MAR 31, 2020 / 6 MIN READ / PACKAGING

What the heck is pyproject.toml?

Recently on Twitter there was a maintainer of a Python project who had a couple of bugs filed against their project due to builds failing (this particular project doesn't provide wheels, only sdists). Eventually it came out that the project was using a `pyproject.toml` file because that's how you configure Black and not for any other purpose. This isn't the first time I have seen setuptools users use `pyproject.toml`

because they were "told to by <insert name of tool>" without knowing the entire point behind the file. And so I decided to write this blog post to try and explain to setuptools users why `pyproject.toml` exists and what it does as it's the future of packaging in the Python ecosystem (if you are not a conda user 😉).

PEP 518 and `pyproject.toml`

I've blogged about this before, but the purpose of PEP 518 was to come up with a way for projects to specify what build tools they required. That's it, real simple and straightforward. Before PEP 518 and the introduction of `pyproject.toml` there was no way for a project to tell a tool like pip what build tools it required in order to build a wheel (let alone an sdist). Now setuptools has a `setup_requires` argument to specify what is necessary to build a project, but your can't read that setting unless you have setuptools installed, which meant you couldn't declare you needed setuptools to read the setting in setuptools. This chicken-and-egg problem is why tools like virtualenv install setuptools by default and why pip always injects setuptools and wheel when running a `setup.py` file regardless of whether you explicitly installed it. Oh, and don't even try to rely on a specific version of setuptools for buildling your project as there was no way to specify that; you had to make do with whatever the user happened to have installed.

But PEP 518 and `pyproject.toml` changed that. Now a tool like pip can read `pyproject.toml`, see what build tools are specified in it, and install those in a virtual environment to build your project. That means you can rely on a specific version of setuptools and 'wheel' if you want. Heck, you can even build with a tool other than setuptools if you want (e.g. flit or Poetry, but since these other tools require `pyproject.toml` their users are already familiar with what's going on). The key point is assumptions no longer need to be made about what is necessary to build your project, which frees up the packaging ecosystem to experiment and grow.

PEP 517 and building wheels

With PEP 518 in place, tools knew *what* needed to be available in order to build a project into a wheel (or sdist). But *how* do you produce a wheel or sdist from a project that has a `pyproject.toml`? This is where PEP 517 comes in. That PEP specifies how build tools are to be executed to build both sdists and wheels. So PEP 518 gets the build tools installed and PEP 517 gets them executed. This opens the

door to using other tools by standardizing how to run build tools. Before, there was no standardized way to build a wheel or sdist except with `python setup.py sdist bdist_wheel` which isn't really flexible; there's no way for the tool running the build to pass in environment details as appropriate, for instance. PEP 517 helped solve that problem.

One other change that PEP 517 & 518 has led to is build isolation. Now that projects can specify arbitrary build tools, tools like pip have to build projects in virtual environments to make sure each project's build tools don't conflict with another project's build tool needs. This also helps with reproducible builds by making sure your build tools are consistent.

Unfortunately this frustrates some setuptools users when they didn't realize a `setup.py` files and/or build environment have become structured in such a way that they can't be built in isolation. For instance, one user was doing their builds offline and didn't have setuptools and 'wheel' in their local cache of wheels (aka their local wheelhouse), so when pip tried to build a project in isolation it failed as pip couldn't find setuptools and 'wheel' to install into the build virtual environment.

Tools standardizing on `pyproject.toml`

An interesting side-effect of PEP 518 trying to introduce a standard file that all projects should (eventually) have is that non-build development tools realized they now had a file where they could put their own configuration. I say this is interesting because originally PEP 518 disallowed this, but people chose to ignore this part of the PEP 😄. We eventually updated the PEP to allow for this use-case since it became obvious people liked the idea of centralizing configuration data in a single file.

And so now projects like Black, coverage.py, towncrier, and tox (in a way) allow you to specify their configurations in `pyproject.toml` instead of in a separate file. Occasionally you do hear people lament the fact that they are adding yet *another* configuration file to their project due to `pyproject.toml`. What I don't think people realize, though, is these project could have also created their own configuration files (and in fact both coverage.py and tox do support their own files). And so, thanks to projects consolidating around `pyproject.toml`, there's actually an argument to be made there are *fewer* configuration files than before thanks to `pyproject.toml`.

How to use `pyproject.toml` with setuptools

Hopefully I have convinced you to introduce `pyproject.toml` into your setuptools-based project so you get benefits like build isolation and the ability to specify the version of setuptools you want to depend on. Now you might be wondering what your `pyproject.toml` should consist of? Unfortunately no one has had the time to document all of this for setuptools, but luckily the issue tracking adding that document outlines what is necessary:

```
[build-system]
requires = ["setuptools >= 40.6.0", "wheel"]
build-backend = "setuptools.build_meta"
```

A `pyproject.toml` file for setuptools users

With that you get to participate in thePEP 517 world of standards! 😉 And as I said, you can now rely on a specific version of setuptools and get build isolation as well (which is why the current directory is not put on `sys.path` automatically; you will need `sys.path.insert(0, os.path.dirname(__file__))` or equivalent if you're importing local files).

But there's a bonus if you use a `pyproject.toml` file with a `setup.cfg` configuration for setuptools: you don't need a `setup.py` file anymore! Since tools like pip are going to call setuptools using the PEP 517 API instead of `setup.py` it means you can delete that `setup.py` file (although if you use editable installs make sure you are using pip 21.3 or newer)!

Where all of this is going

What all of this comes down to is the Python packaging ecosystem is working towards basing itself on *standards*. And those standards are all working towards standardizing *artifacts* and how to work with them. For instance, if we all know how wheels are formatted and how to install them then you don't have to care about how the wheel is made, just that a wheel exists for the thing you want to install and that

it follows the appropriate standards. If you keep pushing this out and standardize more and more it makes it much easier for tools to communicate via artifacts and provide freedom for people to use whatever software they want to produce those artifacts.

For instance, you may have noticed I keep saying "tools like pip" instead of just saying "pip". That's been entirely on purpose. By making all of these standards it means tools don't have to rely solely on pip to do things because "that's how pip does it". As an example, tox could install a wheel by itself by using a library like pep517 to do the building of a wheel and then use another library like distlib to do the wheel installation.

Standards also take out the guessing as to whether something is on purpose or not. This becomes important to make sure everyone agrees on how things should work. There's also coherency as standards start to build on each other and flow into one another nicely. There's also less arguing (eventually 😉) as everyone works toward the same thing that everyone agreed to earlier.

It also takes pressure off of setuptools. It doesn't have to try and be everything to everyone as people can now choose the tool that best fits their project and development style. Same goes for pip.

Besides, don't we all want the platypus? 😉

YOU MIGHT ALSO LIKE...

An experimental pip subcommand for the Python Launcher for Unix

JAN 2, 2024

State of standardized lock files: December 2023

DEC 24, 2023

State of standardized lock files for Python: August 2023

AUG 18, 2023

Contact me

RSS

Powered by Ghost