pyproject.toml, packaging, and you

Moshe Zadka – https://cobordism.com

Acknowledgement of Country

Belmont (in San Francisco Bay Area Peninsula)

Ancestral homeland of the Ramaytush Ohlone people

I live in Belmont, in the San Francisco Bay Area Peninsula. I wish to acknowledge it as the ancestral homeland of the Ramaytush Ohlone people.

Matching Donation: Encircle

LGBTQ+ Charity in Utah

`https://encircletogether.org/`

Up to $100

Encircle is a charity supporting LGBTQ+ youth. I will be matching donations up to $100. E-mail or message me receipts.

0.1 Basics

0.1.1 What is TOML?

What is TOML?

Semantics: JSON + date + float vs. integer

Syntax: more editable than JSON, easier to parse than YAML

TOML is a format for structured data. As far as *semantics* go, TOML can express what JSON can and a little bit more.

TOML has a native "date" type. It can also express which numbers are supposed to be integers and which are supposed to be floats.

TOML is designed to be more editable than JSON. It allows for comments and trailing commas, for example.

It is also designed to be easier to parse than YAML. The specification is shorter, and tries to be fairly clear.

TOML example

```
toml_stuff = """\
[project] # Table -> Dictionary
name = "orbipatch"
authors = [ # Array -> List
# Key/Value -> Dictionary
{ name = "MZ", email = "mz@devskillup.com" }
]
"""
```

A simple example of TOML, sampled from pyproject.toml, can help illustrate some of these. Note that we can have #-style comments.

Tables, as well as Key/Value syntax, parse into dictionaries. There are also arrays, which can include arbitrary objects.

TOML example

```
import tomli # tomllib in Python 3.11+
import json
print(json.dumps(
    tomli.loads(toml_stuff),
    indent=4,
))
```

```
{
    "project": {
        "name": "orbipatch",
        "authors": [
            {
                "name": "MZ",
                "email": "mz@devskillup.com"
            }
        ]
    }
}
```

The tomli library is similar to the built-in tomllib in Python 3.11 and above. Parsing the data above, and redumping it into JSON, helps show the structure.

0.1.2 What is pyproject.toml?

pyproject.toml

Originally: Configure build system, experimenting with alternatives, setuptools plugins, etc.

Now: Still that but also: build-system agnostic metadata, configure ecosystem tools.

The pyproject.toml file was originally intended to configure the build system. This had to be outside of setup.py, in order to install the right version of setuptools and any plugins it depended on.

This is also important to allow other build systems. Allowing alternative build systems, without the backwards-compatibility needs of setuptools, allows faster experimentation.

However, because of TOML's flexibility, it was quickly adopted into other tools. Many tools used the fact that tool.<toolname>.<section> would all be collected into toml_data["tool"]["toolname"].

Finally, some of the fields were standardized. This led to the project section, which is a standard way to express standard metadata about Python packages.

0.1.3 Parsing pyproject.toml

```
minimal_pyproject = """\
[build-system]
```

```
requires = ["setuptool"]
build-backend = "setuptools.build_meta"

[project]
name = "orbipatch"
version = "2022.3.6.2"
description = "silly project named after a niche math thing"
readme = "README.rst"
authors = [{name = "MZ", email = "mz@devskillup.com"}]
"""
```

This is not technically a "minimal" pyproject, but it is fairly minimalistic. The build-system configures the how to build the package.

The requires configures the build-time *dependencies*. The *backend* is the path to a module which has some functions, most importantly build_wheel.

The parameters in project correspond to some often that used to be set in setup.py. More importantly, these are fields which will end up being in a wheel.

Parsing pyproject.toml

```
parsed = tomli.loads(minimal_pyproject)
print(
    "Build system requires:",
    parsed["build-system"].pop("requires")
)
```

```
Build system requires: ['setuptool']
```

When parsing the TOML, sections as well as fields within sections become dictionaries. The result is the typical JSON-like dict-within-dict nested structure.

The build-system section has two important keys. The first one is the requires. More serious requires will sometimes pin the version.

Parsing pyproject.toml

```
print(
    "Build backend",
    parsed["build-system"].pop("build-backend")
)
```

```
Build backend setuptools.build_meta
```

The other field is the build-system. This is a module, typically installed by one of the modules in requires, which knows how to build wheels.

