validate-pyproject Documentation

Release 0.16

Anderson Bravalheri

Jan 23, 2024

CONTENTS

validate-pyproject is a command line tool and Python library for validating `pyproject.toml` files based on JSON Schema, and includes checks for **PEP 517**, **PEP 518** and **PEP 621**.

CHAPTER ONE

CONTENTS

1.1 validate-pyproject

Automated checks on `pyproject.toml` powered by JSON Schema definitions

Important: This project is **experimental** and under active development. Issue reports and contributions are very welcome.

1.1.1 Description

With the approval of PEP 517 and PEP 518, the Python community shifted towards a strong focus on standardisation for packaging software, which allows more freedom when choosing tools during development and make sure packages created using different technologies can interoperate without the need for custom installation procedures.

This shift became even more clear when PEP 621 was also approved, as a standardised way of specifying project metadata and dependencies.

`validate-pyproject` was born in this context, with the mission of validating `pyproject.toml` files, and make sure they are compliant with the standards and PEPs. Behind the scenes, `validate-pyproject` relies on JSON Schema files, which, in turn, are also a standardised way of checking if a given data structure complies with a certain specification.

1.1.2 Usage

The easiest way of using `validate-pyproject` is via CLI. To get started, you need to install the package, which can be easily done using pipx:

```
$ pipx install 'validate-pyproject[all]'
```

Now you can use `validate-pyproject` as a command line tool:

```
# in you terminal
$ validate-pyproject --help
$ validate-pyproject path/to/your/pyproject.toml
```

You can also use `validate-pyproject` in your Python scripts or projects:

```
# in your python code
from validate_pyproject import api, errors

# let's assume that you have access to a `loads` function
# responsible for parsing a string representing the TOML file
# (you can check the `toml` or `tomli` projects for that)
pyproject_as_dict = loads(pyproject_toml_str)

# now we can use validate-pyproject
validator = api.Validator()

try:
    validator(pyproject_as_dict)
except errors.ValidationError as ex:
    print(f"Invalid Document: {ex.message}")
```

To do so, don't forget to add it to your virtual environment or specify it as a project or library dependency.

Note: When you install `validate-pyproject[all]`, the packages `tomli`, `packaging` and `trove-classifiers` will be automatically pulled as dependencies. `tomli` is a lightweight parser for TOML, while `packaging` and `trove-classifiers` are used to validate aspects of PEP 621.

If you are only interested in using the Python API and wants to keep the dependencies minimal, you can also install `validate-pyproject` (without the `[all]` extra dependencies group).

If you don't install `trove-classifiers`, `validate-pyproject` will try to download a list of valid classifiers directly from PyPI (to prevent that, set the environment variable `NO_NETWORK` or `VALIDATE_PYPROJECT_NO_NETWORK`).

On the other hand, if `validate-pyproject` cannot find a copy of `packaging` in your environment, the validation will fail.

More details about `validate-pyproject` and its Python API can be found in our docs, which includes a description of the used JSON schemas, instructions for using it in a *pre-compiled* way and information about extending the validation with your own plugins.

Tip: If you consider contributing to this project, have a look on our contribution guides.

1.1.3 pre-commit

`validate-pyproject` can be installed as a pre-commit hook:

```
---
repos:
  - repo: https://github.com/abravalheri/validate-pyproject
    rev: main
    hooks:
      - id: validate-pyproject
```

By default, this `pre-commit` hook will only validate the `pyproject.toml` file at the root of the project repository. You can customize that by defining a custom regular expression pattern using the `files` parameter.

You can also use `pre-commit autoupdate` to update to the latest stable version of `validate-pyproject` (recommended).

1.1.4 Note

This project and its sister project ini2toml were initially created in the context of PyScaffold, with the purpose of helping migrating existing projects to PEP 621-style configuration when it is made available on `setuptools`. For details and usage information on PyScaffold see https://pyscaffold.org/.

1.2 Schemas

The following sections represent the schemas used in `validate-pyproject`. They were automatically rendered via sphinx-jsonschema for quick reference. In case of doubts or confusion, you can also have a look on the raw JSON files in json-schemas.

1.2.1 Data structure for `pyproject.toml` files

File format containing build-time configurations for the Python ecosystem. **PEP 517** initially defined a build-system independent format for source trees which was complemented by **PEP 518** to provide a way of specifying dependencies for building Python projects. Please notice the `project` table (as initially defined in **PEP 621**) is not included in this schema and should be considered separately.

https://packaging.python.org/en/latest/specifications/declaring-build-dependencies/				
type	*object*			
properties				
• build-system	Table used to store build-related data			
	type	*object*		
	properties			
	• **requires**	List of dependencies in the **PEP 508** format required to execute the build system. Please notice that the resulting dependency graph **MUST NOT contain cycles**		
		type	*array*	
		items	type	*string*
	• build-backend	Python object that will be used to perform the build according to **PEP 517**		
		type	*string*	
		format	pep517-backend-reference	
	• backend-path	List of directories to be prepended to `sys.path` when loading the back-end, and running its hooks		
		type	*array*	
		items	type	*string*
	additionalProperties	False		
• project	https://packaging.python.org/en/latest/specifications/declaring-project-metadata/			
• tool	type	*object*		
additionalProperties	False			

1.2.2 Package metadata stored in the `project` table

Data structure for the **project** table inside `pyproject.toml` (as initially defined in **PEP 621**)

https://packaging.python.org/en/latest/specifications/declaring-project-metadata/			
type	*object*		
properties			
• **name**	The name (primary identifier) of the project. MUST be statically defined.		
	type	*string*	
	format	pep508-identifier	
• version	The version of the project as supported by **PEP 440**.		
	type	*string*	
	format	pep440	
• de-scrip-tion	The summary description of the project		
	type	*string*	
• readme	Full/detailed description of the project in the form of a README with meaning similar to the one defined in core metadata's Description		
	oneOf	Relative path to a text file (UTF-8) containing the full description of the project. If the file path ends in case-insensitive `.md` or `.rst` suffixes, then the content-type is respectively `text/markdown` or `text/x-rst`	
		type	*string*
		type	*object*

continues on next page

Table 1 – continued from previous page

		allOf	anyOf	properties		
				• **file**	Relative path to a text file containing the full description of the project.	
					type	*string*
				properties		
				• **text**	Full text describing the project.	
					type	*string*
			properties			
			• **content-type**	Content-type (**RFC 1341**) of the full description (e.g. `text/markdown`). The `charset` parameter is assumed UTF-8 when not present.		
				type	*string*	
• requires-python	The Python version requirements of the project.					
	type	*string*				
	format	pep508-versionspec				
• license	Project license.					
	oneOf	properties				
		• **file**	Relative path to the file (UTF-8) which contains the license for the project.			
			type	*string*		
		properties				
		• **text**	The license of the project whose meaning is that of the License field from the core metadata.			
			type	*string*		
• authors	The people or organizations considered to be the 'authors' of the project. The exact meaning is open to interpretation (e.g. original or primary authors, current maintainers, or owners of the package).					
	type	*array*				
	items	*Author or Maintainer*				
• maintainers	The people or organizations considered to be the 'maintainers' of the project. Similarly to `authors`, the exact meaning is open to interpretation.					
	type	*array*				
	items	*Author or Maintainer*				
• keywords	List of keywords to assist searching for the distribution in a larger catalog.					
	type	*array*				
	items	type	*string*			
• classifiers	Trove classifiers which apply to the project.					
	type	*array*				
	items	PyPI classifier.				
		type	*string*			
		format	trove-classifier			
• urls	URLs associated with the project in the form `label => value`.					
	type	*object*				
	patternProperties					
	• ^.+$	type	*string*			
		format	url			
	additional-Properties	False				
• scripts	Instruct the installer to create command-line wrappers for the given entry points.					
	Entry-points					
• gui-scripts	Instruct the installer to create GUI wrappers for the given entry points. The difference between `scripts` and `gui-scripts` is only relevant in Windows.					

