Plette Documentation

Tzu-ping Chung, Dan Ryan

Dec 15, 2022

Contents

Plette is an implementation to the Pipfile specification. It offers parsers and style-preserving emitters to both the Pipfile and Pipfile.lock formats, with optional validators to both files.

CHAPTER 1

Quickstart

Plette is available on PyPI. You can install it with pip:

```
pip install plette
```

Now you can load a Pipfile from path like this:

```
>>> import plette
>>> with open('./Pipfile', encoding='utf-8') as f:
...     pipfile = plette.Pipfile.load(f)
...
```

And access contents inside the file:

```
>>> pipfile['scripts']['tests']
'pytest -v tests'
```

Loading from a lock file works similarly:

```
>>> with open('./Pipfile.lock', encoding='utf-8') as f:
...     lockfile = plette.Lockfile.load(f)
...
>>> lockfile.meta.sources[0].url
'https://pypi.org/simple'
```

CHAPTER 2

Contents

2.1 Loading and Saving Files

This chapter discusses how you can load a Pipfile or Pipfile.lock file into a model, and write it back on the disk.

2.1.1 Loading a File into a Model

The Pipfile and lock file can be loaded with the customary `load()` method. This method takes a file-like object to load the file:

```
>>> import plette
>>> with open('Pipfile', encoding='utf-8') as f:
...     pipfile = plette.Pipfile.load(f)
```

Warning: This will not work for Python 2, since the loader is very strict about file encodings, and only accepts a Unicode file. You are required to use `io.open()` to open the file instead.

For manipulating a binary file (maybe because you want to interact with a temporary file created via `tempfile.TemporaryFile()`), `load()` accepts a second, optional argument:

```
>>> import io
>>> with io.open('Pipfile.lock', 'rb') as f:
...     lockfile = plette.Lockfile.load(f, encoding='utf-8')
```

2.1.2 Writing a File from the Model

The loaded model can be written to disk with the customary `dump()` method. For a Pipfile, the dumping logic attempts to preserve the original TOML format as well as possible.

Lock files, on the other hand, are always dumped with the same parameters, to normalize the JSON output. The lock file's format matches the reference implementation, i.e.:

```
indent=4,
separators=(',', ': '),
sort_keys=True,
```

`dump()` always outputs Unicode by default. Similar to `load()`, it takes an optional `encoding` argument. If set, the output would be in bytes, encoded with the specified encoding.

Both the Pipfile and Pipfile.lock are guarenteed to be dumped with a trailing newline at the end.

2.2 Top-Level Sections

This chapter discusses how you can access and manipulate top-level sections in a Pipfile and Pipfile.lock through a loaded model.

2.2.1 Sections as Properties

The Pipfile specification defines a set of standard fields a Pipfile may contain. Those sections are available for access with the dot notation (property access):

```
>>> for script in pipfile.scripts:
...     print(script)
...
build
changelog
docs
draft
release
tests
```

Most property names map directly to the section names defined in the specification, with dashes replaced by underscored:

```
>>> for package in pipfile.dev_packages:
...     print(package)
invoke
parver
towncrier
twine
wheel
pytest
pytest-xdist
pytest-cov
sphinx
sphinx-rtd-theme
```

For ergonomic concerns, some sections have aliases so they have more Pythonic names:

```
>>> for source in pipfile.sources:
...     print(source['url'])
...
https://pypi.org/simple
```

(continues on next page)

(continued from previous page)

```
>>> for key in lockfile.meta:
...     print(key)
...
hash
pipfile-spec
requires
sources
```

Tip: The canonical names are still available as properties, so you can use `pipfile.source` and `lockfile._meta` if you want to.

The section properties are all writable, so you can use them to manipulate contents in of the Pipfile (or Pipfile.lock, although not recommended):

```
>>> pipfile.requires = {'python_version': '3.7'}
>>> pipfile.requires.python_version
'3.7'
```

2.2.2 Key-Value Access

The Pipfile specification allows arbitrary sections. Those sections are available with the bracket (key-value) syntax. Standard dict methods such as `get()` are also available:

```
>>> pipfile.get('pipenv', {}).get('allow_prereleases', False)
False
```

Note: The bracket syntax is also available for standard sections. They are only available in their canonical forms, however, not in normalized forms or aliases, so you will need to use keys like `pipfile['dev-packages']`, `lockfile['_meta']`, etc.

