keep in sync with
my_awesome_package/__init__.py
version = 0.23.1description = Martins Python Utilities
long_description = file: README.md
long_description_content_type = text/markdown
keywords = utility,platforms = Linuxurl =
https://github.com/MartinThoma/mpu
download_url = https://github.com/MartinThoma/mpulicense = MIT#
https://pypi.org/pypi?%3Aaction=list_classifiers
classifiers =
    Development Status :: 3 - Alpha
    Environment :: Console
    Intended Audience :: Developers
    Intended Audience :: Information Technology
    License :: OSI Approved :: MIT License
    Natural Language :: English
    Operating System :: OS Independent
    Programming Language :: Python :: 3.7
    Programming Language :: Python :: 3.8
    Programming Language :: Python :: 3.9
    Topic :: Software Development
    Topic :: Utilities[options]
packages = find:
```

```
python_requires = >=3.7
install_requires =
    requests
    click[tool:pytest]
addopts = --doctest-modules --ignore=docs/ --durations=3 --timeout=30
doctest_encoding = utf-8[pydocstyle]
match_dir = mpu
ignore = D105, D413, D107, D416, D212, D203, D417[flake8]
max-complexity=10
max_line_length = 88
exclude = tests/*,.tox/*,.nox/*,docs/*
ignore = H301,H306,H404,H405,W503,D105,D413,D103[mutmut]
backup = False
runner = python -m pytest
tests_dir = tests/[mypy]
ignore_missing_imports = True
```

Problem 3: Secure Uploads

You want to upload packages securely to PyPI. You need to authenticate and you want to be certain that nobody tampers with your package.

Solution: twine

Install twine via pip install twine and you can upload your distribution files:

```
twine upload dist/*
```

Problem 4: Dependency Conflicts

You want to install youtube-downloader which needs the library requests in version 1.2.3 and vimeo-downloader which needs the library requests in version 3.2.1 . Hence the library requests is a dependency of both applications. Both applications need to be executed with Python 3.8. That is a problem as both applications store requests in the same site-packages directory. Once you install one version, the other one is gone. You need two different environments to run those two applications.

A Python environment is the python executable, pip, and the set of installed packages. Different environments are isolated from each other and thus don’t influence each other.

We solve this dependency conflict by creating a virtual environment. We call it virtual because they actually share the Python executable and other things like the shells' environment variables.

Solution: venv

Python has the venv module which happens to be executable as well. You can create and use a fresh virtual environment like this:

```
$ python -m venv my-fresh-venv
$ source my-fresh-venv/bin/activate(my-fresh-venv)$ pip --version
pip 20.1.1 from /home/moose/my-fresh-venv/lib/python3.8/site-
packages/pip (python 3.8)
```

The environment is called “fresh” because there is nothing in it. Everything you install after source-ing the activate script will be installed in this local directory. This means when you install youtube-downloader in one such virtual environment and vimeo-downloader in another, you can have both. You can go out of a virtual environment by executing deactivate .

If you want more details, I recommend to read Python Virtual Environments: A Primer.

Problem 5: Inconvenience

You would still need to switch between the virtual environments all the time which is inconvenient.

Solution: pipx

pipx automatically installs packages into their own virtual environment. It also automatically executes the applications within that environment 😍

Note: This only makes sense for applications! You need libraries within the same environment as your application. So don’t ever install libraries with pipx. Install applications (and indirectly the libraries) with pipx.

Problem 6: Changing third-party code

As an application developer, I want to be certain that my application keeps working. I want to be independent of potential breaking changes of third party software I use.

For example, think about the youtube-downloader which needed requests in version 1.2.3. At some point, probably during development, that version of requests was likely the latest version. Then the development of the youtube-downloader was stopped, but requests kept changing.

Solution: Dependency pinning

Give the exact version you want to install:

```
numpy==3.2.1
scipy==1.2.3
pandas==4.5.6
```

However, this has a problem of its own if you do it in setup.py . You will force this version upon other packages in the same environment. Python is pretty messy here: Once another package installs one of your dependencies in another version in the same

environment, it's simply overwritten. Your dependencies might still work, but you don't get the expected version.

For applications, you can pin the dependencies like this in the setup.py and tell your users to use pipx to install them. This way you and your users can be happy 💕

For libraries, you cannot do this. By definition, libraries are included by other code. Code that potentially includes a lot of libraries. If all of them pinned their dependencies, it would be very likely to get a dependency conflict. This makes library development hard if the developed library itself has several dependencies.

It's common practice to NOT pin dependencies in the setup.py file, but instead create a flat text file with pinned dependencies. PEP 440 defined the format or requirements files in 2013. It's usually called requirements.txt or requirements-dev.txt and typically looks like this:

```
numpy==3.2.1
scipy==1.2.3
pandas==4.5.6
```

You can also specify locations where the packages can be downloaded (e.g. not only the name but a git repository) according to PEP 440.

Packages within a requirements.txt (including their dependencies) can be installed with

```
$ pip install -r requirements.txt
```

Problem 7: Changing Transitive Dependencies

Imagine you write code which depends on the packages foo and bar . Those two packages might themselves have dependencies as well. Those dependencies are called transitive dependencies of your code. They are indirect dependencies. The reason why you need to care is the following.

Assume there are multiple versions of foo and bar published. foo and bar happened to both have exactly one dependency: fizz

Here is the situation:

```
foo 1.0.0 requires fizz==1.0.0
foo 1.2.0 requires fizz>=1.5.0, fizz<2.0.0
foo 2.0.0 requires fizz>=1.5.0, fizz<3.0.0bar 1.0.0 requires fizz>2.0.0
bar 1.0.1 requires fizz==3.0.0fizz 1.0.0 is available
fizz 1.2.0 is available
fizz 1.5.0 is available
fizz 2.0.0 is available
fizz 2.0.1 is available
fizz 3.0.0 is available
```

You might be tempted to just say “I need foo==2.0.0 and bar==1.0.0 . There are two problems:

1. Dependency satisfaction can be hard: The client needs to figure out that those two requirements can (only) be satisfied by fizz==2.0.0 orfizz==2.0.1 . This can be time-consuming as Python source distributions are not well designed and do not expose this information well (example discussion). The dependency resolver actually needs to download the package to find the dependencies.
2. Breaking transitive change: The packages foo and bar could not state their dependencies. You install them and things work, because you happen to have foo==2.0.0 , bar==1.0.0 , fizz==2.0.1 . But after a while, fizz==3.0.0 is released. Without telling pip what to install, it will install the latest version of fizz . Nobody tested that before as it didn’t exist. Your user is the first one and it breaks for them 😨

Solution: Pinning Transitive Dependencies

You need to figure out the transitive dependencies as well and tell pip exactly what to install. To do so, I start either with a setup.py or a requirements.in file. The requirements.in file contains what I know must be fulfilled — it’s pretty similar to the setup.py file. In contrast to the setup.py file it is a flat text file.

Then I use pip-compile from pip-tools to find the transitive dependencies. It will generate a requirements.txt file which looks like this:

```
#
# This file is autogenerated by pip-compile
# To update, run:
#
#    pip-compile setup.py
#
foo==2.0.0   # via setup.py
bar==1.0.0   # via setup.py
fizz==2.0.1  # via foo, bar
```

Typically, I have the following:

- setup.py: Defining abstract dependencies and known minimum versions / maximum versions.
- requirements.txt: One version combination that I know works on my machine. For web services where I control the installation, this is also used to install the dependencies via pip install -r requirements.txt
- requirements-dev.in: Development tools I use. Things like pytest, flake8, flake8 plugins, mypy, black … see my static code analysis post.
- requirements-dev.txt: The exact version of the tools I use + their transitive dependencies. Those are also installed in the CI pipeline. For applications, I also include the requirements.txt file in here. Please note that I create a combined requirements-dev.txt which includes the requirements.txt . If I would install the

> requirements.txt before the requirements-dev.txt, it could change the version. That would mean I would not test against exactly the same package versions. If I would install the requirements.txt after the requirements-dev.txt , I could break something for the dev tools. Hence I create one combined file via
> pip-compile --output-file requirements-dev.txt requirements.txt

You can also add --generate-hashes if you want to be certain it's exactly the same.

Problem 8: Non-Python Code

Packages like cryptography have code written in C. If you install the source distribution of cryptography, you need to be able to compile that code. You might not have a compile like gcc installed and compiling takes quite a bit of time.

Solution: Built Distributions (Wheels)

Package creators can also upload built distributions, e.g. as wheels files. This prevents you from having to compile stuff yourself. It is done like this:

```
$ pip install wheels
$ python setup.py bdist_wheel
```

For example, NumPy does this:

The screenshot of pypi.org was taken by the author.

Problem 9: Specification of build-system

The Python ecosystem is very strongly attached to setuptools. No matter how good setuptools are, there will always be features people are missing. But we couldn't change the build system for quite a while.

Solution: pyproject.toml

PEP 517 and PEP 518 specified thepyproject.toml file format. It looks like this:

```
[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

Yes, it's not much. It tells pip what is necessary to build your package. But it was a good step towards more flexibility.

Other tools, like poetry and black, used this file for their configuration to the pyproject.toml , similar as flake8 , pytest , pylint and many more allow you to add configuration to the setup.cfg .