continues on next page

Table 1 – continued from previous page

	Entry-points		
• entry-points	Instruct the installer to expose the given modules/functions via `entry-point` discovery mechanism (useful for plugins). More information available in the Python packaging guide.		
	patternProperties		
	• ^.+$	*Entry-points*	
	additional-Properties	False	
• dependencies	Project (mandatory) dependencies.		
	type	*array*	
	items	*Dependency*	
• optional-dependencies	Optional dependency for the project		
	type	*object*	
	patternProperties		
	• ^.+$	type	*array*
		items	*Dependency*
	additional-Properties	False	
• dynamic	Specifies which fields are intentionally unspecified and expected to be dynamically provided by build tools		
	type	*array*	
	items	enum	version, description, readme, requires-python, license, authors, maintainers, keywords, classifiers, urls, scripts, gui-scripts, entry-points, dependencies, optional-dependencies
additional-Properties	False		
if	not	properties	
		• **dynamic**	version is listed in `dynamic`
then	version should be statically defined in the `version` field		

Author or Maintainer

#/definitions/author			
type	*object*		
properties			
• name	MUST be a valid email name, i.e. whatever can be put as a name, before an email, in **RFC 822**.		
	type	*string*	
• email	MUST be a valid email address		
	type	*string*	
	format	idn-email	
additionalProperties	False		

Entry-points

Entry-points are grouped together to indicate what sort of capabilities they provide. See the packaging guides and setuptools docs for more information.

#/definitions/entry-point-group		
type	*object*	
patternProperties		
• ^.+$	Reference to a Python object. It is either in the form `importable.module`, or `importable.module:object.attr`.	
	type	*string*
	format	python-entrypoint-reference
additionalProperties	False	

Dependency

Project dependency specification according to PEP 508

#/definitions/dependency	
type	*string*
format	pep508

1.2.3 `tool` table

According to **PEP 518**, tools can define their own configuration inside `pyproject.toml` by using custom subtables under `tool`.

In `validate-pyproject`, schemas for these subtables can be specified via *Plugins*. The following subtables are defined by *built-in* plugins (i.e. plugins that are included in the default distribution of `validate-pyproject`):

`tool.setuptools` table

`setuptools`-specific configurations that can be set by users that require customization. These configurations are completely optional and probably can be skipped when creating simple packages. They are equivalent to some of the Keywords used by the `setup.py` file, and can be set via the `tool.setuptools` table. It considers only `setuptools` parameters that are not covered by **PEP 621**; and intentionally excludes `dependency_links` and `setup_requires` (incompatible with modern workflows/standards).

https://setuptools.pypa.io/en/latest/userguide/pyproject_config.html			
type	*object*		
properties			
• platforms	type	*array*	
	items	type	*string*
• provides	Package and virtual package names contained within this package **(not supported by pip)**		
	type	*array*	
	items	type	*string*
		format	pep508-identifier
• obsoletes	Packages which this package renders obsolete **(not supported by pip)**		
	type	*array*	

continues on next page

Table 2 – continued from previous page

	items	type	*string*	
		format	pep508-identifier	
• zip-safe	Whether the project can be safely installed and run from a zip file. **OBSOLETE**: only relevant for `pkg_resources`, `easy_install` and `setup.py install` in the context of `eggs` (**DEPRECATED**).			
	type	*boolean*		
• script-files	Legacy way of defining scripts (entry-points are preferred). Equivalent to the `script` keyword in `setup.py` (it was renamed to avoid confusion with entry-point based `project.scripts` defined in **PEP 621**). **DISCOURAGED**: generic script wrappers are tricky and may not work properly. Whenever possible, please use `project.scripts` instead.			
	type	*array*		
	items	type	*string*	
• eager-resources	Resources that should be extracted together, if any of them is needed, or if any C extensions included in the project are imported. **OBSOLETE**: only relevant for `pkg_resources`, `easy_install` and `setup.py install` in the context of `eggs` (**DEPRECATED**).			
	type	*array*		
	items	type	*string*	
• packages	Packages that should be included in the distribution. It can be given either as a list of package identifiers or as a `dict`-like structure with a single key `find` which corresponds to a dynamic call to `setuptools.config.expand.find_packages` function. The `find` key is associated with a nested `dict`-like structure that can contain `where`, `include`, `exclude` and `namespaces` keys, mimicking the keyword arguments of the associated function.			
	oneOf	*Array of Python package identifiers*		
		type	*array*	
		items	*Valid package name*	
		'find:' directive		
• package-dir	`dict`-like structure mapping from package names to directories where their code can be found. The empty string (as key) means that all packages are contained inside the given directory will be included in the distribution.			
	type	*object*		
	patternProperties			
	• ^.*$	type	*string*	
	additionalProperties	False		
• package-data	Mapping from package names to lists of glob patterns. Usually this option is not needed when using `include-package-data = true` For more information on how to include data files, check `setuptools` docs.			
	type	*object*		
	patternProperties			
	• ^.*$	type	*array*	
		items	type	*string*
	additionalProperties	False		
• include-package-data	Automatically include any data files inside the package directories that are specified by `MANIFEST.in` For more information on how to include data files, check `setuptools` docs.			
	type	*boolean*		
• exclude-package-data	Mapping from package names to lists of glob patterns that should be excluded For more information on how to include data files, check `setuptools` docs.			
	type	*object*		
	patternProperties			

continues on next page

Table 2 – continued from previous page

	• ^.*$	type	*array*	
		items	type	*string*
	additionalProperties	False		
• namespace-packages	**DEPRECATED**: use implicit namespaces instead (**PEP 420**).			
	type	*array*		
	items	type	*string*	
		format	python-module-name	
• py-modules	Modules that setuptools will manipulate			
	type	*array*		
	items	type	*string*	
		format	python-module-name	
• data-files	`dict`-like structure where each key represents a directory and the value is a list of glob patterns that should be installed in them. **DISCOURAGED**: please notice this might not work as expected with wheels. Whenever possible, consider using data files inside the package directories (or create a new namespace package that only contains data files). See data files support.			
	type	*object*		
	patternProperties			
	• ^.*$	type	*array*	
		items	type	*string*
• cmdclass	Mapping of distutils-style command names to `setuptools.Command` subclasses which in turn should be represented by strings with a qualified class name (i.e., "dotted" form with module), e.g.: `cmdclass = {mycmd = "pkg.subpkg.module.CommandClass"}` The command class should be a directly defined at the top-level of the containing module (no class nesting).			
	type	*object*		
	patternProperties			
	• ^.*$	type	*string*	
		format	python-qualified-identifier	
• license-files	**PROVISIONAL**: list of glob patterns for all license files being distributed. (likely to become standard with **PEP 639**). By default: `['LICEN[CS]E*', 'COPYING*', 'NOTICE*', 'AUTHORS*']`			
	type	*array*		
	items	type	*string*	
• dynamic	Instructions for loading **PEP 621**-related metadata dynamically			
	type	*object*		
	properties			
	• version	A version dynamically loaded via either the `attr:` or `file:` directives. Please make sure the given file or attribute respects **PEP 440**. Also ensure to set `project.dynamic` accordingly.		
		oneOf	*'attr:' directive*	
			'file:' directive	
	• classifiers	*'file:' directive*		
	• description	*'file:' directive*		

continues on next page

Table 2 – continued from previous page

	• entry-points	*'file:' directive*			
	• dependencies	*'file:' directive for dependencies*			
	• optional-dependenci	type	*object*		
		patternProperties			
		• .+	*'file:' directive for dependencies*		
		additionalProperties	False		
	• readme	type	*object*		
		anyOf	*'file:' directive*		
			type	*object*	
			properties		
			• content-type	type	*string*
			• file	#/definitions/file-directive/properties/file	
			additionalProperties	False	
	additionalProperties	False			
additionalProperties	False				

Valid package name

Valid package name (importable or **PEP 561**).