The default backend, here used explicitly, it the setuptools one. The setuptools package exposes its build-system-compatible module as build_meta.

Parsing pyproject.toml

```
print(
    "Project authors:\n",
    parsed["project"].pop("authors")
)
```

```
Project authors:
 [{'name': 'MZ', 'email': 'mz@devskillup.com'}]
```

The project section of the pyproject.toml file has several interesting metadata fields. Of special interest is the authors one, particularly because of its complexity: it is an *array* of *key-value* pairs.

Accessing the name of the first author is a good example of how deep the nesting here is. It would be parsed["project"]["authors"][0]["name"], for a total of four nesting levels.

Parsing pyproject.toml

```
print(
    "Project description:\n",
    parsed["project"].pop("description"),
)
```

```
Project description:
 silly project named after a niche math thing
```

The description is interesting in that even when it is a short description, it is often objectively long. Using a regular string is fine for it, but TOML does support triple quoted strings if the need arises.

Parsing pyproject.toml

```
print(
    "Project:\n",
    json.dumps(parsed.pop("project"), indent=2),
)
```

```
Project:
 {
  "name": "orbipatch",
  "version": "2022.3.6.2",
  "readme": "README.rst"
}
```

The other fields in the project section are some of the most important ones. In particular, name and version make it not only to the contents of the wheel, but also to its *name*.

In theory, pyproject.toml allows inlining the README, or long description. There are good reasons to avoid this "saving".

First, this makes it hard to read natively. Even worse, this won't render on your favorite code-collaboration platform at *all*.

Luckily, you can include it by reference. This allows unifying the project's README on GitHub or similar with the description.

0.1.4 Build system

build system

requires: Dependencies

backend: Module that has the right methods

The build−system has the build *dependencies* and the *backend*. Together, they determine how to build the package into a wheel.

0.1.5 Project

project

Must: name, version

Recommended: Short description (usually inline), Long description (usually from file), License (usually from file, can be inlined), URLs (especially "Homepage")

The project section is interesting. It is not *required* to be there.

If it *does* exist, it must have the name and version fields. Other fields are highly recommended.

Some of the other fields will be displayed directly in PyPI. For example, this is the case with the URLs and the description.

One interesting file to inline is the license. This reduces by one the number of files at your top level.

0.2 Extensibility

0.2.1 The tools section

```
minimal_pyproject = """\
[build-system]
requires = ["setuptool"]
build-backend = "setuptools.build_meta"

[project]
name = "orbipatch"
version = "2022.3.6.2"
description = "silly project named after a niche math thing"
readme = "README.rst"
authors = [{name = "MZ", email = "mz@devskillup.com"}]
```

```
[tool.black]
include = '\.pyi'
"""
parsed = tomli.loads(minimal_pyproject)
```

The tools section

Under "tool.NAME"

```
# [tool.black]
# include = '\.pyi'
print(
    "Black configuration:",
    parsed["tool"]["black"],
)
```

```
Black configuration: {'include': '\\.pyi'}
```

Many, though not all, Python ecosystem tools support pyproject-based configuration. For example, black will allow you to configure which files it should check.

0.2.2 Example: Configuring coverage

```
minimal_pyproject = """\
[build-system]
requires = ["setuptool"]
build-backend = "setuptools.build_meta"

[project]
name = "orbipatch"
version = "2022.3.6.2"
description = "silly project named after a niche math thing"
readme = "README.rst"
authors = [{name = "MZ", email = "mz@devskillup.com"}]

[tool.coverage.run]
branch = true
"""
parsed = tomli.loads(minimal_pyproject)
```

Configuring coverage

Like with setup.cfg, with prefix "tool.":

```
# [tool.coverage.run]
# branch = true
print(
```

```
    "Coverage configuration:",
    parsed["tool"]["coverage"],
)
```

```
Coverage configuration: {'run': {'branch': True}}
```

Coverage can also be configured via pyproject. In this example, we are configuring branch coverage (as you should!)

