PEP 517
Python Packaging's New World Order

BY CHARLIE LI

One day on IRC, bofh@ ran into a problem whilst trying to update a Python port: the source no longer included a setup.py file. Without such a file, the Python framework within the Ports framework just did not work, there was no path forward. Intrigued, I started doing some light digging on the matter, and very quickly found an entire new packaging and distribution *standard* that we would need to support sooner or later. A growing collection of packages were leading or following suit, and the eventual Python 3.12 release would exacerbate this issue even further. I emphasise *standard* because setup.py and previous iterations of third-party/add-on Python software distribution were never standardised or architected at all…

Enter PEP 517, an actual design and architecture of Python package building and distribution, using the wheel package format first standardised in PEP 427. Immediately upon skimming all relevant Python Enhancement Proposals (PEPs), I exclaimed, this is way better, let's figure out how to implement this in the Ports framework.

A Brief History of Python Packaging Generally

Mostly adapted from "Why you shouldn't invoke setup.py directly"

Many perceptions of "brief" exist, so this may seem anything but. This history is unfortunately long and convoluted, with a load of nuance left out, so this is as "brief" as it can get.

Prior to Python 2.0, no organised way of distributing Python code existed, not like a C project's configure-build-install workflow. Python 2.0 introduced distutils, a new module within the standard library/distribution to provide something akin to the more common make(1) targets dealing with configure-build-install. This made it relatively straightforward to integrate into distro systems like our Ports framework to create operating system-level packages.

Unfortunately, no good way to specify and enforce dependencies could be had with only distutils. Unlike projects that configured and compiled code, where missing or incorrect dependencies would result in errors at any stage, no practical enforcement mechanism other than running the code itself existed for the interpreted Python (CPython has a bytecode compiler, the output of which is actually executed, but that's an entirely different topic with its own details and pitfalls). Distro systems like our Ports framework handle the dependency part of the equation, but most people outside of that context were not going to read READMEs or otherwise figure out dependencies to install Python code in their local environments.

Enter setuptools, intended as a drop-in enhanced replacement to distutils that provides, amongst other features, dependency management. Worked great for most packages that have setuptools at the top of the build chain. However, certain packages import dependencies before setuptools can do its thing. Different targets do not necessarily need the same dependency set. Some packages even specified exact setuptools versions. Worst of all, everything played in the same (host) environment, and setuptools itself cannot properly create the correct environment to execute properly.

For distro systems like our Ports framework, these shortcomings were less of an issue. We have ways to automatically manage dependencies and isolate environments to provide the correct environment for setuptools to do its thing, especially with poudriere. Not quite the case in developer scenarios, especially before Python virtual environments came along.

Additionally, the package formats defined by distutils/setuptools were inflexible, hard to maintain and hindered innovation with regards to the act of building and installing Python packages. The wheel standard was developed as a dedicated package format independent of any build and install scheme. This is much like our own pkg(8) or other operating system-level package manager package formats. Great, except distutils/setuptools themselves relied on yet another external Python package for this functionality, aptly named wheel, rather than integrating it in the first place. Can anyone smell circular dependency…

In the intervening years, different projects sprung up to experiment with non-distutils/setuptools build systems, particularly when setuptools grew as bloated as necessary but those projects were aiming for simplicity à la old-school Unix philosophy. But because distutils/setuptools was still the de facto interface for building and installing, none of these new projects could be used in production to do what they intended. With the package format defined to be independent of any build and install scheme, enter PEP 517, a minimal interface for generating wheels that build systems implement, allowing for choice in this area.

USE_PYTHON=distutils

Consider a Python package sample with the following source layout:

```
sample/
├── setup.py
├── …
└── src/
  └── sample/
    ├── __init__.py
    └── …
```

The port would look something like this:

```
PORTNAME=          sample
DISTVERSION=       1.2.3
CATEGORIES=        devel python
MASTER_SITES=      PYPI
PKGNAMEPREFIX=     ${PYTHON_PKGNAMEPREFIX}

MAINTAINER=        freebsd@example.org
COMMENT=           Python sample module

RUN_DEPENDS=       ${PYTHON_PKGNAMEPREFIX}six>0:devel/py-six@${PY_FLAVOR}

USES=              python
USE_PYTHON=        autoplist distutils

.include <bsd.port.mk>
```

With a properly equipped, normal setup.py, the ports framework executes setup.py's configure, build, install targets for each port target respectively. The Python package does not specify any build dependencies other than the implicit distutils/setuptools providing all the structure. A runtime dependency is specified, which distutils/setuptools checks only on install. Installation metadata is generated, which includes a list of installed artefacts that we use, with minor modifications, for our packing list.