#/definitions/package-name		
type	*string*	
anyOf	type	*string*
	format	python-module-name
	type	*string*
	format	pep561-stub-name

‘file:’ directive

Value is read from a file (or list of files and then concatenated)

#/definitions/file-directive				
type	*object*			
properties				
• **file**	oneOf	type	*string*	
		type	*array*	
		items	type	*string*
additionalProperties	False			

‘file:’ directive for dependencies

allOf	**BETA**: subset of the `requirements.txt` format without `pip` flags and options (one **PEP 508**-compliant string per line, lines that are blank or start with # are excluded). See dynamic metadata.
	‘file:’ directive

‘attr:’ directive

Value is read from a module attribute. Supports callables and iterables; unsupported types are cast via `str()`

#/definitions/attr-directive		
type	*object*	
properties		
• **attr**	type	*string*
	format	python-qualified-identifier
additionalProperties	False	

'find:' directive

#/definitions/find-directive				
type	*object*			
properties				
• find	Dynamic package discovery.			
	type	*object*		
	properties			
	• where	Directories to be searched for packages (Unix-style relative path)		
		type	*array*	
		items	type	*string*
	• exclude	Exclude packages that match the values listed in this field. Can container shell-style wildcards (e.g. `'pkg.*'`)		
		type	*array*	
		items	type	*string*
	• include	Restrict the found packages to just the ones listed in this field. Can container shell-style wildcards (e.g. `'pkg.*'`)		
		type	*array*	
		items	type	*string*
	• namespaces	When `True`, directories without a `__init__.py` file will also be scanned for **PEP 420**-style implicit namespaces		
		type	*boolean*	
	additionalProperties	False		
additionalProperties	False			

1.3 Embedding validations in your project

`validate-pyproject` can be used as a dependency in your project in the same way you would use any other Python library, i.e. by adding it to the same virtual environment you run your code in, or by specifying it as a project or library dependency that is automatically retrieved every time your project is installed. Please check *this example* for a quick overview on how to use the Python API.

Alternatively, if you cannot afford having external dependencies in your project you can also opt to *"vendorise"*[1] `validate-pyproject`. This can be done automatically via tools such as vendoring or vendorize and many others others, however this technique will copy several files into your project.

However, if you want to keep the amount of files to a minimum, `validate-pyproject` offers a different solution that consists in pre-compiling the JSON Schemas (thanks to fastjsonschema).

After *installing* `validate-pyproject` this can be done via CLI as indicated in the command below:

```
# in you terminal
$ python -m validate_pyproject.pre_compile --help
$ python -m validate_pyproject.pre_compile -O dir/for/genereated_files
```

This command will generate a few files under the directory given to the CLI. Please notice this directory should, ideally, be empty, and will correspond to a "sub-package" in your package (a `__init__.py` file will be generated, together with a few other ones).

Assuming you have created a `genereated_files` directory, and that the value for the `--main-file` option in the CLI was kept as the default `__init__.py`, you should be able to invoke the validation function in your code by doing:

[1] The words "vendorise" or "vendoring" in this text refer to the act of copying external dependencies to a folder inside your project, so they are distributed in the same package and can be used directly without relying on installation tools, such as pip.

```python
from .genereated_files import validate, JsonSchemaValueException

try:
    validate(dict_representing_the_parsed_toml_file)
except JsonSchemaValueException:
    print("Invalid File")
```

1.4 FAQ

1.4.1 Why JSON Schema?

This design was initially inspired by an issue in the `setuptools` repository, and brings a series of advantages and disadvantages.

Disadvantages include the fact that JSON Schema might be limited at times and incapable of describing more complex checks. Additionally, error messages produced by JSON Schema libraries might not be as pretty as the ones used when bespoke validation is in place.

On the other hand, the fact that JSON Schema is standardised and have a widespread usage among several programming language communities, means that a bigger number of people can easily understand the schemas and modify them if necessary.

Additionally, **PEP 518** already includes a JSON Schema representation, which suggests that it can be used at the same time as specification language and validation tool.

1.4.2 Why `fastjsonschema`?

While there are other (more popular) JSON Schema libraries in the Python community, none of the ones the original author of this package investigated (other than fastjsonschema) fulfilled the following requirements:

- Minimal number of dependencies (ideally 0)
- Easy to "vendorise", i.e. copy the source code of the package to be used directly without requiring installation.

fastjsonschema has no dependency and can generate validation code directly, which bypass the need for copying most of the files when *"embedding"*.

1.4.3 Why draft-07 of JSON Schema and not a more modern version?

The most modern version of JSON Schema supported by fastjsonschema is Draft 07. It is not as bad as it may sound, it even supports if-then-else-style conditions…

1.4.4 Why the URLs used as `$id` do not point to the schemas themselves?

According to the JSON Schema, the $id keyword is just a unique identifier to differentiate between schemas and is not required to match a real URL. The text on the standard is:

> Note that this URI is an identifier and not necessarily a network locator. In the case of a network-addressable URL, a schema need not be downloadable from its canonical URI.

This information is confirmed in a similar document submitted to the IETF.

1.5 Contributing

Welcome to `validate-pyproject` contributor's guide.

This document focuses on getting any potential contributor familiarized with the development processes, but other kinds of contributions are also appreciated.

If you are new to using git or have never collaborated in a project previously, please have a look at contribution-guide.org. Other resources are also listed in the excellent guide created by FreeCodeCamp.

Please notice, all users and contributors are expected to be **open, considerate, reasonable, and respectful**. When in doubt, Python Software Foundation's Code of Conduct is a good reference in terms of behavior guidelines.

1.5.1 Issue Reports

If you experience bugs or general issues with `validate-pyproject`, please have a look on the issue tracker. If you don't see anything useful there, please feel free to fire an issue report.

Tip: Please don't forget to include the closed issues in your search. Sometimes a solution was already reported, and the problem is considered **solved**.

New issue reports should include information about your programming environment (e.g., operating system, Python version) and steps to reproduce the problem. Please try also to simplify the reproduction steps to a very minimal example that still illustrates the problem you are facing. By removing other factors, you help us to identify the root cause of the issue.

1.5.2 Documentation Improvements

You can help improve `validate-pyproject` docs by making them more readable and coherent, or by adding missing information and correcting mistakes.

`validate-pyproject` documentation uses Sphinx as its main documentation compiler. This means that the docs are kept in the same repository as the project code, in the form of reStructuredText files, and that any documentation update is done in the same way was a code contribution.

Tip: Please notice that the GitHub web interface provides a quick way of propose changes in `validate-pyproject`'s files. While this mechanism can be tricky for normal code contributions, it works perfectly fine for contributing to the docs, and can be quite handy.

If you are interested in trying this method out, please navigate to the `docs` folder in the source repository, find which file you would like to propose changes and click in the little pencil icon at the top, to open GitHub's code editor. Once you finish editing the file, please write a message in the form at the bottom of the page describing which changes have you made and what are the motivations behind them and submit your proposal.

When working on documentation changes in your local machine, you can compile them using `tox`:

```
tox -e docs
```

and use Python's built-in web server for a preview in your web browser (`http://localhost:8000`):

```
python3 -m http.server --directory 'docs/_build/html'
```

1.5.3 Code Contributions

Understanding how the project works

If you have a change in mind, please have a look in our *Developer Guide*. It explains the main aspects of the project and provide a brief overview on how it is organised and how to implement *Plugins*.