2.2.3 Missing Sections

The Pipfile specification allows any top-level sections to be missing. Plette does *not* attempt to normalize most them, and will raise *KeyError* or *AttributeError* if you access a missing key, to distinguish them from blank sections. You need to catch them manually, or use convenience dict methods (e.g. `get()`).

One exception to this rule is the `source` section in Pipfile. The specification explicitly states there will be a default source, and Plette reflects this by automatically adding one if the loaded Pipfile does not contain any sources. This means that the `source` section will always be present and not empty when you load it.

The automatically generated source contains the following data

```
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true
```

Warning: You *can* delete either the automatically generated source, or the source section itself from the model after it is aloaded. Plette assumes you know what you're doing.

2.3 Working with Nested Structures

It should be enough to work with Plette via the two top-level classes, `Pipfile` and `Lockfile`, in general, and let Plette handle automatic type conversion for you. Sometimes, however, you would want to peek under the hood. This chapter discusses how you can handle those structures yourself.

The `plette.models` submodule contains definitions of nested structures in Pipfile and Pipfile.lock, such as indivisual entries in `[packages]`, `[dev-packages]`, and `lockfile['_meta']`.

2.3.1 The Data View

Every non-scalar valuea you get from Plette (e.g. sequence, mapping) is represented as a *DataView*, or one of its subclasses. This class is simply a wrapper around the basic collection class, and you can access the underlying data strucuture via the `_data` attribute:

```
>>> import plette.models
>>> source = plette.models.Source({
...     'name': 'pypi',
...     'url': 'https://pypi.org/simple',
...     'verify_ssl': True,
... })
...
>>> source._data
{'name': 'pypi', 'url': 'https://pypi.org/simple', 'verify_ssl': True}
```

2.3.2 Data View Collections

There are two special collection classes, `DataViewMapping` and `DataViewSequence`, that hold homogeneous `DataView` members. They are also simply wrappers to `dict` and `list`, respectively, but have specially implemented magic methods to automatically coerce contained data into a `DataView` subclass:

```
>>> sources = plette.models.SourceCollection([source._data])
>>> sources._data
[{'name': 'pypi', 'url': 'https://pypi.org/simple', 'verify_ssl': True}]
>>> type(sources[0])
<class 'plette.models.sources.Source'>
>>> sources[0] == source
True
>>> sources[0] = {
...     'name': 'devpi',
...     'url': 'http://localhost/simple',
...     'verify_ssl': True,
... }
...
>>> sources._data
[{'name': 'devpi', 'url': 'http://localhost/simple', 'verify_ssl': True}]
```

2.4 Validating Data

Plette provides optional validation for input data. This chapter discusses how validation works.

2.4.1 Setting up Validation

Validation is provided by the Cerberus library. You can install it along with Plette manually, or by specifying the "validation" extra when installing Plette:

```
pip install plette[validation]
```

Plette automatically enables validation when Cerberus is available.

2.4.2 Validating Data

Data is validated on input (or when a model is loaded). `ValidationError` is raised when validation fails:

```
>>> plette.models.Source({})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "plette/models/base.py", line 37, in __init__
    self.validate(data)
  File "plette/models/base.py", line 67, in validate
    return validate(cls, data)
  File "plette/models/base.py", line 27, in validate
    raise ValidationError(data, v)
plette.models.base.ValidationError: {}
```

This exception class has a `validator` member to allow you to access the underlying Cerberus validator, so you can know what exactly went wrong:

```
>>> try:
...     plette.models.Source({'verify_ssl': True})
... except plette.models.ValidationError as e:
...     for error in e.validator._errors:
...         print(error.schema_path)
...
('name', 'required')
('url', 'required')
```

See Ceberus's error handling documentation to know how the errors are represented and reported.