This follows the procedure traditionally used for C/C++ projects with Makefiles. Artefacts are installed into a stage directory hierarchy which is then packaged up.

USE_PYTHON=pep517

Consider an updated Python package sample with the following source layout:

```
sample/
├── pyproject.toml
├── …
└── src/
    └── sample/
        ├── __init__.py
        └── …
```

The port would look something like this:

```
PORTNAME=          sample
DISTVERSION=       1.2.4
CATEGORIES=        devel python
MASTER_SITES=      PYPI
PKGNAMEPREFIX=     ${PYTHON_PKGNAMEPREFIX}

MAINTAINER=        freebsd@example.org
COMMENT=           Python sample module

BUILD_DEPENDS=     ${PYTHON_PKGNAMEPREFIX}flit-core>0:devel/py-flit-core@${PY_FLAVOR}
RUN_DEPENDS=       ${PYTHON_PKGNAMEPREFIX}six>0:devel/py-six@${PY_FLAVOR}

USES=              python
USE_PYTHON=        autoplist pep517

.include <bsd.port.mk>
```

At first glance, this port looks almost identical. When designing the porting workflow, careful consideration was taken to ensure as seamless of conversion as possible as Python packages update and adopt PEP 517.

With this method, setuptools is no longer the centre of attention. Instead, the implicit build dependencies are two separate Python package ports, a *build frontend* and *integration frontend*. The build frontend parses build-specific metadata in pyproject.toml to determine the *build backend*, make sure it exists in the environment, and executes it to build the wheel. In this example, the build backend is flit-core, and it is specified as a regular build dependency. If the build succeeds, a wheel file is generated with a strictly formatted file name. The integration frontend references the wheel file in that strict formatting, checks runtime dependencies and installs into staging, metadata included. Our packing list comes from this metadata similar to before.

A notable omission compared to the other method is a separate configure stage, which is integrated into the build stage. This allows Python packages to choose which build backend is most appropriate for their project, whether something available in PyPI or something custom within their source tree.

Caveats and Future Considerations

In the intervening period since PEP 517 support landed in the Ports framework, a number of people have commented on some perceived inflexibility in our implementation. More likely than not, the inflexibility is intentional, based on adherence to Python standards and security/integrity considerations amongst others.

One focal point has involved the wheel file name called during the stage process. We could pass an entire wildcard wheel file name into the PEP 517 integration frontend and call it a day, but such squanders a golden opportunity to improve metadata congruency between the Python side and the port. Some ports' names or versions do not match their Python package metadata counterparts. To add insult to injury, PyPI has had a history of typosquatted packages leading to malware. Being able to prematurely fail a port build based on incongruent metadata provides another mechanism to keep our tree secure from even the stealthiest of attacks even before any patches are offered up in pre-commit project spaces.

Since the wheel Python package itself switched to PEP 517, our setuptools ports can now depend on it rather than the other way around. This opens up the USE_PYTHON=distutils case to build wheels rather than the original method, which would not only unify the stage workflow around the PEP 517 integration frontend but would enjoy the same metadata integrity checks in the process.

In all, this was and will continue to be an ordeal. More modern languages include their own package managers, primarily aimed at developers and their isolated environments, but expect most everyone including operating system-level packagers to use them. At least with Python, there has always been some way to at least sidestep that notion, first with distutils/setuptools, and now with PEP 517 making things even cleaner. We still have to mind things like mapping languages' acceptable package version schemes to our standards and overall metadata congruence, but such can be tackled gradually.

CHARLIE LI is a ports committer focusing on GTK-based desktops, Python, some Rust and amateur radio (callsign: K3CL). Sometimes ventures into other areas for root cause analysis. In real life, he is a technical consultant and sometimes dispatches buses at his local public transportation agency..