Submit an issue

Before you work on any non-trivial code contribution it's best to first create a report in the issue tracker to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

Create an environment

Before you start coding, we recommend creating an isolated virtual environment to avoid any problems with your installed Python packages. This can easily be done via either `virtualenv`:

```
virtualenv <PATH TO VENV>
source <PATH TO VENV>/bin/activate
```

or Miniconda:

```
conda create -n validate-pyproject python=3 six virtualenv pytest pytest-cov
conda activate validate-pyproject
```

Clone the repository

1. Create an user account on GitHub if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on GitHub.
3. Clone this copy to your local disk:

   ```
   git clone git@github.com:YourLogin/validate-pyproject.git
   cd validate-pyproject
   ```

4. You should run:

```
pip install -U pip setuptools -e .
```

to be able to import the package under development in the Python REPL.

5. Install `pre-commit`:

```
pip install pre-commit
pre-commit install
```

`validate-pyproject` comes with a lot of hooks configured to automatically help the developer to check the code being written.

Implement your changes

1. Create a branch to hold your changes:

```
git checkout -b my-feature
```

and start making changes. Never work on the main branch!

2. Start your work on this branch. Don't forget to add docstrings to new functions, modules and classes, especially if they are part of public APIs.

3. Add yourself to the list of contributors in `AUTHORS.rst`.

4. When you're done editing, do:

```
git add <MODIFIED FILES>
git commit
```

to record your changes in git.

Please make sure to see the validation messages from `pre-commit` and fix any eventual issues. This should automatically use ruff to check/fix the code style in a way that is compatible with the project.

Important: Don't forget to add unit tests and documentation in case your contribution adds an additional feature and is not just a bugfix.

Moreover, writing a descriptive commit message is highly recommended. In case of doubt, you can check the commit history with:

```
git log --graph --decorate --pretty=oneline --abbrev-commit --all
```

to look for recurring communication patterns.

5. Please check that your changes don't break any unit tests with:

```
tox
```

(after having installed `tox` with `pip install tox` or `pipx`).

You can also use `tox` to run several other pre-configured tasks in the repository. Try `tox -av` to see a list of the available checks.

Submit your contribution

1. If everything works fine, push your local branch to GitHub with:

   ```
   git push -u origin my-feature
   ```

2. Go to the web page of your fork and click "Create pull request" to send your changes for review.

   Find more detailed information in creating a PR. You might also want to open the PR as a draft first and mark it as ready for review after the feedbacks from the continuous integration (CI) system or any required fixes.

Troubleshooting

The following tips can be used when facing problems to build or test the package:

1. Make sure to fetch all the tags from the upstream repository. The command `git describe --abbrev=0 --tags` should return the version you are expecting. If you are trying to run CI scripts in a fork repository, make sure to push all the tags. You can also try to remove all the egg files or the complete egg folder, i.e., `.eggs`, as well as the `*.egg-info` folders in the `src` folder or potentially in the root of your project.

2. Sometimes `tox` misses out when new dependencies are added, especially to `setup.cfg` and `docs/requirements.txt`. If you find any problems with missing dependencies when running a command with `tox`, try to recreate the `tox` environment using the `-r` flag. For example, instead of:

   ```
   tox -e docs
   ```

   Try running:

   ```
   tox -r -e docs
   ```

3. Make sure to have a reliable `tox` installation that uses the correct Python version (e.g., 3.7+). When in doubt you can run:

   ```
   tox --version
   # OR
   which tox
   ```

   If you have trouble and are seeing weird errors upon running `tox`, you can also try to create a dedicated virtual environment with a `tox` binary freshly installed. For example:

   ```
   virtualenv .venv
   source .venv/bin/activate
   .venv/bin/pip install tox
   .venv/bin/tox -e all
   ```

4. Pytest can drop you in an interactive session in the case an error occurs. In order to do that you need to pass a `--pdb` option (for example by running `tox -- -k <NAME OF THE FALLING TEST> --pdb`). You can also setup breakpoints manually instead of using the `--pdb` option.

1.5.4 Maintainer tasks

If you are part of the group of maintainers and have correct user permissions on PyPI, the following steps can be used to release a new version for `validate-pyproject`:

1. Make sure all unit tests are successful.
2. Tag the current commit on the main branch with a release tag, e.g., `v1.2.3`.
3. Push the new tag to the upstream repository, e.g., `git push upstream v1.2.3`
4. Clean up the `dist` and `build` folders with `tox -e clean` (or `rm -rf dist build`) to avoid confusion with old builds and Sphinx docs.
5. Run `tox -e build` and check that the files in `dist` have the correct version (no `.dirty` or git hash) according to the git tag. Also check the sizes of the distributions, if they are too big (e.g., > 500KB), unwanted clutter may have been accidentally included.
6. Run `tox -e publish -- --repository pypi` and check that everything was uploaded to PyPI correctly.

1.6 Developer Guide

This document describes the internal architecture and main concepts behind `validate-pyproject` and targets contributors and plugin writers.

1.6.1 How it works

`validate-pyproject` relies mostly on a set of specification documents represented as JSON Schema. To run the checks encoded under these schema files `validate-pyproject` uses the fastjsonschema package.

This procedure is defined in the *`api`* module, specifically under the *`Validator`* class. *`Validator`* objects use *`SchemaRegistry`* instances to store references to the JSON schema documents being used for the validation. The *`formats`* module is also important to this process, since it defines how to validate the custom values for the `"format"` field defined in JSON Schema, for `"string"` values.

Checks for **PEP 517**, **PEP 518** and **PEP 621** are performed by default, however these standards do not specify how the `tool` table and its subtables are populated.

Since different tools allow different configurations, it would be impractical to try to create schemas for all of them inside the same project. Instead, `validate-pyproject` allows *Plugins* to provide extra JSON Schemas, against which `tool` subtables can be checked.

1.6.2 Plugins

Plugins are a way of extending the built-in functionality of `validate-pyproject`, can be simply described as functions that return a JSON schema parsed as a Python `dict`:

```
def plugin(tool_name: str) -> dict:
    ...
```

These functions receive as argument the name of the tool subtable and should return a JSON schema for the data structure **under** this table (it **should** not include the table name itself as a property).

To use a plugin you can pass a `plugins` argument to the *`Validator`* constructor, but you will need to wrap it with *`PluginWrapper`* to be able to specify which `tool` subtable it would be checking:

```python
from validate_pyproject import api, plugins


def your_plugin(tool_name: str) -> dict:
    return {
        "$id": "https://your-urn-or-url",  # $id is mandatory
        "type": "object",
        "description": "Your tool configuration description",
        "properties": {
            "your-config-field": {"type": "string", "format": "python-module-name"}
        },
    }


available_plugins = [
    *plugins.list_from_entry_points(),
    plugins.PluginWrapper("your-tool", your_plugin),
]
validator = api.Validator(available_plugins)
```

Please notice that you can also make your plugin "autoloadable" by creating and distributing your own Python package as described in the following section.

Distributing Plugins

To distribute plugins, it is necessary to create a Python package with a `validate_pyproject.tool_schema` entry-point.

For the time being, if using setuptools, this can be achieved by adding the following to your `setup.cfg` file:

```
# in setup.cfg
[options.entry_points]
validate_pyproject.tool_schema =
    your-tool = your_package.your_module:your_plugin
```

When using a **PEP 621**-compliant backend, the following can be add to your `pyproject.toml` file:

```
# in pyproject.toml
[project.entry-points."validate_pyproject.tool_schema"]
your-tool = "your_package.your_module:your_plugin"
```

The plugin function will be automatically called with the `tool_name` argument as same name as given to the entrypoint (e.g. `your_plugin("your-tool")`).

Also notice plugins are activated in a specific order, using Python's built-in `sorted` function.

1.7 License

Mozilla Public License, version 2.0

1. Definitions

1.1. “Contributor”

means each individual or legal entity that creates, contributes to the creation of, or owns Covered Software.

1.2. “Contributor Version”

means the combination of the Contributions of others (if any) used by a Contributor and that particular Contributor’s Contribution.

1.3. “Contribution”

means Covered Software of a particular Contributor.