0.2.3 Example: Configuring isort

```
minimal_pyproject = """\
[build-system]
requires = ["setuptool"]
build-backend = "setuptools.build_meta"

[project]
name = "orbipatch"
version = "2022.3.6.2"
description = "silly project named after a niche math thing"
readme = "README.rst"
authors = [{name = "MZ", email = "mz@devskillup.com"}]

[tool.isort]
src_paths = ["isort", "test"]
"""
parsed = tomli.loads(minimal_pyproject)
```

Configuring isort

Like with setup.cfg, with prefix "tool.":

```
# [tool.isort]
# src_paths = ["isort", "test"]
print(
    "isort configuration:\n",
    parsed["tool"]["isort"],
)
```

```
isort configuration:
 {'src_paths': ['isort', 'test']}
```

The last Python ecosystem tool I'll show is isort. In this example, it shows how to configure the source paths that isort will cover.

0.3 Project metadata

project metadata

project section: packaging semantics edition

Some project metadata is "semantically neutral". Python doesn't care if your project is called "super_cool" or "really_sad".

Other fields in the project metadata are less so. Let's dive into them!

0.3.1 Dependencies

```
minimal_pyproject = """\
[build-system]
requires = ["setuptool"]
build-backend = "setuptools.build_meta"

[project]
name = "orbipatch"
version = "2022.3.6.2"
description = "silly project named after a niche math thing"
readme = "README.rst"
authors = [{name = "MZ", email = "mz@devskillup.com"}]
dependencies = ["six"]
"""
parsed = tomli.loads(minimal_pyproject)
```

Configuring dependencies

```
# [project]
# ...
# dependencies = ["six"]
print(
    "dependencies:",
    parsed["project"]["dependencies"],
)
```

The most common semantically meaningful field is dependencies. It is not universal: some packages do not depend on any others as a matter of policy.

For everything else, the 99%, the dependencies field in the metadata does the trick. Those dependencies will be translated into wheel dependencies by most packaging backends, including setuptools.

0.3.2 Extra dependencies

```
minimal_pyproject = """\
[build-system]
requires = ["setuptool"]
```

```
build-backend = "setuptools.build_meta"

[project]
name = "orbipatch"
version = "2022.3.6.2"
description = "silly project named after a niche math thing"
readme = "README.rst"
authors = [{name = "MZ", email = "mz@devskillup.com"}]

[project.optional-dependencies]
tests = ["pytest"]
docs = ["sphinx"]
"""
parsed = tomli.loads(minimal_pyproject)
```

Configuring optional dependencies

```
# [project.optional-dependencies]
# tests = ["pytest"]
# docs = ["sphinx"]
print(
    "Optional dependencies:\n",
    parsed["project"]["optional-dependencies"],
)

Optional dependencies:
 {'tests': ['pytest'], 'docs': ['sphinx']}
```

Second only to the regular dependencies come the "optional" dependencies. Whether it is dev-only dependencies or dependencies that support a particular way to use the package, these are everywhere.

In theory, you could use a KV format with optional-dependnecies = { tests = ["pytest"] } but this feels a little on the busy side.

0.3.3 Console scripts

```
minimal_pyproject = """\
[build-system]
requires = ["setuptool"]
build-backend = "setuptools.build_meta"

[project]
name = "orbipatch"
version = "2022.3.6.2"
description = "silly project named after a niche math thing"
readme = "README.rst"
```

```
authors = [{name = "MZ", email = "mz@devskillup.com"}]

[project.scripts]
awesome-command = "my_package:main"
"""
parsed = tomli.loads(minimal_pyproject)
```

Configuring console scripts

```
# [project.scripts]
# awesome-command = "my_package:main"
print(
    "scripts:",
    parsed["project"]["scripts"],
)
```

```
scripts: {'awesome-command': 'my_package:main'}
```

Next in the list of popularity is "console scripts". These used to be part of entrypoints but got so big they deserve their own section.

The semantics are full .path.to.module:callable_object. The wheel builder will take care of the rest.

0.3.4 Entrypoints

```
minimal_pyproject = """\
[build-system]
requires = ["setuptool"]
build-backend = "setuptools.build_meta"

[project]
name = "orbipatch"
version = "2022.3.6.2"
description = "silly project named after a niche math thing"
readme = "README.rst"
authors = [{name = "MZ", email = "mz@devskillup.com"}]

[project.entry-points."paste.app_factory"]
main = "my-package:main"
"""
parsed = tomli.loads(minimal_pyproject)
```

Configuring entry points

```
# [project.entry-points."paste.app_factory"]
# main = "my-package:main"
```

```
print(
    "entry points:\n",
    parsed["project"]["entry-points"],
)

entry points:
 {'paste.app_factory': {'main': 'my-package:main'}}
```

Generic entrypoints are not gone, though. Your hand-crafted entrypoints for configuring paste or including a setuptools plugins are still working.