Photo by Giorgio Trovato on Unsplash

Honorable mentions

The tools in this section are relatively wide-spread, but as of today, they don't really solve any issue that one of the tools from above doesn't solve. They might be more convenient to use than others.

virtualenv and virtualenvwrapper

The 3rd party tool virtualenv existed before the core module venv. They are not completely identical, but for me, venv was always good enough. I'm happy if somebody can show me a problem to which virtualenv (and not venv) is the solution 🙂

virtualenvwrapper extends virtualenv.

pipenv

Pipenv is a tool for dependency management and packaging. It introduces two new files:

- Pipfile: A TOML file. Its content is similar in thought to the one of requirements.in : Abstract dependencies.
- Pipfile.lock: A TOML file. Its content is similar in thought to the one of requirements.txt : Pinned concrete dependencies, including transitive dependencies.

Essentially, it wraps venv.

poetry

Poetry is a tool for dependency management and packaging. It combines a lot of tools, but it's core functionality is identical to pipenv. The main difference is that it uses pyproject.toml and poetry.lock instead of Pipfile and Pipfile.lock . A detailed comparison between poetry and pipenv was written by Frost Ming.

The projects poetry wraps or replaces are:

- Scaffolding: poetry new project-name vs cookie-cutter
- Building Distributions: poetry build vs python setup.py build sdist_build
- Dependency Management: poetry add foobar vs manually editing the setup.py / requirements.txt file. Poetry will then create a virtual environment, a poetry.lock file which is identical in concept to the Pipfile.lock and update the pyproject.toml . You can see an example of that below. They use their own dependency section which will not be compatible with anything else. I hope they move to PEP 631 (see issue for updates).
- Upload to PyPI: poetry publish vs twine upload dist/*
- Bump version: poetry version minor vs manually editing setup.py / setup.cfg or using bumpversion . ⚠️ Although poetry generates an __init__.py in the scaffolding which contains a version, poetry version does not change that!

It goes away from the de-facto standard setup.py / setup.cfg for specifying dependencies. Instead, poetry expects dependencies to be within it's configuration:

```
[tool.poetry]
name = "mpu"
version = "0.1.0"
description = ""
authors = ["Martin Thoma <info@martin-thoma.de>"]
license = "MIT"[tool.poetry.dependencies]
python = "^3.8"
awscli = "^1.18.172"
pydantic = "^1.7.2"
click = "^7.1.2"[tool.poetry.dev-dependencies]
```

I hope that they will also implement PEP 621 and PEP 631 which gives metadata and dependencies an official place under the [project] section. Let's see, maybe they change that.

Some people like to have one tool which does everything. I rather go with the Unix philosophy:

Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".

As poetry combines a lot of tools, it's also important what it doesn't do:

- Package management: You still need pip. And pip supports pyproject.toml.
- Scaffolding: Cookiecutter has a lot of templates. I created two myself: one for typical Python projects and one for Flake8 plugins.
- Setup.py: You might not need to create one yourself, but poetry creates a setup.py file for you. Just look at the distribution file.

I should also point out that poetry has a super nice CLI and a visually pleasing website.

hatch

Hatch also aims to replace quite a lot of tools:

- Scaffolding: hatch new project-name vs cookie-cutter
- Bump version: hatch grow minor vs manually editing setup.py / setup.cfg or using bumpversion
- Running pytest: hatch test vs pytest
- Create a virtual environment: hatch env my-venv vs python -m venv my-venv
- Installing packages: hatch install package vs pip install package

I had a couple of errors when I tried hatch.

Flit

Flit is a way to put Python packages and modules on PyPI. It is a 3rd party replacement for setuptools. In that sense, it's similar to setuptools + twine or a part of poetry.

Conda

Conda is the package manager of Anaconda. It is way more powerful than pip and can build/install code of arbitrary languages. With the pyproject.toml , I wonder if conda will be necessary in future 🧐

Red Herrings

- easy_install : That is the oldest way to install stuff in Python. It is similar to pip , but you cannot (easily) uninstall things that were installed with easy_install
- distutils : Although it's core Python, it's not used anymore. setuptools is more powerful and installed everywhere.
- distribute : I'm not sure if that ever was a thing?
- pyvenv : Deprecated in favor of venv .

Summary

- pip is Pythons package manager. It goes to the Python package index PyPI.org to install your packages and their dependencies.
- Abstract dependencies can be denoted with setup.py, requirements.in, Pipfile, or pyproject.toml. You only need one.
- Concrete dependencies can be denoted with requirements.txt, Pipfile.lock, or poetry.lock. You only need one.
- Building packages is done with setuptools or with poetry.
- Uploading packages is done with twine or poetry.
- Virtual environments are created with venv or with poetry / pipenv / hatch / conda
- pipx is cool if you want to install applications. Don't use it for libraries.