1.4. “Covered Software”

means Source Code Form to which the initial Contributor has attached the notice in Exhibit A, the Executable Form of such Source Code Form, and Modifications of such Source Code Form, in each case including portions thereof.

1.5. “Incompatible With Secondary Licenses”
means

a. that the initial Contributor has attached the notice described in Exhibit B to the Covered Software; or

b. that the Covered Software was made available under the terms of version 1.1 or earlier of the License, but not also under the terms of a Secondary License.

1.6. “Executable Form”

means any form of the work other than Source Code Form.

1.7. “Larger Work”

means a work that combines Covered Software with other material, in a separate file or files, that is not Covered Software.

1.8. “License”

means this document.

1.9. “Licensable”

means having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently, any and all of the rights conveyed by this License.

1.10. “Modifications”

means any of the following:

a. any file in Source Code Form that results from an addition to, deletion from, or modification of the contents of Covered Software; or

b. any new file in Source Code Form that contains any Covered Software.

1.11. “Patent Claims” of a Contributor

means any patent claim(s), including without limitation, method, process, and apparatus claims, in any patent Licensable by such Contributor that would be infringed, but for the grant of the License, by the making, using, selling, offering for sale, having made, import, or transfer of either its Contributions or its Contributor Version.

1.12. "Secondary License"

means either the GNU General Public License, Version 2.0, the GNU Lesser General Public License, Version 2.1, the GNU Affero General Public License, Version 3.0, or any later versions of those licenses.

1.13. "Source Code Form"

means the form of the work preferred for making modifications.

1.14. "You" (or "Your")

means an individual or a legal entity exercising rights under this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50%) of the outstanding shares or beneficial ownership of such entity.

2. License Grants and Conditions

2.1. Grants

Each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license:

a. under intellectual property rights (other than patent or trademark) Licensable by such Contributor to use, reproduce, make available, modify, display, perform, distribute, and otherwise exploit its Contributions, either on an unmodified basis, with Modifications, or as part of a Larger Work; and

b. under Patent Claims of such Contributor to make, use, sell, offer for sale, have made, import, and otherwise transfer either its Contributions or its Contributor Version.

2.2. Effective Date

The licenses granted in Section 2.1 with respect to any Contribution become effective for each Contribution on the date the Contributor first distributes such Contribution.

2.3. Limitations on Grant Scope

The licenses granted in this Section 2 are the only rights granted under this License. No additional rights or licenses will be implied from the distribution or licensing of Covered Software under this License. Notwithstanding Section 2.1(b) above, no patent license is granted by a Contributor:

a. for any code that a Contributor has removed from Covered Software; or

b. for infringements caused by: (i) Your and any other third party's modifications of Covered Software, or (ii) the combination of its Contributions with other software (except as part of its Contributor Version); or

c. under Patent Claims infringed by Covered Software in the absence of its Contributions.

This License does not grant any rights in the trademarks, service marks, or logos of any Contributor (except as may be necessary to comply with the notice requirements in Section 3.4).

2.4. Subsequent Licenses

No Contributor makes additional grants as a result of Your choice to distribute the Covered Software under a subsequent version of this License (see Section 10.2) or under the terms of a Secondary License (if permitted under the terms of Section 3.3).

2.5. Representation

Each Contributor represents that the Contributor believes its Contributions are its original creation(s) or it has sufficient rights to grant the rights to its Contributions conveyed by this License.

2.6. Fair Use

This License is not intended to limit any rights You have under applicable copyright doctrines of fair use, fair dealing, or other equivalents.

2.7. Conditions

Sections 3.1, 3.2, 3.3, and 3.4 are conditions of the licenses granted in Section 2.1.

3. Responsibilities

3.1. Distribution of Source Form

All distribution of Covered Software in Source Code Form, including any Modifications that You create or to which You contribute, must be under the terms of this License. You must inform recipients that the Source Code Form of the Covered Software is governed by the terms of this License, and how they can obtain a copy of this License. You may not attempt to alter or restrict the recipients' rights in the Source Code Form.

3.2. Distribution of Executable Form

If You distribute Covered Software in Executable Form then:

a. such Covered Software must also be made available in Source Code Form, as described in Section 3.1, and You must inform recipients of the Executable Form how they can obtain a copy of such Source Code Form by reasonable means in a timely manner, at a charge no more than the cost of distribution to the recipient; and

b. You may distribute such Executable Form under the terms of this License, or sublicense it under different terms, provided that the license for the Executable Form does not attempt to limit or alter the recipients' rights in the Source Code Form under this License.

3.3. Distribution of a Larger Work

You may create and distribute a Larger Work under terms of Your choice, provided that You also comply with the requirements of this License for the Covered Software. If the Larger Work is a combination of Covered Software with a work governed by one or more Secondary Licenses, and the Covered Software is not Incompatible With Secondary Licenses, this License permits You to additionally distribute such Covered Software under the terms of such Secondary License(s), so that the recipient of the Larger Work may, at their option, further distribute the Covered Software under the terms of either this License or such Secondary License(s).

3.4. Notices

You may not remove or alter the substance of any license notices (including copyright notices, patent notices, disclaimers of warranty, or limitations of liability) contained within the Source Code Form of the Covered Software, except that You may alter any license notices to the extent required to remedy known factual inaccuracies.

3.5. Application of Additional Terms

You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Software. However, You may do so only on Your own behalf, and not on behalf of any Contributor. You must make it absolutely clear that any such warranty, support, indemnity, or liability obligation is offered by You alone, and You hereby agree to indemnify every Contributor for any liability incurred by such Contributor as a result of warranty, support, indemnity or liability terms You offer. You may include additional disclaimers of warranty and limitations of liability specific to any jurisdiction.

4. Inability to Comply Due to Statute or Regulation

If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Software due to statute, judicial order, or regulation then You must: (a) comply with the terms of this License to the maximum extent possible; and (b) describe the limitations and the code they affect. Such description must

1.8 Contributors

- Anderson Bravalheri <andersonbravalheri@gmail.com>

1.9 Changelog

1.9.1 Version 0.16

- Fix setuptools `readme` field , #116
- Fix `oneOf` <> `anyOf` in setuptools schema, #117
- Add previously omitted type keywords for string values, #117
- Add schema validator check, #118
- Add `SchemaStore` conversion script, #119
- Allow tool(s) to be specified via URL (added CLI option: `--tool`), #121

- Support `uint` formats (as used by Ruff's schema), #128
- Allow schemas to be loaded from `SchemaStore` (added CLI option: `--store`), #133

1.9.2 Version 0.15

- Update `setuptools` schema definitions, #112
- Add `__repr__` to plugin wrapper, by @henryiii #114
- Fix standard `$schema` ending `#`, by @henryiii #113

1.9.3 Version 0.14

- Ensure reporting show more detailed error messages for `RedefiningStaticFieldAsDynamic`, #104
- Add support for `repo-review`, by @henryiii in #105

1.9.4 Version 0.13

- Make it clear when using input from `stdin`, #96
- Fix summary for `allOf`, #100
- **`setuptools` plugin:**
  - Improve validation of `attr` directives, #101

1.9.5 Version 0.12.2

- **`setuptools` plugin:**
  - Fix problem with `license-files` patterns, by removing `default` value.

1.9.6 Version 0.12.1

- **`setuptools` plugin:**
  - Allow PEP 561 stub names in `tool.setuptools.package-dir`, #87

1.9.7 Version 0.12

- **`setuptools` plugin:**
  - Allow PEP 561 stub names in `tool.setuptools.packages`, #86

1.9.8 Version 0.11

- Improve error message for invalid replacements in the `pre_compile` CLI, #71
- Allow package to be build from git archive, #53
- Improve error message for invalid replacements in the `pre_compile` CLI, #71
- Error-out when extra keys are added to `project.authors/maintainers`, #82
- De-vendor `fastjsonschema`, #83

1.9.9 Version 0.10.1

- Ensure `LICENSE.txt` is added to wheel.