In theory, the semantics are arbitrary, but the convention of module:object is still strong.

0.4 Building with setuptools

0.4.1 Build system details

Build system

Requires: Usual dependency rules (can include minimal, pinned, etc.)

Build system: PEP 517:

```
def build_wheel(
    wheel_directory,
    config_settings=None,
    metadata_directory=None,
):
    ...
```

PEP517 describes how a build system is supposed to work. There are a few optional parts, but the most important part is a build_wheel function. This function is similar to WSGI.

A *builder* is supposed to call this function with appropriate parameters. A *build system* is supposed to implement the function correctly.

This allows multiple build front ends and multiple build back ends. While there are many popular back ends, just like in web servers, front ends are fewer.

0.4.2 Using `python -m build`

Packaging Python

The pyproject.toml file is useful in building packages.

python -m build

Usually: Just works

Hint: Use "src/" structure

Sometimes: BETA! "tools.setup.¡something¿"

The most popular builder front end is −m build. It mostly just works with a correct pyproject.toml which configures setuptools.

The most common failure with setuptools is that it will fail to recognize where your sources are. The best way to fix it is to use the src/ layout, which removes a lot of the ambiguity.

The alternative is to have a section for tools.setup.<something> to configure it explicitly. Note that the fields and their semantics are currently "beta" as far as setuptools is concerned.

0.4.3 Editable installs with setuptools

Editable installs

Empty setup.cfg: no longer needed

For *editable* installs, you do need an empty setup.cfg. Luckily, this is all you need.

Hopefully, at some point, this will no longer be needed.

0.4.4 Dynamic fields

```
minimal_pyproject = """\
[build-system]
requires = ["setuptool"]
build-backend = "setuptools.build_meta"

[project]
name = "orbipatch"
dynamic = ["version"]
"""
```

```
parsed = tomli.loads(minimal_pyproject)
```

Dynamic fields

```
# [project]
# name = "orbipatch"
# dynamic = ["version"]
print(
    "name",
    parsed["project"].pop("name"),
)
print(
    "project",
    parsed["project"],
)
```

```
name orbipatch
project {'dynamic': ['version']}
```

Earlier, I said the project *has* to include name and version. This was a mild lie.

A project has to include the version *or* explicitly note that it will be filled in by other tools. This is what the dynamic field does. This allows using your wheel builder, in our case setuptools, to set the version.

0.4.5 Example: Using `setuptools_scm`

setuptools scm

```
# [build-system]
# requires = [
#   "setuptools",
#   "setuptools_scm",
# ]
#
# [project]
# name = "orbipatch"
# dynamic = ["version"]
```

Setuptools will not set the version itself, though. However, there are plugins which allow setting the version.

One popular plugin is setuptools_scm, which will set the version from tags and other aspects of the version-controlled repository.

0.5 Recap

0.5.1 pyproject.toml: source of truth

pyproject.toml

Packaging!

Also: Everything else

Support in your own tooling

pyproject.toml can be the ultimate source of truth. As more and more tools use it, including packaging, you need fewer "boilerplate" files.

If you write your own tooling for Python, consider supporting pyproject.toml as your configuration file.

0.5.2 Important fields

project fields

Name

Version

Description, license, readme

Dependencies (and Optional dependencies)

Scripts

Entry points

The important fields in pyproject.toml's project section are:

- Name
- Version
- Person facing metadata: Description, license, README
- Dependencies and optional dependencies
- Scripts and entrypoints

0.5.3 setuptools support

setuptools support

Defaults usually good

Use src/ (convention over configuration!)

Configure lightly where you must

"python -m build" future-proofing

With a reasonably-written pyproject.toml, setuptools will usually do the right thing without any specific configuration. This is especially if you use the less ambiguous src/ layout.

Where you need to configure, tread lightly. In your build tooling, use python −m build. This makes switching to flit, hatch, poetry, or something else possible by changing nothing more than pyproject.toml.