1.9.10 Version 0.10

- Add `NOTICE.txt` to `license_files`, #58
- Use default SSL context when downloading classifiers from PyPI, #57
- Remove `setup.py`, #52
- Explicitly limit oldest supported Python version
- Replace usage of `cgi.parse_header` with `email.message.Message`

1.9.11 Version 0.9

- Use `tomllib` from the standard library in Python 3.11+, #42

1.9.12 Version 0.8.1

- Workaround typecheck inconsistencies between different Python versions
- Publish **PEP 561** type hints, #43

1.9.13 Version 0.8

- New pre-commit hook, #40
- Allow multiple TOML files to be validated at once via **CLI** (*no changes regarding the Python API*).

1.9.14 Version 0.7.2

- **`setuptools` plugin:**
  - Allow `dependencies/optional-dependencies` to use file directives, #37

1.9.15 Version 0.7.1

- CI: Enforced doctests
- CI: Add more tests for situations when downloading classifiers is disabled

1.9.16 Version 0.7

- **Deprecated** use of `validate_pyproject.vendoring`. This module is replaced by `validate_pyproject.pre_compile`.

1.9.17 Version 0.6.1

- Fix validation of `version` to ensure it is given either statically or dynamically, #29

1.9.18 Version 0.6

- Allow private classifiers, #26
- **`setuptools` plugin:**
  - Remove `license` and `license-files` from `tool.setuptools.dynamic`, #27

1.9.19 Version 0.5.2

- Exported `ValidationError` from the main file when vendored, PR #23
- Removed `ValidationError` traceback to avoid polluting the user logs with generate code, PR #24

1.9.20 Version 0.5.1

- Fixed typecheck errors (only found against GitHub Actions, not Cirrus CI), PR #22

1.9.21 Version 0.5

- Fixed entry-points format to allow values without the `:obj.attr part`, PR #8
- Improved trove-classifier validation, even when the package is not installed, PR #9
- Improved URL validation when scheme prefix is not present, PR #14
- Vendor fastjsonschema to facilitate applying patches and latest updates, PR #15
- Remove fixes for old version of fastjsonschema, PR #16, PR #19
- Replaced usage of `importlib.resources` legacy functions with the new API, PR #17
- Improved error messages, PR #18
- Added GitHub Actions for automatic test and release of tags, PR #11

1.9.22 Version 0.4

- Validation now fails when non-standardised fields to be added to the project table (issue #4, PR #5)
- Terminology and schema names were also updated to avoid specific PEP numbers and refer instead to living standards (issue #6, PR #7)

1.9.23 Version 0.3.3

- Remove upper pin from the tomli dependency by @hukkin (PR #1)
- Fix failing blacken-docs pre-commit hook by @hukkin (PR #2)
- Update versions of tools and containers used in the CI setup (PR #3)

1.9.24 Version 0.3.2

- Updated `fastjsonschema` dependency version.
- Removed workarounds for `fastjsonschema` pre 2.15.2

1.9.25 Version 0.3.1

- **`setuptools` plugin:**
  - Fixed missing `required` properties for the `attr:` and `file:` directives (previously empty objects were allowed).

1.9.26 Version 0.3

- **`setuptools` plugin:**
  - Added support for `readme`, `license` and `license-files` via `dynamic`.

    Warning: `license` and `license-files` in `dynamic` are **PROVISIONAL** they are likely to change depending on **PEP 639**

  - Removed support for `tool.setuptools.dynamic.{scripts,gui-scripts}`. Dynamic values for `project.{scripts,gui-scripts}` are expected to be dynamically derived from `tool.setuptools.dynamic.entry-points`.

1.9.27 Version 0.2

- **`setuptools` plugin:**
  - Added `cmdclass` support

1.9.28 Version 0.1

- **`setuptools` plugin:**
  - Added `data-files` support (although this option is marked as deprecated).
  - Unified `tool.setuptools.packages.find` and `tool.setuptools.packages.find-namespace` options by adding a new keyword `namespaces`
  - `tool.setuptools.packages.find.where` now accepts a list of directories (previously only one directory was accepted).

1.9.29 Version 0.0.1

- Initial release with basic functionality

1.10 validate_pyproject

1.10.1 validate_pyproject package

Subpackages

validate_pyproject.plugins package

Module contents

exception `validate_pyproject.plugins.`**`ErrorLoadingPlugin`**(*plugin: str = '', entry_point: EntryPoint | None = None*)

Bases: `RuntimeError`

There was an error loading '{plugin}'. Please make sure you have installed a version of the plugin that is compatible with {package} {version}. You can also try uninstalling it.

class `validate_pyproject.plugins.`**`PluginProtocol`**

Bases: `object`

property fragment: `str`

property help_text: `str`

property id: `str`

property schema: *`Schema`*

property tool: `str`

class `validate_pyproject.plugins.`**`PluginWrapper`**(*tool: str, load_fn: Plugin*)

Bases: `object`

property fragment: `str`

property help_text: `str`

property id: `str`

property schema: *Schema*

property tool: str

`validate_pyproject.plugins.`**`iterate_entry_points`**(*group: str = 'validate_pyproject.tool_schema'*) → Iterable[EntryPoint]

Produces a generator yielding an EntryPoint object for each plugin registered via `setuptools` entry point mechanism.

This method can be used in conjunction with *`load_from_entry_point`* to filter the plugins before actually loading them.

`validate_pyproject.plugins.`**`list_from_entry_points`**(*group: str = 'validate_pyproject.tool_schema', filtering: ~typing.Callable[[~importlib.metadata.EntryPoint], bool] = <function <lambda>>*) → List[*PluginWrapper*]

Produces a list of plugin objects for each plugin registered via `setuptools` entry point mechanism.

Parameters

- **group** – name of the setuptools' entry point group where plugins is being registered
- **filtering** – function returning a boolean deciding if the entry point should be loaded and included (or not) in the final list. A `True` return means the plugin should be included.

`validate_pyproject.plugins.`**`load_from_entry_point`**(*entry_point: EntryPoint*) → *PluginWrapper*

Carefully load the plugin, raising a meaningful message in case of errors

validate_pyproject.pre_compile package

Submodules

validate_pyproject.pre_compile.cli module

class `validate_pyproject.pre_compile.cli.`**`CliParams`**(*plugins*, *output_dir*, *main_file*, *replacements*, *loglevel*, *tool*, *store*)

Bases: `NamedTuple`

loglevel: int

Alias for field number 4

main_file: str

Alias for field number 2

output_dir: Path

Alias for field number 1

plugins: List[*PluginWrapper*]

Alias for field number 0

replacements: Mapping[str, str]

Alias for field number 3

store: str

Alias for field number 6

tool: Sequence[str]
Alias for field number 5

`validate_pyproject.pre_compile.cli.`**`JSON_dict`**(*name: str*, *value: str*) → Dict[str, Any]

`validate_pyproject.pre_compile.cli.`**`ensure_dict`**(*name: str*, *value: Any*) → dict

`validate_pyproject.pre_compile.cli.`**`main`**(*args: Sequence[str] = ()*) → int

`validate_pyproject.pre_compile.cli.`**`parser_spec`**(*plugins: Sequence[PluginWrapper]*) → Dict[str, dict]

`validate_pyproject.pre_compile.cli.`**`run`**(*args: Sequence[str] = ()*) → int

Module contents

`validate_pyproject.pre_compile.`**`copy_fastjsonschema_exceptions`**(*output_dir: Path*, *replacements: Dict[str, str]*) → Path

`validate_pyproject.pre_compile.`**`copy_module`**(*name: str*, *output_dir: Path*, *replacements: Dict[str, str]*) → Path

`validate_pyproject.pre_compile.`**`load_licenses`**() → Dict[str, str]

`validate_pyproject.pre_compile.`**`pre_compile`**(*output_dir: str | PathLike = '.'*, *main_file: str = '__init__.py'*, *original_cmd: str = ''*, *plugins: AllPlugins | Sequence[PluginProtocol] = AllPlugins.ALL_PLUGINS*, *text_replacements: Mapping[str, str] = mappingproxy({'from fastjsonschema import': 'from .fastjsonschema_exceptions import'})*, ***, *extra_plugins: Sequence[PluginProtocol] = ()*) → Path

Populate the given `output_dir` with all files necessary to perform the validation. The validation can be performed by calling the `validate` function inside the the file named with the `main_file` value. `text_replacements` can be used to

`validate_pyproject.pre_compile.`**`replace_text`**(*text: str*, *replacements: Dict[str, str]*) → str

`validate_pyproject.pre_compile.`**`write_main`**(*file_path: Path*, *schema: Schema*, *replacements: Dict[str, str]*) → Path

`validate_pyproject.pre_compile.`**`write_notice`**(*out: Path*, *main_file: str*, *cmd: str*, *replacements: Dict[str, str]*) → Path

validate_pyproject.vendoring package

Submodules

validate_pyproject.vendoring.cli module

`validate_pyproject.vendoring.cli.`**`main`**(**args: Any*, ***kwargs: Any*) → Any

`validate_pyproject.vendoring.cli.`**`run`**(**args: Any*, ***kwargs: Any*) → Any

Module contents

validate_pyproject.vendoring.**vendorify**(**args: Any*, ***kwargs: Any*) → Any

Submodules

validate_pyproject.api module

Retrieve JSON schemas for validating dicts representing a `pyproject.toml` file.

class validate_pyproject.api.**AllPlugins**(*value*)

Bases: `Enum`

An enumeration.

ALL_PLUGINS = 1

class validate_pyproject.api.**RefHandler**(*registry: Mapping[str, Schema]*)

Bases: `Mapping`[`str`, `Callable`[[`str`], *`Schema`*]]

`fastjsonschema` allows passing a dict-like object to load external schema `$ref``s. Such objects map the URI schema (e.g. ``http`, `https`, `ftp`) into a function that receives the schema URI and returns the schema (as parsed JSON) (otherwise `urllib` is used and the URI is assumed to be a valid URL). This class will ensure all the URIs are loaded from the local registry.

class validate_pyproject.api.**SchemaRegistry**(*plugins: Sequence[PluginProtocol] = ()*)

Bases: `Mapping`[`str`, *`Schema`*]

Repository of parsed JSON Schemas used for validating a `pyproject.toml`.

During instantiation the schemas equivalent to PEP 517, PEP 518 and PEP 621 will be combined with the schemas for the `tool` subtables provided by the plugins.

Since this object work as a mapping between each schema `$id` and the schema itself, all schemas provided by plugins **MUST** have a top level `$id`.

property main: str

Top level schema for validating a `pyproject.toml` file

property spec_version: str

Version of the JSON Schema spec in use

class validate_pyproject.api.**Validator**(*plugins: ~typing.Sequence[PluginProtocol] | ~validate_pyproject.api.AllPlugins = AllPlugins.ALL_PLUGINS, format_validators: ~typing.Mapping[str, ~typing.Callable[[str], bool]] = mappingproxy({'chain': <class 'itertools.chain'>, 'pep440': <function pep440>, 'pep508-identifier': <function pep508_identifier>, 'pep508': <function pep508>, 'pep508-versionspec': <function pep508_versionspec>, 'pep517-backend-reference': <function pep517_backend_reference>, 'trove-classifier': <function trove_classifier>, 'pep561-stub-name': <function pep561_stub_name>, 'url': <function url>, 'python-identifier': <function python_identifier>, 'python-qualified-identifier': <function python_qualified_identifier>, 'python-module-name': <function python_module_name>, 'python-entrypoint-group': <function python_entrypoint_group>, 'python-entrypoint-name': <function python_entrypoint_name>, 'python-entrypoint-reference': <function python_entrypoint_reference>, 'uint8': <function uint8>, 'uint16': <function uint16>, 'uint': <function uint>, 'int': <function int>}), extra_validations: ~typing.Sequence[~typing.Callable[[~validate_pyproject.types.T], ~validate_pyproject.types.T]] = (<function validate_project_dynamic>,), *, extra_plugins: ~typing.Sequence[PluginProtocol] = ()*)

Bases: `object`

property extra_validations: Sequence[Callable[[T], T]]

List of extra validation functions that run after the JSON Schema check

property formats: Mapping[str, Callable[[str], bool]]

Mapping between JSON Schema formats and functions that validates them

property generated_code: str

property registry: *SchemaRegistry*

property schema: *Schema*

Top level `pyproject.toml` JSON Schema

validate_pyproject.api.**load**(*name: str, package: str = 'validate_pyproject', ext: str = '.schema.json'*) → *Schema*

Load the schema from a JSON Schema file. The returned dict-like object is immutable.

validate_pyproject.api.**load_builtin_plugin**(*name: str*) → *Schema*

validate_pyproject.api.**read_text**(*package: str | module, resource: str*) → str

validate_pyproject.cli module

class `validate_pyproject.cli.`**`CliParams`**(*input_file*, *plugins*, *tool*, *store*, *loglevel*, *dump_json*)

Bases: `NamedTuple`

`dump_json:` `bool`

Alias for field number 5

`input_file:` `List[TextIOBase]`

Alias for field number 0

`loglevel:` `int`

Alias for field number 4

`plugins:` `List[`*`PluginWrapper`*`]`

Alias for field number 1

`store:` `str`

Alias for field number 3

`tool:` `List[str]`

Alias for field number 2

class `validate_pyproject.cli.`**`Formatter`**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: `RawTextHelpFormatter`

`validate_pyproject.cli.`**`critical_logging`**() → Generator[None, None, None]

Make sure the logging level is set even before parsing the CLI args

`validate_pyproject.cli.`**`exceptions2exit`**() → Generator[None, None, None]

`validate_pyproject.cli.`**`main`**(*args: Sequence[str] = ()*) → int

Wrapper allowing `Translator` to be called in a CLI fashion.

Instead of returning the value from `Translator.translate()`, it prints the result to the given `output_file` or `stdout`.

Parameters

args (*`List[str]`*) – command line parameters as list of strings (for example `["--verbose", "setup.cfg"]`).

`validate_pyproject.cli.`**`parse_args`**(*args: ~typing.Sequence[str], plugins: ~typing.Sequence[~validate_pyproject.plugins.PluginWrapper], description: str = 'Validate a given TOML file', get_parser_spec: ~typing.Callable[[~typing.Sequence[~validate_pyproject.plugins.PluginWrapper]], ~typing.Dict[str, dict]] = <function __meta__>, params_class: ~typing.Type[~validate_pyproject.cli.T] = <class 'validate_pyproject.cli.CliParams'>*) → T

Parse command line parameters

Parameters

args – command line parameters as list of strings (for example `["--help"]`).

Returns: command line parameters namespace

`validate_pyproject.cli.`**`plugins_help`**(*plugins: Sequence[PluginWrapper]*) → str `validate_pyproject.cli.`**`run`**(*args: Sequence[str] = ()*) → int

Wrapper allowing `Translator` to be called in a CLI fashion.

Instead of returning the value from `Translator.translate()`, it prints the result to the given `output_file` or `stdout`.

Parameters
`args` (*`List[str]`*) – command line parameters as list of strings (for example `["--verbose", "setup.cfg"]`).

`validate_pyproject.cli.`**`select_plugins`**(*plugins: Sequence[*PluginWrapper*], enabled: Sequence[str] = (), disabled: Sequence[str] = ()*) → List[*PluginWrapper*]

`validate_pyproject.cli.`**`setup_logging`**(*loglevel: int*) → None

Setup basic logging

Parameters
`loglevel` – minimum loglevel for emitting messages

validate_pyproject.error_reporting module

`exception` `validate_pyproject.error_reporting.`**`ValidationError`**(*message, value=None, name=None, definition=None, rule=None*)

Bases: *`JsonSchemaValueException`*

Report violations of a given JSON schema.

This class extends `JsonSchemaValueException` by adding the following properties:

- `summary`: an improved version of the `JsonSchemaValueException` error message with only the necessary information)
- `details`: more contextual information about the error like the failing schema itself and the value that violates the schema.

Depending on the level of the verbosity of the `logging` configuration the exception message will be only `summary` (default) or a combination of `summary` and `details` (when the logging level is set to `logging.DEBUG`).

`details` `= ''`

`summary` `= ''`

`validate_pyproject.error_reporting.`**`detailed_errors`**`()` → Generator[None, None, None]

validate_pyproject.errors module

`exception` `validate_pyproject.errors.`**`InvalidSchemaVersion`**(*name: str, given_version: str, required_version: str*)

Bases: *`JsonSchemaDefinitionException`*

All schemas used in the validator should be specified using the same version as the toplevel schema ({version!r}).

Schema for {name!r} has version {given!r}.

exception validate_pyproject.errors.**JsonSchemaDefinitionException**

Bases: *JsonSchemaException*

Exception raised by generator of validation function.

exception validate_pyproject.errors.**JsonSchemaException**

Bases: ValueError

Base exception of `fastjsonschema` library.

exception validate_pyproject.errors.**JsonSchemaValueException**(*message*, *value=None*, *name=None*, *definition=None*, *rule=None*)

Bases: *JsonSchemaException*

Exception raised by validation function. Available properties:

- `message` containing human-readable information what is wrong (e.g. `data.property[index] must be smaller than or equal to 42`),
- invalid `value` (e.g. `60`),
- `name` of a path in the data structure (e.g. `data.property[index]`),
- `path` as an array in the data structure (e.g. `['data', 'property', 'index']`),
- the whole `definition` which the `value` has to fulfil (e.g. `{'type': 'number', 'maximum': 42}`),
- `rule` which the `value` is breaking (e.g. `maximum`)
- and `rule_definition` (e.g. `42`).

Changed in version 2.14.0: Added all extra properties.

property path

property rule_definition

exception validate_pyproject.errors.**SchemaMissingId**(*reference: str*)

Bases: *JsonSchemaDefinitionException*

All schemas used in the validator MUST define a unique toplevel *"$id"*. No *"$id"* was found for schema associated with {reference!r}.

exception validate_pyproject.errors.**SchemaWithDuplicatedId**(*schema_id: str*)

Bases: *JsonSchemaDefinitionException*

All schemas used in the validator MUST define a unique toplevel *"$id"*. *$id = {schema_id!r}* was found at least twice.

exception validate_pyproject.errors.**ValidationError**(*message*, *value=None*, *name=None*, *definition=None*, *rule=None*)

Bases: *JsonSchemaValueException*

Report violations of a given JSON schema.

This class extends `JsonSchemaValueException` by adding the following properties:

- `summary`: an improved version of the `JsonSchemaValueException` error message with only the necessary information)
- `details`: more contextual information about the error like the failing schema itself and the value that violates the schema.

Depending on the level of the verbosity of the `logging` configuration the exception message will be only `summary` (default) or a combination of `summary` and `details` (when the logging level is set to `logging.DEBUG`).

`details = ''`

`summary = ''`

validate_pyproject.extra_validations module

The purpose of this module is implement PEP 621 validations that are difficult to express as a JSON Schema (or that are not supported by the current JSON Schema library).

`exception` `validate_pyproject.extra_validations.`**`RedefiningStaticFieldAsDynamic`**`(`*message*, *value=None*, *name=None*, *definition=None*, *rule=None*`)`

Bases: *`ValidationError`*

According to PEP 621:

Build back-ends MUST raise an error if the metadata specifies a field statically as well as being listed in dynamic.

`validate_pyproject.extra_validations.`**`validate_project_dynamic`**`(`*pyproject: T*`)` → T

validate_pyproject.formats module

`validate_pyproject.formats.`**`int`**`(`*value: int*`)` → bool

`validate_pyproject.formats.`**`pep440`**`(`*version: str*`)` → bool

`validate_pyproject.formats.`**`pep508`**`(`*value: str*`)` → bool

`validate_pyproject.formats.`**`pep508_identifier`**`(`*name: str*`)` → bool

`validate_pyproject.formats.`**`pep508_versionspec`**`(`*value: str*`)` → bool

Expression that can be used to specify/lock versions (including ranges)

`validate_pyproject.formats.`**`pep517_backend_reference`**`(`*value: str*`)` → bool

`validate_pyproject.formats.`**`pep561_stub_name`**`(`*value: str*`)` → bool

`validate_pyproject.formats.`**`python_entrypoint_group`**`(`*value: str*`)` → bool

`validate_pyproject.formats.`**`python_entrypoint_name`**`(`*value: str*`)` → bool

`validate_pyproject.formats.`**`python_entrypoint_reference`**`(`*value: str*`)` → bool

`validate_pyproject.formats.`**`python_identifier`**`(`*value: str*`)` → bool

`validate_pyproject.formats.`**`python_module_name`**`(`*value: str*`)` → bool

`validate_pyproject.formats.`**`python_qualified_identifier`**`(`*value: str*`)` → bool

`validate_pyproject.formats.`**`trove_classifier`**`(`*value: str*`)` → bool `validate_pyproject.formats.`**`uint`**(*value: int*) → bool

`validate_pyproject.formats.`**`uint16`**(*value: int*) → bool

`validate_pyproject.formats.`**`uint8`**(*value: int*) → bool

`validate_pyproject.formats.`**`url`**(*value: str*) → bool

validate_pyproject.remote module

`class` `validate_pyproject.remote.`**`RemotePlugin`**(*, *tool: str*, *schema: Schema*, *fragment: str = ''*)

Bases: `object`

`classmethod from_str`(*tool_url: str*) → Self

`classmethod from_url`(*tool: str*, *url: str*) → Self

`validate_pyproject.remote.`**`load_store`**(*pyproject_url: str*) → Generator[*RemotePlugin*, None, None]

Takes a URL / Path and loads the tool table, assuming it is a set of ref's. Currently ignores "inline" sections. This is the format that SchemaStore (https://json.schemastore.org/pyproject.json) is in.

validate_pyproject.repo_review module

`class` `validate_pyproject.repo_review.`**`VPP001`**

Bases: `object`

Validate pyproject.toml

`static check`(*pyproject: Dict[str, Any]*) → str

`family = 'validate-pyproject'`

`validate_pyproject.repo_review.`**`repo_review_checks`**() → Dict[str, *VPP001*]

`validate_pyproject.repo_review.`**`repo_review_families`**(*pyproject: Dict[str, Any]*) → Dict[str, Dict[str, str]]

validate_pyproject.types module

`validate_pyproject.types.`**`FormatValidationFn`**

Should return `True` when the input string satisfies the format

alias of `Callable`[[`str`], `bool`]

`validate_pyproject.types.`**`Plugin`**

A plugin is something that receives the name of a *tool* sub-table (as defined in PEPPEP621) and returns a *`Schema`*.

For example `plugin("setuptools")` should return the JSON schema for the `[tool.setuptools]` table of a `pyproject.toml` file.

alias of `Callable`[[`str`], *`Schema`*]

`class` `validate_pyproject.types.`**`Schema`**

JSON Schema represented as a Python dict

alias of `Mapping`

`validate_pyproject.types.`**`ValidationFn`**

Custom validation function. It should receive as input a mapping corresponding to the whole `pyproject.toml` file and raise a `fastjsonschema.JsonSchemaValueException` if it is not valid.

alias of `Callable`[[`T`], `T`]

Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX