nitpick Documentation

Release 0.33.2

W. Augusto Andreoli

May 29, 2023

CONTENTS:

Command-line tool and flake8 plugin to enforce the same settings across multiple language-independent projects.

Useful if you maintain multiple projects and are tired of copying/pasting the same INI/TOML/YAML/JSON keys and values over and over, in all of them.

The CLI now has a `nitpick fix` command that modifies configuration files directly (pretty much like black and isort do with Python files). See *Command-line interface* for more info.

Many more features are planned for the future, check the roadmap.

Note: This project is still a work in progress, so the API is not fully defined:

- *The style file* syntax might have changes before the 1.0 stable release;
- The numbers in the `NIP*` error codes might change; don't fully rely on them;
- See also *Breaking changes*.

CHAPTER

ONE

QUICKSTART

1.1 Install

Install in an isolated environment with pipx:

```
# Latest PyPI release
pipx install nitpick

# Development branch from GitHub
pipx install git+https://github.com/andreoliwa/nitpick
```

On macOS/Linux, install the latest release with Homebrew:

```
brew install andreoliwa/formulae/nitpick

# Development branch from GitHub
brew install andreoliwa/formulae/nitpick --HEAD
```

On Arch Linux, install with yay:

```
yay -Syu nitpick
```

Add to your project with Poetry:

```
poetry add --dev nitpick
```

Or install it with pip:

```
pip install -U nitpick
```

1.2 Run

To fix and modify your files directly:

```
nitpick fix
```

To check for errors only:

```
nitpick check
```

Nitpick is also a flake8 plugin, so you can run this on a project with at least one Python (`.py`) file:

```
flake8 .
```

Nitpick will download and use the opinionated default style file.

You can use it as a template to *Configure your own style*.

1.3 Run as a pre-commit hook

If you use pre-commit on your project, add this to the `.pre-commit-config.yaml` in your repository:

```yaml
repos:
  - repo: https://github.com/andreoliwa/nitpick
    rev: v0.33.2
    hooks:
      - id: nitpick
```

There are 3 available hook IDs:

- `nitpick` and `nitpick-fix` both run the `nitpick fix` command;
- `nitpick-check` runs `nitpick check`.

If you want to run Nitpick as a flake8 plugin instead:

```yaml
repos:
  - repo: https://github.com/PyCQA/flake8
    rev: 4.0.1
    hooks:
      - id: flake8
        additional_dependencies: [nitpick]
```

To install the `pre-commit` and `commit-msg` Git hooks:

```
pre-commit install --install-hooks
pre-commit install -t commit-msg
```

To start checking all your code against the default rules:

```
pre-commit run --all-files
```

1.4 Run as a MegaLinter plugin

If you use MegaLinter you can run Nitpick as a plugin. Add the following two entries to your `.mega-linter.yml` configuration file:

```yaml
PLUGINS:
  - https://raw.githubusercontent.com/andreoliwa/nitpick/v0.33.2/mega-linter-plugin-
↪nitpick/nitpick.megalinter-descriptor.yml
ENABLE_LINTERS:
  - NITPICK
```

1.5 Modify files directly

Nitpick includes a CLI to apply your style and modify the configuration files directly:

```
nitpick fix
```

Read more details here: *Command-line interface*.

CHAPTER

TWO

STYLES

2.1 The style file

A "Nitpick code style" is a TOML file with the settings that should be present in config files from other tools.

Example of a style:

```toml
["pyproject.toml".tool.black]
line-length = 120

["pyproject.toml".tool.poetry.dev-dependencies]
pylint = "*"

["setup.cfg".flake8]
ignore = "D107,D202,D203,D401"
max-line-length = 120
inline-quotes = "double"

["setup.cfg".isort]
line_length = 120
multi_line_output = 3
include_trailing_comma = true
force_grid_wrap = 0
combine_as_imports = true
```

This example style will assert that:

- … black, isort and flake8 have a line length of 120;
- … flake8 and isort are configured with the above options in `setup.cfg`;
- … Pylint is present as a Poetry dev dependency in `pyproject.toml`.

2.2 Configure your own style

After creating your own TOML file with your style, add it to your `pyproject.toml` file. See *Configuration* for details.

You can also check *the style library*, and use the styles to build your own.

2.3 Default search order for a style

1. A file or URL configured in the `pyproject.toml` file, `[tool.nitpick]` section, `style` key, as described in *Configuration*.
2. Any nitpick-style.toml file found in the current directory (the one in which flake8 runs from) or above.
3. If no style is found, then the default style file from GitHub is used.

2.4 Style file syntax

A style file contains basically the configuration options you want to enforce in all your projects.

They are just the config to the tool, prefixed with the name of the config file.

E.g.: To configure the black formatter with a line length of 120, you use this in your `pyproject.toml`:

```
[tool.black]
line-length = 120
```

To enforce that all your projects use this same line length, add this to your nitpick-style.toml file:

```
["pyproject.toml".tool.black]
line-length = 120
```

It's the same exact section/key, just prefixed with the config file name (`"pyproject.toml".`)

The same works for `setup.cfg`.

To configure mypy to ignore missing imports in your project, this is needed on `setup.cfg`:

```
[mypy]
ignore_missing_imports = true
```

To enforce all your projects to ignore missing imports, add this to your nitpick-style.toml file:

```
["setup.cfg".mypy]
ignore_missing_imports = true
```

2.5 Special configurations

2.5.1 Comparing elements on lists

Note: At the moment, this feature only works on **:ref:`the YAML plugin <yamlplugin>`_**.

On YAML files, a list (called a "sequence" in the YAML spec) can contain different elements:

- Scalar values like int, float and string;
- Dictionaries or objects with key/value pairs (called "mappings" in the YAML spec)
- Other lists (sequences).

The default behaviour for all lists: when applying a style, Nitpick searches if the element already exists in the list; if not found, the element is appended at the end of list.

1. With scalar values, Nitpick compares the elements by their exact value. Strings are compared in a case-sensitive manner, and spaces are considered.
2. Dicts are compared by a "hash" of their key/value pairs.

On lists containing dicts, it is possible to define a custom "list key" used to search for an element, overriding the default compare mechanism above.

If an element is found by its key, the other key/values are compared and Nitpick applies the difference. If the key doesn't exist, the new dict is appended at the end of the list.

Some files have a predefined configuration:

1. On `.pre-commit-config.yaml`, repos are searched by their hook IDs.
2. On GitHub Workflow YAML files, steps are searched by their names.

You can define your own list keys for your YAML files (or override the predefined configs above) by using `__list_keys` on your style:

```
["path/from/root/to/your/config.yaml".__list_keys]
"<list expression>" = "<search key expression>"
```

`<list expression>`:

- a dotted path to a list, e.g. `repos`, `path.to.some.key`; or
- a pattern containing wildcards (`?` and `*`). For example, `jobs.*.steps`: all `jobs` containing `steps` will use the same `<search key expression>`.

`<search key expression>`:

- a key from the dict inside the list, e.g. `name`; or
- a parent key and its child key separated by a dot, e.g. `hooks.id`.

Warning: For now, only two-level nesting is possible: parent and child keys.

If you have suggestions for predefined list keys for other popular YAML files not covered by Nitpick (e.g.: GitLab CI config), feel free to create an issue or submit a pull request.

2.6 Breaking changes

Warning: Below are the breaking changes in the style before the API is stable. If your style was working in a previous version and now it's not, check below.

2.6.1 `missing_message` key was removed

`missing_message` was removed. Use `[nitpick.files.present]` now.

Before:

```
[nitpick.files."pyproject.toml"]
missing_message = "Install poetry and run 'poetry init' to create it"
```

Now:

```
[nitpick.files.present]
"pyproject.toml" = "Install poetry and run 'poetry init' to create it"
```

CHAPTER

THREE

CONFIGURATION

The style file for your project should be configured in the `[tool.nitpick]` section of the configuration file.

Possible configuration files (in order of precedence):

1. `.nitpick.toml`
2. `pyproject.toml`

The first file found will be used; the other files will be ignored.

Run the `nipick init` CLI command to create a config file (*init: Initialise a configuration file*).

To configure your own style, you can either use `nitpick init`:

```
$ nitpick init /path/to/your-style-file.toml
```

or edit your configuration file and set the `style` option:

```
[tool.nitpick]
style = "/path/to/your-style-file.toml"
```

You can set `style` with any local file or URL.

3.1 Remote style

Use the URL of the remote file.

If it's hosted on GitHub, use any of the following formats:

GitHub URL scheme (`github://` or `gh://`) pinned to a specific version:

```
[tool.nitpick]
style = "github://andreoliwa/nitpick@v0.33.2/nitpick-style.toml"
# or
style = "gh://andreoliwa/nitpick@v0.33.2/nitpick-style.toml"
```

The `@` syntax is used to get a Git reference (commit, tag, branch). It is similar to the syntax used by `pip` and `pipx`:

- pip install - VCS Support - Git;
- pypa/pipx: Installing from Source Control.

If no Git reference is provided, the default GitHub branch will be used (for Nitpick, it's `develop`):

```toml
[tool.nitpick]
style = "github://andreoliwa/nitpick/nitpick-style.toml"
# or
style = "gh://andreoliwa/nitpick/nitpick-style.toml"

# It has the same effect as providing the default branch explicitly:
style = "github://andreoliwa/nitpick@develop/nitpick-style.toml"
# or
style = "gh://andreoliwa/nitpick@develop/nitpick-style.toml"
```

A regular GitHub URL also works. The corresponding raw URL will be used.

```toml
[tool.nitpick]
style = "https://github.com/andreoliwa/nitpick/blob/v0.33.2/nitpick-style.toml"
```

Or use the raw GitHub URL directly:

```toml
[tool.nitpick]
style = "https://raw.githubusercontent.com/andreoliwa/nitpick/v0.33.2/nitpick-style.toml"
```

You can also use the raw URL of a GitHub Gist:

```toml
[tool.nitpick]
style = "https://gist.githubusercontent.com/andreoliwa/f4fccf4e3e83a3228e8422c01a48be61/
↪raw/ff3447bddfc5a8665538ddf9c250734e7a38eabb/remote-style.toml"
```

If your style is on a private GitHub repo, you can provide the token directly on the URL. Or you can use an environment variable to avoid keeping secrets in plain text.

```toml
[tool.nitpick]
# A literal token
style = "github://p5iCG5AJuDgY@some-user/a-private-repo@some-branch/nitpick-style.toml"

# Or reading the secret value from the MY_AUTH_KEY env var
style = "github://$MY_AUTH_KEY@some-user/a-private-repo@some-branch/nitpick-style.toml"
```

Note: A literal token cannot start with a `$`. All tokens must not contain any `@` or `:` characters.

3.2 Style inside Python package

The style file can be fetched from an installed Python package.

Example of a use case: you create a custom flake8 extension and you also want to distribute a (versioned) Nitpick style bundled as a resource inside the Python package (check out this issue: Get style file from python package · Issue #202).

Python package URL scheme is `pypackage://` or `py://`:

```toml
[tool.nitpick]
style = "pypackage://some_python_package/styles/nitpick-style.toml"
# or
style = "py://some_python_package/styles/nitpick-style.toml"
```

Thanks to @isac322 for this feature.

3.3 Cache

Remote styles can be cached to avoid unnecessary HTTP requests. The cache can be configured with the `cache` key; see the examples below.

By default, remote styles will be cached for **one hour**. This default will also be used if the `cache` key has an invalid value.

3.3.1 Expiring after a predefined time

The cache can be set to expire after a defined time unit. Use the format `cache = "<integer> <time unit>"`. *Time unit* can be one of these (plural or singular, it doesn't matter):

- `minutes` / `minute`
- `hours` / `hour`
- `days` / `day`
- `weeks` / `week`

To cache for 15 minutes:

```
[tool.nitpick]
style = "https://example.com/remote-style.toml"
cache = "15 minutes"
```

To cache for 1 day:

```
[tool.nitpick]
style = "https://example.com/remote-style.toml"
cache = "1 day"
```

3.3.2 Forever

With this option, once the style(s) are cached, they never expire.

```
[tool.nitpick]
style = "https://example.com/remote-style.toml"
cache = "forever"
```

3.3.3 Never

With this option, the cache is never used. The remote style file(s) are always looked-up and a HTTP request is always executed.

```
[tool.nitpick]
style = "https://example.com/remote-style.toml"
cache = "never"
```

3.3.4 Clearing

The cache files live in a subdirectory of your project: `/path/to/your/project/.cache/nitpick/`. To clear the cache, simply remove this directory.

3.4 Local style

Using a file in your home directory:

```
[tool.nitpick]
style = "~/some/path/to/another-style.toml"
```

Using a relative path from another project in your hard drive:

```
[tool.nitpick]
style = "../another-project/another-style.toml"
```

3.5 Multiple styles

You can also use multiple styles and mix local files and URLs.

Example of usage: the `[tool.nitpick]` table on Nitpick's own pyproject.toml.

```
[tool.nitpick]
style = [
    "/path/to/first.toml",
    "/another/path/to/second.toml",
    "https://example.com/on/the/web/third.toml"
]
```

Note: The order is important: each style will override any keys that might be set by the previous `.toml` file.

If a key is defined in more than one file, the value from the last file will prevail.

3.6 Override a remote style

You can use a remote style as a starting point, and override settings on your local style file.

Use `./` to indicate the local style:

```
[tool.nitpick]
style = [
    "https://example.com/on/the/web/remote-style.toml",
    "./my-local-style.toml",
]
```

For Windows users: even though the path separator is a backslash, use the example above as-is. The "dot-slash" is a convention for Nitpick to know this is a local style file.

CHAPTER

FOUR

LIBRARY (PRESETS)

If you want to *configure your own style*, those are the some styles you can reuse.

Many TOML configs below are used in the default style file.

You can use these examples directly with their `py://` URL (see *Multiple styles*), or copy/paste the TOML into your own style file.

4.1 any

Style URL	Description
py://nitpick/resources/any/codeclimate	CodeClimate
py://nitpick/resources/any/commitizen	Commitizen (Python)
py://nitpick/resources/any/commitlint	commitlint
py://nitpick/resources/any/editorconfig	EditorConfig
py://nitpick/resources/any/git-legal	Git.legal - CodeClimate Community Edition
py://nitpick/resources/any/markdownlint	Markdown lint
py://nitpick/resources/any/pre-commit-hooks	pre-commit hooks for any project
py://nitpick/resources/any/prettier	Prettier

4.2 javascript

Style URL	Description
py://nitpick/resources/javascript/package-json	package.json

4.3 kotlin

Style URL	Description
py://nitpick/resources/kotlin/ktlint	ktlint

4.4 presets

Style URL	Description
py://nitpick/resources/presets/nitpick	Default style file for Nitpick

4.5 proto

Style URL	Description
py://nitpick/resources/proto/protolint	protolint (Protobuf linter)

4.6 python

Style URL	Description
py://nitpick/resources/python/310	Python 3.10
py://nitpick/resources/python/311	Python 3.11
py://nitpick/resources/python/37	Python 3.7
py://nitpick/resources/python/38	Python 3.8
py://nitpick/resources/python/39	Python 3.9
py://nitpick/resources/python/absent	Files that should not exist
py://nitpick/resources/python/autoflake	autoflake
py://nitpick/resources/python/bandit	Bandit
py://nitpick/resources/python/black	Black
py://nitpick/resources/python/flake8	Flake8
py://nitpick/resources/python/github-workflow	GitHub Workflow for Python
py://nitpick/resources/python/ipython	IPython
py://nitpick/resources/python/isort	isort
py://nitpick/resources/python/mypy	Mypy
py://nitpick/resources/python/poetry-editable	Poetry (editable projects; PEP 600 support)
py://nitpick/resources/python/poetry	Poetry
py://nitpick/resources/python/pre-commit-hooks	pre-commit hooks for Python projects
py://nitpick/resources/python/pylint	Pylint
py://nitpick/resources/python/radon	Radon
py://nitpick/resources/python/readthedocs	Read the Docs
py://nitpick/resources/python/sonar-python	SonarQube Python plugin
py://nitpick/resources/python/stable	Current stable Python version
py://nitpick/resources/python/tox	tox

4.7 shell

Style URL	Description
py://nitpick/resources/shell/bashate	bashate (code style for Bash)
py://nitpick/resources/shell/shellcheck	ShellCheck (static analysis for shell scripts)
py://nitpick/resources/shell/shfmt	shfmt (shell script formatter)

CHAPTER

FIVE

THE [NITPICK] SECTION

The [nitpick] section in *the style file* contains global settings for the style.

Those are settings that either don't belong to any specific config file, or can be applied to all config files.

5.1 Minimum version

Show an upgrade message to the developer if Nitpick's version is below `minimum_version`:

```
[nitpick]
minimum_version = "0.10.0"
```

5.2 [nitpick.files]

5.2.1 Files that should exist

To enforce that certain files should exist in the project, you can add them to the style file as a dictionary of "file name" and "extra message".

Use an empty string to not display any extra message.

```
[nitpick.files.present]
".editorconfig" = ""
"CHANGELOG.md" = "A project should have a changelog"
```

5.2.2 Files that should be deleted

To enforce that certain files should not exist in the project, you can add them to the style file.

```
[nitpick.files.absent]
"some_file.txt" = "This is an optional extra string to display after the warning"
"another_file.env" = ""
```

Multiple files can be configured as above. The message is optional.

5.2.3 Comma separated values

On `setup.cfg`, some keys are lists of multiple values separated by commas, like `flake8.ignore`.

On the style file, it's possible to indicate which key/value pairs should be treated as multiple values instead of an exact string. Multiple keys can be added.

```
[nitpick.files."setup.cfg"]
comma_separated_values = ["flake8.ignore", "isort.some_key", "another_section.another_key
↪"]
```

5.3 [nitpick.styles]

Styles can include other styles. Just provide a list of styles to include.

Example of usage: Nitpick's default style.

```
[nitpick.styles]
include = ["styles/python37", "styles/poetry"]
```

The styles will be merged following the sequence in the list. The `.toml` extension for each referenced file can be onitted.

Relative references are resolved relative to the URI of the style doument they are included in according to the normal rules of RFC 3986.

E.g. for a style file located at `gh://$GITHUB_TOKEN@foo_dev/bar_project@branchname/styles/foobar.toml` the following strings all reference the exact same canonical location to include:

```
[nitpick.styles]
include = [
  "foobar.toml",
  "../styles/foobar.toml",
  "/bar_project@branchname/styles/foobar.toml",
  "//$GITHUB_TOKEN@foo_dev/bar_project@branchname/styles/foobar.toml",
]
```

For style files on the local filesystem, the canonical path (after symbolic links have been resolved) of the style file is used as the base.

If a key/value pair appears in more than one sub-style, it will be overridden; the last declared key/pair will prevail.

CHAPTER

SIX

COMMAND-LINE INTERFACE

Note: The CLI is experimental, still under active development.

Nitpick has a CLI command to fix files automatically.

1. It doesn't work for all the plugins yet. Currently, it works for:
   - *INI files* (like `setup.cfg`, `tox.ini`, `.editorconfig`, `.pylintrc`, and any other `.ini`)
   - *TOML files*
2. It tries to preserve the comments and the formatting of the original file.
3. Some changes still have to be done manually; Nitpick cannot guess how to make certain changes automatically.
4. Run `nitpick fix` to modify files directly, or `nitpick check` to only display the violations.
5. The flake8 plugin only checks the files and doesn't make changes. This is the default for now; once the CLI becomes more stable, the "fix mode" will become the default.
6. The output format aims to follow pycodestyle (pep8) default output format.

If you use Git, you can review the files before committing.

The available commands are described below.

6.1 Main options

```
Usage: nitpick [OPTIONS] COMMAND [ARGS]...

  Enforce the same settings across multiple language-independent projects.

Options:
  -p, --project DIRECTORY  Path to project root
  --offline                Offline mode: no style will be downloaded (no HTTP
                           requests at all)
  --version                Show the version and exit.
  --help                   Show this message and exit.

Commands:
  check  Don't modify files, just print the differences.
  fix    Fix files, modifying them directly.
```

(continues on next page)

(continued from previous page)

```
  init   Create a [tool.nitpick] section in the configuration file if it...
  ls     List of files configured in the Nitpick style.
```

6.2 `fix`: Modify files directly

At the end of execution, this command displays:

- the number of fixed violations;
- the number of violations that have to be changed manually.

```
Usage: nitpick fix [OPTIONS] [FILES]...

  Fix files, modifying them directly.

  You can use partial and multiple file names in the FILES argument.

Options:
  -v, --verbose  Increase logging verbosity (-v = INFO, -vv = DEBUG)
  --help         Show this message and exit.
```

6.3 `check`: Don't modify, just print the differences

```
Usage: nitpick check [OPTIONS] [FILES]...

  Don't modify files, just print the differences.

  Return code 0 means nothing would change. Return code 1 means some files
  would be modified. You can use partial and multiple file names in the FILES
  argument.

Options:
  -v, --verbose  Increase logging verbosity (-v = INFO, -vv = DEBUG)
  --help         Show this message and exit.
```

6.4 `ls`: List configures files

```
Usage: nitpick ls [OPTIONS] [FILES]...

  List of files configured in the Nitpick style.

  Display existing files in green and absent files in red. You can use partial
  and multiple file names in the FILES argument.

Options:
  --help  Show this message and exit.
```

6.5 `init`: Initialise a configuration file

```
Usage: nitpick init [OPTIONS] [STYLE_URLS]...

  Create a [tool.nitpick] section in the configuration file if it doesn't
  exist already.

Options:
  --help  Show this message and exit.
```

Note: Try running Nitpick with the new *Command-line interface* instead of a flake8 plugin.

In the future, there are plans to make flake8 an optional dependency.

CHAPTER

SEVEN

FLAKE8 PLUGIN

Nitpick is not a proper flake8 plugin; it piggybacks on flake8's messaging system though.

Flake8 lints Python files; Nitpick "lints" configuration (text) files instead.

To act like a flake8 plugin, Nitpick does the following:

1. Find *any Python file in your project*;
2. Use the first Python file found and ignore other Python files in the project.
3. Check the style file and compare with the configuration/text files.
4. Report violations on behalf of that Python file, and not on the configuration file that's actually wrong.

So, if you have a violation on `setup.cfg`, it will be reported like this:

```
./tasks.py:0:1: NIP323 File setup.cfg: [flake8]max-line-length is 80 but it should be␣
↪like this:
[flake8]
max-line-length = 120
```

Notice the `tasks.py` at the beginning of the line, and not `setup.cfg`.

Note: To run Nitpick as a Flake8 plugin, the project must have *at least one* Python file*.

If your project is not a Python project, creating a `dummy.py` file on the root of the project is enough.

7.1 Pre-commit hook and flake8

Currently, the default pre-commit hook uses flake8 in an unconventional and not recommended way.

It calls flake8 directly:

```
flake8 --select=NIP
```

This current default pre-commit hook (called `nitpick`) is a placeholder for the future, when flake8 will be only an optional dependency.

7.1.1 Why `always_run: true`?

This in intentional, because *Nitpick is not a conventional flake8 plugin*.

Since flake8 only lints Python files, the pre-commit hook will only run when a Python file is modified. It won't run when a config/text changes.

An example: suppose you're using a remote Nitpick style (like the style from WeMake).

At the moment, their style currently checks `setup.cfg` only.

Suppose they change or add an option on their isort.toml file.

If the nitpick pre-commit hook had `always_run: false` and `pass_filenames: true`, your local `setup.cfg` would only be verified:

1. If a Python file was changed.
2. If you ran `pre-commit run --all-files`.

So basically the pre-commit hook would be useless to guarantee that your config files would always match the remote style… which is precisely the purpose of Nitpick.

Note: To avoid this, use the other pre-commit hooks, the ones that call the Nitpick CLI directly instead of running `flake8`.

7.2 Root dir of the project

You should run Nitpick in the *root dir* of your project.

A directory is considered a root dir if it contains one of the following files:

- `.pre-commit-config.yaml` (pre-commit)
- `pyproject.toml`
- `setup.py`
- `setup.cfg`
- `requirements*.txt`
- `Pipfile` (Pipenv)
- `tox.ini` (tox)
- `package.json` (JavaScript, NodeJS)
- `Cargo.*` (Rust)
- `go.mod`, `go.sum` (Golang)
- `app.py` and `wsgi.py` (Flask CLI)
- `autoapp.py` (Flask)
- `manage.py` (Django)

7.3 Main Python file

On the *root dir of the project* Nitpick searches for a main Python file. Every project must have at least one `*.py` file, otherwise flake8 won't even work.

Those are the Python files that are considered:

- `setup.py`
- `app.py` and `wsgi.py` (Flask CLI)
- `autoapp.py`
- `manage.py`
- any `*.py` file

CHAPTER

EIGHT

PLUGINS

Nitpick uses plugins to handle configuration files.

There are plans to add plugins that handle certain file types, specific files, and user plugins. Check the roadmap.

Below are the currently included plugins.

8.1 INI files

Enforce configurations and autofix INI files.

Examples of `.ini` files handled by this plugin:

- setup.cfg
- .editorconfig
- tox.ini
- .pylintrc

Style examples enforcing values on INI files: flake8 configuration.

8.2 JSON files

Enforce configurations and autofix JSON files.

Add the configurations for the file name you wish to check. Style example: the default config for package.json.

8.3 Text files

Enforce configuration on text files.

To check if `some.txt` file contains the lines `abc` and `def` (in any order):

```
[["some.txt".contains]]
line = "abc"

[["some.txt".contains]]
line = "def"
```

8.4 TOML files

Enforce configurations and autofix TOML files.

E.g.: pyproject.toml (PEP 518).

See also the [tool.poetry] section of the pyproject.toml file.

Style example: Python 3.8 version constraint. There are *many other examples here*.

8.5 YAML files

Enforce configurations and autofix YAML files.

- Example: .pre-commit-config.yaml.
- Style example: the default pre-commit hooks.

Warning: The plugin tries to preserve comments in the YAML file by using the `ruamel.yaml` package. It works for most cases. If your comment was removed, place them in a different place of the fil and try again. If it still doesn't work, please report a bug.

Known issue: lists like `args` and `additional_dependencies` might be joined in a single line, and comments between items will be removed. Move your comments outside these lists, and they should be preserved.

Note: No validation of `.pre-commit-config.yaml` will be done anymore in this generic YAML plugin. Nitpick will not validate hooks and missing keys as it did before; it's not the purpose of this package.

CHAPTER

NINE

TROUBLESHOOTING

9.1 Crash on multi-threading

On macOS, flake8 might raise this error when calling `requests.get(url)`:

```
objc[93329]: +[__NSPlaceholderDate initialize] may have been in progress in another␣
↪thread when fork() was called.
objc[93329]: +[__NSPlaceholderDate initialize] may have been in progress in another␣
↪thread when fork() was called. We cannot safely call it or ignore it in the fork()␣
↪child process. Crashing instead. Set a breakpoint on objc_initializeAfterForkError to␣
↪debug.
```

To solve this issue, add this environment variable to `.bashrc` (or the initialization file for your favorite shell):

```
export OBJC_DISABLE_INITIALIZE_FORK_SAFETY=YES
```

Thanks to this StackOverflow answer.

9.2 ModuleNotFoundError: No module named 'nitpick.plugins.XXX'

When upgrading to new versions, old plugins might be renamed in setuptools entry points.

But they might still be present in the entry_points.txt plugin metadata in your virtualenv.

```
$ rg nitpick.plugins.setup ~/Library/Caches/pypoetry/
/Users/john.doe/Library/Caches/pypoetry/virtualenvs/nitpick-UU_pZ5zs-py3.7/lib/python3.7/
↪site-packages/nitpick-0.24.1.dist-info/entry_points.txt
11:setup_cfg=nitpick.plugins.setup_cfg
```

Remove and recreate the virtualenv; this should fix it.

During development, you can run `invoke clean --venv install --dry`. It will display the commands that would be executed; remove `--dry` to actually run them.

Read this page on how to install Invoke.

9.3 Executable `.tox/lint/bin/pylint` not found

You might get this error while running `make` locally.

1. Run `invoke lint` (or `tox -e lint` directly) to create this tox environment.
2. Run `make` again.

CHAPTER

TEN

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

Check the projects on GitHub, you might help coding a planned feature.

10.1 Bug reports or feature requests

- First, search the GitHub issue tracker to see if your bug/feature is already there.
- If nothing is found, just add a new issue and follow the instructions.

10.2 Documentation improvements

Nitpick could always use more documentation, whether as part of the official docs, in docstrings, or even on the web in blog posts, articles, and such.

10.3 Development

To set up Nitpick for local development:

1. Fork Nitpick (look for the "Fork" button).
2. Clone your fork locally:

   ```
   cd ~/Code
   git clone git@github.com:your_name_here/nitpick.git
   cd nitpick
   ```

3. Install Poetry globally using the recommended way.
4. Install Invoke. You can use pipx to install it globally: `pipx install invoke`.
5. Install dependencies and pre-commit hooks:

   ```
   invoke install --hooks
   ```

6. Create a branch for local development:

   ```
   git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

7. When you're done making changes, run tests and checks locally with:

```
# Quick tests and checks
make
# Or use this to simulate a full CI build with tox
invoke ci-build
```

8. Commit your changes and push your branch to GitHub:

```
git add .

# For a feature:
git commit -m "feat: short description of your feature"
# For a bug fix:
git commit -m "fix: short description of what you fixed"

git push origin name-of-your-bugfix-or-feature
```

9. Submit a pull request through the GitHub website.

10.3.1 Commit convention

Nitpick follows Conventional Commits

No need to rebase the commits in your branch. If your pull request is accepted, all your commits will be squashed into a single one, and the commit message will be adjusted to follow the current standard.

10.3.2 Pull Request Guidelines

If you need some code review or feedback while you're developing the code, just make a draft pull request.

For merging, follow the checklist on the pull request template itself.

When running `invoke test`: if you don't have all the necessary Python versions available locally (needed by tox), you can rely on GitHub Workflows. Tests will run for each change you add in the pull request. It will be slower though...

CHAPTER

ELEVEN

AUTHORS

- W. Augusto Andreoli <andreoliwa@gmail.com>

CHAPTER

TWELVE

NITPICK

12.1 nitpick package

Main module.

class nitpick.**Nitpick**

Bases: object

The Nitpick API.

configured_files(**partial_names: str*) → List[Path]

List of files configured in the Nitpick style. Filter only the selected partial names.

echo(*message: str*)

Echo a message on the terminal, with the relative path at the beginning.

enforce_present_absent(**partial_names: str*) → Iterator[*Fuss*]

Enforce files that should be present or absent.

Parameters
partial_names – Names of the files to enforce configs for.

Returns
Fuss generator.

enforce_style(**partial_names: str*, *autofix=True*) → Iterator[*Fuss*]

Read the merged style and enforce the rules in it.

1. Get all root keys from the merged style (every key is a filename, except "nitpick").
2. For each file name, find the plugin(s) that can handle the file.

Parameters

- **partial_names** – Names of the files to enforce configs for.
- **autofix** – Flag to modify files, if the plugin supports it (default: True).

Returns
Fuss generator.

init(*project_root: Path | str | None = None*, *offline: bool | None = None*) → *Nitpick*

Initialize attributes of the singleton.

project: *Project*

run(**partial_names: str*, *autofix=False*) → Iterator[*Fuss*]

Run Nitpick.

Parameters

- **partial_names** – Names of the files to enforce configs for.
- **autofix** – Flag to modify files, if the plugin supports it (default: True).

Returns

Fuss generator.

classmethod singleton() → *Nitpick*

Return a single instance of the class.

12.1.1 Subpackages

nitpick.plugins package

Hook specifications used by Nitpick plugins.

Note: The hook specifications and the plugin classes are still experimental and considered as an internal API. They might change at any time; use at your own risk.

Submodules

nitpick.plugins.base module

Base class for file checkers.

class nitpick.plugins.base.**NitpickPlugin**(*info: FileInfo*, *expected_config: JsonDict*, *autofix=False*)

Bases: object

Base class for Nitpick plugins.

Parameters

- **data** – File information (project, path, tags).
- **expected_config** – Expected configuration for the file
- **autofix** – Flag to modify files, if the plugin supports it (default: True).

abstract enforce_rules() → Iterator[*Fuss*]

Enforce rules for this file. It must be overridden by inherited classes if needed.

entry_point() → Iterator[*Fuss*]

Entry point of the Nitpick plugin.

filename = ''

fixable: bool = False

Can this plugin modify its files directly? Are the files fixable?

identify_tags: set[str] = {}

Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

abstract property initial_contents: str

Suggested initial content when the file doesn't exist.

property nitpick_file_dict: Dict[str, Any]

Nitpick configuration for this file as a TOML dict, taken from the style file.

post_init()

Hook for plugin initialization after the instance was created.

The name mimics `__post_init__()` on dataclasses, without the magic double underscores: Post-init processing

predefined_special_config() → *SpecialConfig*

Create a predefined special configuration for this plugin. Each plugin can override this method.

skip_empty_suggestion = False

validation_schema: Schema | None = None

Nested validation field for this file, to be applied in runtime when the validation schema is rebuilt. Useful when you have a strict configuration for a file type (e.g. *nitpick.plugins.json.JsonPlugin*).

violation_base_code: int = 0

write_file(*file_exists: bool*) → *Fuss* | None

Hook to write the new file when autofix mode is on. Should be used by inherited classes.

write_initial_contents(*doc_class: type[*BaseDoc*]*, *expected_dict: dict* | *None = None*) → str

Helper to write initial contents based on a format.

nitpick.plugins.info module

Info needed by the plugins.

class nitpick.plugins.info.**FileInfo**(*project: ~nitpick.project.Project*, *path_from_root: str*, *tags: ~typing.Set[str] = <factory>*)

Bases: object

File information needed by the plugin.

classmethod create(*project:* Project, *path_from_root: str*) → *FileInfo*

Clean the file name and get its tags.

path_from_root: str

project: *Project*

tags: Set[str]

nitpick.plugins.ini module

INI files.

class nitpick.plugins.ini.**IniPlugin**(*info:* FileInfo, *expected_config: JsonDict*, *autofix=False*)

Bases: *NitpickPlugin*

Enforce configurations and autofix INI files.

Examples of `.ini` files handled by this plugin:

- setup.cfg
- .editorconfig
- tox.ini
- .pylintrc

Style examples enforcing values on INI files: flake8 configuration.

add_options_before_space(*section: str*, *options: dict*) → None

Add new options before a blank line in the end of the section.

comma_separated_values: set[str]

compare_different_keys(*section*, *key*, *raw_actual: Any*, *raw_expected: Any*) → Iterator[*Fuss*]

Compare different keys, with special treatment when they are lists or numeric.

static contents_without_top_section(*multiline_text: str*) → str

Remove the temporary top section from multiline text, and keep the newline at the end of the file.

property current_sections: set[str]

Current sections of the .ini file, including updated sections.

dirty: bool

enforce_comma_separated_values(*section*, *key*, *raw_actual: Any*, *raw_expected: Any*) → Iterator[*Fuss*]

Enforce sections and keys with comma-separated values. The values might contain spaces.

enforce_missing_sections() → Iterator[*Fuss*]

Enforce missing sections.

enforce_rules() → Iterator[*Fuss*]

Enforce rules on missing sections and missing key/value pairs in an INI file.

enforce_section(*section: str*) → Iterator[*Fuss*]

Enforce rules for a section.

entry_point() → Iterator[*Fuss*]

Entry point of the Nitpick plugin.

expected_config: JsonDict

property expected_sections: set[str]

Expected sections (from the style config).

file_path: Path

filename = ''

fixable: bool = True

Can this plugin modify its files directly? Are the files fixable?

static get_example_cfg(*parser: ConfigParser*) → str

Print an example of a config parser in a string instead of a file.

get_missing_output() → str

Get a missing output string example from the missing sections in an INI file.

identify_tags: set[str] = {'editorconfig', 'ini'}

Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

property initial_contents: str

Suggest the initial content for this missing file.

property missing_sections: set[str]

Missing sections.

property needs_top_section: bool

Return True if this .ini file needs a top section (e.g.: .editorconfig).

property nitpick_file_dict: Dict[str, Any]

Nitpick configuration for this file as a TOML dict, taken from the style file.

post_init()

Post initialization after the instance was created.

predefined_special_config() → *SpecialConfig*

Create a predefined special configuration for this plugin. Each plugin can override this method.

show_missing_keys(*section: str*, *values: list[tuple[str, Any]]*) → Iterator[*Fuss*]

Show the keys that are not present in a section.

skip_empty_suggestion = False

updater: ConfigUpdater

validation_schema: Schema | None = None

Nested validation field for this file, to be applied in runtime when the validation schema is rebuilt. Useful when you have a strict configuration for a file type (e.g. *nitpick.plugins.json.JsonPlugin*).

violation_base_code: int = 320

write_file(*file_exists: bool*) → *Fuss* | None

Write the new file.

write_initial_contents(*doc_class: type[*BaseDoc*]*, *expected_dict: dict | None = None*) → str

Helper to write initial contents based on a format.

class nitpick.plugins.ini.**Violations**(*value*)

Bases: *ViolationEnum*

Violations for this plugin.

INVALID_COMMA_SEPARATED_VALUES_SECTION = (325, ': invalid sections on comma_separated_values:')

MISSING_OPTION = (324, ': section [{section}] has some missing key/value pairs. Use this:')

MISSING_SECTIONS = (321, ' has some missing sections. Use this:')

MISSING_VALUES_IN_LIST = (322, ' has missing values in the {key!r} key. Include those values:')

OPTION_HAS_DIFFERENT_VALUE = (323, ': [{section}]{key} is {actual} but it should be like this:')

PARSING_ERROR = (326, ': parsing error ({cls}): {msg}')

TOP_SECTION_HAS_DIFFERENT_VALUE = (327, ': {key} is {actual} but it should be:')

TOP_SECTION_MISSING_OPTION = (328, ': top section has missing options. Use this:')

nitpick.plugins.ini.**can_handle**(*info:* FileInfo) → type[*NitpickPlugin*] | None

Handle INI files.

nitpick.plugins.ini.**plugin_class**() → type[*NitpickPlugin*]

Handle INI files.

nitpick.plugins.json module

JSON files.

class nitpick.plugins.json.**JsonFileSchema**(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: *BaseNitpickSchema*

Validation schema for any JSON file added to the style.

class Meta

Bases: object

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **`additional`: Tuple or list of fields to include *in addition* to the** explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **`include`: Dictionary of additional fields to include in the schema. It is** usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **`exclude`: Tuple or list of fields to exclude in the serialized result.** Nested fields can be represented with dot delimiters.

- `dateformat`: Default format for *Date* <*fields.Date*> fields.
- `datetimeformat`: Default format for *DateTime* <*fields.DateTime*> fields.
- `timeformat`: Default format for *Time* <*fields.Time*> fields.
- **`render_module`: Module to use for *loads* <*Schema.loads*> and *dumps* <*Schema.dumps*>.**
  Defaults to *json* from the standard library.
- **`ordered`: If *True*, order serialization output according to the**
  order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **`index_errors`: If *True*, errors dictionaries will include the index**
  of invalid items in a collection.
- `load_only`: Tuple or list of fields to exclude from serialized results.
- `dump_only`: Tuple or list of fields to exclude from deserialization
- **`unknown`: Whether to exclude, include, or raise an error for unknown**
  fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **`register`: Whether to register the *Schema* with marshmallow's internal**
  class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

OPTIONS_CLASS

alias of `SchemaOpts`

TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}

property dict_class: type

dump(*obj: Any*, *, *many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/nitpick_section.html.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, ***, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, ***, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

loads(*json_data: str*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: SchemaOpts = <marshmallow.schema.SchemaOpts object>

property set_class: type

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns
A dictionary of validation errors.

New in version 1.1.0.

class nitpick.plugins.json.**JsonPlugin**(*info:* FileInfo, *expected_config: JsonDict*, *autofix=False*)

Bases: *NitpickPlugin*

Enforce configurations and autofix JSON files.

Add the configurations for the file name you wish to check. Style example: the default config for package.json.

dirty: bool

enforce_rules() → Iterator[*Fuss*]

Enforce rules for missing keys and JSON content.

entry_point() → Iterator[*Fuss*]

Entry point of the Nitpick plugin.

expected_config: JsonDict

expected_dict_from_contains_json()

Expected dict created from "contains_json" values.

expected_dict_from_contains_keys()

Expected dict created from "contains_keys" values.

file_path: Path

filename = ''

fixable: bool = True

Can this plugin modify its files directly? Are the files fixable?

identify_tags: set[str] = {'json'}

Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

property initial_contents: str

Suggest the initial content for this missing file.

property nitpick_file_dict: Dict[str, Any]

Nitpick configuration for this file as a TOML dict, taken from the style file.

post_init()

Hook for plugin initialization after the instance was created.

The name mimics `__post_init__()` on dataclasses, without the magic double underscores: Post-init processing

predefined_special_config() → *SpecialConfig*

Create a predefined special configuration for this plugin. Each plugin can override this method.

report(*violation:* ViolationEnum, *blender: JsonDict*, *change:* BaseDoc | *None*)

Report a violation while optionally modifying the JSON dict.

skip_empty_suggestion = False

validation_schema

alias of *JsonFileSchema*

violation_base_code: int = 340

write_file(*file_exists: bool*) → *Fuss* | None

Hook to write the new file when autofix mode is on. Should be used by inherited classes.

write_initial_contents(*doc_class: type[*BaseDoc*]*, *expected_dict: dict* | *None* = *None*) → str

Helper to write initial contents based on a format.

nitpick.plugins.json.**can_handle**(*info:* FileInfo) → type[*NitpickPlugin*] | None

Handle JSON files.

nitpick.plugins.json.**plugin_class**() → type[*NitpickPlugin*]

Handle JSON files.

nitpick.plugins.text module

Text files.

class nitpick.plugins.text.**TextItemSchema**(*, *only: types.StrSequenceOrSet* | *None* = *None*, *exclude: types.StrSequenceOrSet* = *()*, *many: bool* = *False*, *context: dict* | *None* = *None*, *load_only: types.StrSequenceOrSet* = *()*, *dump_only: types.StrSequenceOrSet* = *()*, *partial: bool* | *types.StrSequenceOrSet* = *False*, *unknown: str* | *None* = *None*)

Bases: Schema

Validation schema for the object inside `contains`.

class Meta

Bases: object

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **`additional`: Tuple or list of fields to include *in addition* to the** explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **`include`: Dictionary of additional fields to include in the schema. It is** usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **`exclude`: Tuple or list of fields to exclude in the serialized result.** Nested fields can be represented with dot delimiters.
- `dateformat`: Default format for *Date <fields.Date>* fields.
- `datetimeformat`: Default format for *DateTime <fields.DateTime>* fields.
- `timeformat`: Default format for *Time <fields.Time>* fields.
- **`render_module`: Module to use for *loads <Schema.loads>* and *dumps <Schema.dumps>*.** Defaults to *json* from the standard library.
- **`ordered`: If *True*, order serialization output according to the** order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **`index_errors`: If *True*, errors dictionaries will include the index** of invalid items in a collection.
- `load_only`: Tuple or list of fields to exclude from serialized results.
- `dump_only`: Tuple or list of fields to exclude from deserialization
- **`unknown`: Whether to exclude, include, or raise an error for unknown** fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **`register`: Whether to register the *Schema* with marshmallow's internal** class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

OPTIONS_CLASS

alias of SchemaOpts

TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}

property dict_class: type

dump(*obj: Any*, *, *many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.

- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/plugins.html#text-files.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, ***, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, *, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, *, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

loads(*json_data: str*, *, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: SchemaOpts = <marshmallow.schema.SchemaOpts object>

property set_class: type

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns

A dictionary of validation errors.

New in version 1.1.0.

class nitpick.plugins.text.**TextPlugin**(*info:* FileInfo, *expected_config: JsonDict*, *autofix=False*)

Bases: *NitpickPlugin*

Enforce configuration on text files.

To check if `some.txt` file contains the lines `abc` and `def` (in any order):

```
[["some.txt".contains]]
line = "abc"

[["some.txt".contains]]
line = "def"
```

dirty: bool

enforce_rules() → Iterator[*Fuss*]

Enforce rules for missing lines.

entry_point() → Iterator[*Fuss*]

Entry point of the Nitpick plugin.

expected_config: JsonDict

file_path: Path

filename = ''

fixable: bool = False

Can this plugin modify its files directly? Are the files fixable?

identify_tags: set[str] = {'text'}

Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

property initial_contents: str

Suggest the initial content for this missing file.

property nitpick_file_dict: Dict[str, Any]

Nitpick configuration for this file as a TOML dict, taken from the style file.

post_init()

Hook for plugin initialization after the instance was created.

The name mimics `__post_init__()` on dataclasses, without the magic double underscores: Post-init processing

predefined_special_config() → *SpecialConfig*

Create a predefined special configuration for this plugin. Each plugin can override this method.

skip_empty_suggestion = True

validation_schema

alias of *TextSchema*

violation_base_code: int = 350

write_file(*file_exists: bool*) → *Fuss* | None

Hook to write the new file when autofix mode is on. Should be used by inherited classes.

write_initial_contents(*doc_class: type[BaseDoc]*, *expected_dict: dict | None = None*) → str

Helper to write initial contents based on a format.

class nitpick.plugins.text.**TextSchema**(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: Schema

Validation schema for the text file TOML configuration.

class Meta

Bases: object

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **`additional`: Tuple or list of fields to include *in addition* to the** explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **`include`: Dictionary of additional fields to include in the schema. It is** usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **`exclude`: Tuple or list of fields to exclude in the serialized result.** Nested fields can be represented with dot delimiters.

- `dateformat`: Default format for *Date* <*fields.Date*> fields.
- `datetimeformat`: Default format for *DateTime* <*fields.DateTime*> fields.
- `timeformat`: Default format for *Time* <*fields.Time*> fields.
- **`render_module`: Module to use for *loads* <*Schema.loads*> and *dumps* <*Schema.dumps*>.**
  Defaults to *json* from the standard library.
- **`ordered`: If *True*, order serialization output according to the**
  order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **`index_errors`: If *True*, errors dictionaries will include the index**
  of invalid items in a collection.
- `load_only`: Tuple or list of fields to exclude from serialized results.
- `dump_only`: Tuple or list of fields to exclude from deserialization
- **`unknown`: Whether to exclude, include, or raise an error for unknown**
  fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **`register`: Whether to register the *Schema* with marshmallow's internal**
  class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

OPTIONS_CLASS

alias of `SchemaOpts`

TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}

property dict_class: type

dump(*obj: Any*, **, many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/plugins.html#text-files.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, ***, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, ***, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

loads(*json_data: str*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: SchemaOpts = <marshmallow.schema.SchemaOpts object>

property set_class: type

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns
A dictionary of validation errors.

New in version 1.1.0.

class nitpick.plugins.text.**Violations**(*value*)

Bases: *ViolationEnum*

Violations for this plugin.

MISSING_LINES = (352, ' has missing lines:')

nitpick.plugins.text.**can_handle**(*info:* FileInfo) → type[*NitpickPlugin*] | None

Handle text files.

nitpick.plugins.text.**plugin_class**() → type[*NitpickPlugin*]

Handle text files.

nitpick.plugins.toml module

TOML files.

class nitpick.plugins.toml.**TomlPlugin**(*info:* FileInfo, *expected_config: JsonDict*, *autofix=False*)

Bases: *NitpickPlugin*

Enforce configurations and autofix TOML files.

E.g.: pyproject.toml (PEP 518).

See also the [tool.poetry] section of the pyproject.toml file.

Style example: Python 3.8 version constraint. There are *many other examples here*.

dirty: bool

enforce_rules() → Iterator[*Fuss*]

Enforce rules for missing key/value pairs in the TOML file.

entry_point() → Iterator[*Fuss*]

Entry point of the Nitpick plugin.

expected_config: JsonDict

file_path: Path

filename = ''

fixable: bool = True

Can this plugin modify its files directly? Are the files fixable?

identify_tags: set[str] = {'toml'}

Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

property initial_contents: str

Suggest the initial content for this missing file.

property nitpick_file_dict: Dict[str, Any]

Nitpick configuration for this file as a TOML dict, taken from the style file.

post_init()

Hook for plugin initialization after the instance was created.

The name mimics `__post_init__()` on dataclasses, without the magic double underscores: Post-init processing

predefined_special_config() → *SpecialConfig*

Create a predefined special configuration for this plugin. Each plugin can override this method.

report(*violation:* ViolationEnum, *document: TOMLDocument* | *None*, *change:* TomlDoc | *None*, *replacement:* TomlDoc | *None* = *None*)

Report a violation while optionally modifying the TOML document.

skip_empty_suggestion = False

validation_schema: Schema | None = None

Nested validation field for this file, to be applied in runtime when the validation schema is rebuilt. Useful when you have a strict configuration for a file type (e.g. *nitpick.plugins.json.JsonPlugin*).

violation_base_code: int = 310

write_file(*file_exists: bool*) → *Fuss* | None

Hook to write the new file when autofix mode is on. Should be used by inherited classes.

write_initial_contents(*doc_class: type[*BaseDoc*]*, *expected_dict: dict* | *None* = *None*) → str

Helper to write initial contents based on a format.

nitpick.plugins.toml.**can_handle**(*info:* FileInfo) → type[*NitpickPlugin*] | None

Handle TOML files.

nitpick.plugins.toml.**plugin_class**() → type[*NitpickPlugin*]

Handle TOML files.

nitpick.plugins.yaml module

YAML files.

class nitpick.plugins.yaml.**YamlPlugin**(*info:* FileInfo, *expected_config: JsonDict*, *autofix=False*)

Bases: *NitpickPlugin*

Enforce configurations and autofix YAML files.

- Example: .pre-commit-config.yaml.
- Style example: the default pre-commit hooks.

Warning: The plugin tries to preserve comments in the YAML file by using the `ruamel.yaml` package. It works for most cases. If your comment was removed, place them in a different place of the fil and try again. If it still doesn't work, please report a bug.

Known issue: lists like `args` and `additional_dependencies` might be joined in a single line, and comments between items will be removed. Move your comments outside these lists, and they should be preserved.

Note: No validation of `.pre-commit-config.yaml` will be done anymore in this generic YAML plugin. Nitpick will not validate hooks and missing keys as it did before; it's not the purpose of this package.

dirty: `bool`

enforce_rules() → Iterator[*Fuss*]

Enforce rules for missing data in the YAML file.

entry_point() → Iterator[*Fuss*]

Entry point of the Nitpick plugin.

expected_config: `JsonDict`

file_path: `Path`

filename = ''

fixable: `bool` = True

Can this plugin modify its files directly? Are the files fixable?

identify_tags: `set[str]` = {'yaml'}

Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

property initial_contents: `str`

Suggest the initial content for this missing file.

property nitpick_file_dict: `Dict[str, Any]`

Nitpick configuration for this file as a TOML dict, taken from the style file.

post_init()

Hook for plugin initialization after the instance was created.

The name mimics `__post_init__()` on dataclasses, without the magic double underscores: Post-init processing

predefined_special_config() → *SpecialConfig*

Predefined special config, with list keys for .pre-commit-config.yaml and GitHub Workflow files.

report(*violation:* ViolationEnum, *yaml_object: YamlObject*, *change:* YamlDoc | *None*, *replacement:* YamlDoc | *None* = *None*)

Report a violation while optionally modifying the YAML document.

skip_empty_suggestion = False

validation_schema: `Schema | None` = None

Nested validation field for this file, to be applied in runtime when the validation schema is rebuilt. Useful when you have a strict configuration for a file type (e.g. *nitpick.plugins.json.JsonPlugin*).

violation_base_code: int = 360

write_file(*file_exists: bool*) → *Fuss* | None

Hook to write the new file when autofix mode is on. Should be used by inherited classes.

write_initial_contents(*doc_class: type[*BaseDoc*]*, *expected_dict: dict* | *None* = *None*) → str

Helper to write initial contents based on a format.

nitpick.plugins.yaml.**can_handle**(*info:* FileInfo) → type[*NitpickPlugin*] | None

Handle YAML files.

nitpick.plugins.yaml.**plugin_class**() → type[*NitpickPlugin*]

Handle YAML files.

nitpick.resources package

A library of Nitpick styles. Building blocks that can be combined and reused.

Subpackages

nitpick.resources.any package

Styles for any language.

nitpick.resources.javascript package

Styles for JavaScript.

nitpick.resources.kotlin package

Styles for Kotlin.

nitpick.resources.presets package

Style presets.

nitpick.resources.proto package

Styles for Protobuf files.

nitpick.resources.python package

Styles for Python.

nitpick.resources.shell package

Styles for shell scripts.

nitpick.style package

Styles parsing and merging.

class nitpick.style.**StyleManager**(*project:* Project, *offline:* *bool*, *cache_option:* *str*)

Bases: object

Include styles recursively from one another.

property cache_dir: Path

Clear the cache directory (on the project root or on the current directory).

cache_option: str

static file_field_pair(*filename:* *str*, *base_file_class:* *type[*NitpickPlugin*]*) → dict[str, *fields.Field*]

Return a schema field with info from a config file class.

find_initial_styles(*configured_styles: Sequence[str]*, *base:* *str* | *None* = *None*) → Iterator[*Fuss*]

Find the initial style(s) and include them.

base is the URL for the source of the initial styles, and is used to resolve relative references. If omitted, defaults to the project root.

static get_default_style_url(*github=False*) → furl

Return the URL of the default style/preset.

include_multiple_styles(*chosen_styles: Iterable[furl]*) → Iterator[*Fuss*]

Include a list of styles (or just one) into this style tree.

load_fixed_name_plugins() → Set[Type[*NitpickPlugin*]]

Separate classes with fixed file names from classes with dynamic files names.

merge_toml_dict() → JsonDict

Merge all included styles into a TOML (actually JSON) dictionary.

offline: bool

project: *Project*

rebuild_dynamic_schema() → None

Rebuild the dynamic Marshmallow schema when needed, adding new fields that were found on the style.

nitpick.style.**parse_cache_option**(*cache_option:* *str*) → tuple[*CachingEnum*, timedelta | int]

Parse the cache option provided on pyproject.toml.

If no cache if provided or is invalid, the default is *one hour*.

Subpackages

nitpick.style.fetchers package

Style fetchers with protocol support.

class nitpick.style.fetchers.**Scheme**(*value*)

Bases: LowercaseStrEnum

URL schemes.

FILE = 'file'

GH = 'gh'

GITHUB = 'github'

HTTP = 'http'

HTTPS = 'https'

PY = 'py'

PYPACKAGE = 'pypackage'

class nitpick.style.fetchers.**StyleFetcherManager**(*offline: bool*, *cache_dir: Path*, *cache_option: str*)

Bases: object

Manager that controls which fetcher to be used given a protocol.

cache_dir: Path

cache_option: str

fetch(*url: furl*) → str | None

Determine which fetcher to be used and fetch from it.

Returns None when offline is True and the fetcher would otherwise require a connection.

fetchers: dict[str, *StyleFetcher*]

normalize_url(*url: str | furl*, *base: furl*) → furl

Normalize a style URL.

The URL is made absolute against base, then passed to individual fetchers to produce a canonical version of the URL.

offline: bool

schemes: tuple[str]

session: CachedSession

Submodules

nitpick.style.fetchers.base module

Base class for fetchers that wrap inner fetchers with caching ability.

class nitpick.style.fetchers.base.**StyleFetcher**(*session: CachedSession | None = None*, *protocols: tuple[str, ...] = ()*, *domains: tuple[str, ...] = ()*)

Bases: object

Base class of all fetchers, it encapsulates get/fetch from a specific source.

domains: tuple[str, ...] = ()

fetch(*url: furl*) → str

Fetch a style from a specific fetcher.

normalize(*url: furl*) → furl

Normalize a URL.

Produces a canonical URL, meant to be used to uniquely identify a style resource.

- The base name has .toml appended if not already ending in that extension
- Individual fetchers can further normalize the path and scheme.

preprocess_relative_url(*url: str*) → str

Preprocess a relative URL.

Only called for urls that lack a scheme (at the very least), being resolved against a base URL that matches this specific fetcher.

protocols: tuple[str, ...] = ()

requires_connection: ClassVar[bool] = False

session: CachedSession | None = None

nitpick.style.fetchers.file module

Basic local file fetcher.

class nitpick.style.fetchers.file.**FileFetcher**(*session: CachedSession | None = None*, *protocols: tuple[str, ...] = (<Scheme.FILE: 'file'>,)*, *domains: tuple[str, ...] = ()*)

Bases: *StyleFetcher*

Fetch a style from a local file.

domains: tuple[str, ...] = ()

fetch(*url: furl*) → str

Fetch a style from a local file.

normalize(*url: furl*) → furl

Normalize a URL.

Produces a canonical URL, meant to be used to uniquely identify a style resource.

- The base name has .toml appended if not already ending in that extension
- Individual fetchers can further normalize the path and scheme.

preprocess_relative_url(*url: str*) → str

Preprocess a relative URL.

Only called for urls that lack a scheme (at the very least), being resolved against a base URL that matches this specific fetcher.

Relative paths are file paths; any ~ home reference is expanded at this point.

protocols: tuple[str, ...] = (<Scheme.FILE: 'file'>,)

requires_connection: ClassVar[bool] = False

session: CachedSession | None = None

nitpick.style.fetchers.github module

Support for gh and github schemes.

class nitpick.style.fetchers.github.**GitHubFetcher**(*session: CachedSession | None = None*, *protocols: tuple[str, ...] = (<Scheme.GH: 'gh'>, <Scheme.GITHUB: 'github'>)*, *domains: tuple[str, ...] = ('github.com',)*)

Bases: *HttpFetcher*

Fetch styles from GitHub repositories.

domains: tuple[str, ...] = ('github.com',)

fetch(*url: furl*) → str

Fetch the style from a web location.

normalize(*url: furl*) → furl

Normalize a URL.

Produces a canonical URL, meant to be used to uniquely identify a style resource.

- The base name has .toml appended if not already ending in that extension
- Individual fetchers can further normalize the path and scheme.

preprocess_relative_url(*url: str*) → str

Preprocess a relative URL.

Only called for urls that lack a scheme (at the very least), being resolved against a base URL that matches this specific fetcher.

protocols: tuple[str, ...] = (<Scheme.GH: 'gh'>, <Scheme.GITHUB: 'github'>)

requires_connection: ClassVar[bool] = True

session: CachedSession | None = None

class nitpick.style.fetchers.github.**GitHubURL**(*owner: str*, *repository: str*, *git_reference: str*, *path: tuple[str, ...] = ()*, *auth_token: str | None = None*, *query_params: tuple[tuple[str, str], ...] | None = None*)

Bases: object

Represent a GitHub URL, created from a URL or from its parts.

property api_url: furl

API URL for this repo.

auth_token: str | None = None

property credentials: tuple[str, str] | tuple[()]

Credentials encoded in this URL.

A tuple of `(api_token, '')` if present, or empty tuple otherwise.

property default_branch: str

Default GitHub branch.

classmethod from_furl(*url: furl*) → *GitHubURL*

Create an instance from a parsed URL in any accepted format.

See the code for `test_parsing_github_urls()` for more examples.

git_reference: str

property git_reference_or_default: str

Return the Git reference if informed, or return the default branch.

property long_protocol_url: furl

Long protocol URL (`github`).

owner: str

path: tuple[str, ...] = ()

query_params: tuple[tuple[str, str], ...] | None = None

property raw_content_url: furl

Raw content URL for this path.

repository: str

property short_protocol_url: furl

Short protocol URL (`gh`).

property token: str | None

Token encoded in this URL.

If present and it starts with a `$`, it will be replaced with the value of the environment corresponding to the remaining part of the string.

property url: furl

Default URL built from attributes.

nitpick.style.fetchers.github.**get_default_branch**(*api_url: str*, *, *token: str | None = None*) → str

Get the default branch from the GitHub repo using the API.

For now, for URLs without an authorization token embedded, the request is not authenticated on GitHub, so it might hit a rate limit with: `requests.exceptions.HTTPError: 403 Client Error: rate limit exceeded for url`

This function is using `lru_cache()` as a simple memoizer, trying to avoid this rate limit error.

nitpick.style.fetchers.http module

Base HTTP fetcher, other fetchers can inherit from this to wrap http errors.

class nitpick.style.fetchers.http.**HttpFetcher**(*session: CachedSession | None = None*, *protocols: tuple[str, ...] = (<Scheme.HTTP: 'http'>, <Scheme.HTTPS: 'https'>)*, *domains: tuple[str, ...] = ()*)

Bases: *StyleFetcher*

Fetch a style from an http/https server.

domains: `tuple[str, ...] = ()`

fetch(*url: furl*) → str

Fetch the style from a web location.

normalize(*url: furl*) → furl

Normalize a URL.

Produces a canonical URL, meant to be used to uniquely identify a style resource.

- The base name has .toml appended if not already ending in that extension
- Individual fetchers can further normalize the path and scheme.

preprocess_relative_url(*url: str*) → str

Preprocess a relative URL.

Only called for urls that lack a scheme (at the very least), being resolved against a base URL that matches this specific fetcher.

protocols: `tuple[str, ...] = (<Scheme.HTTP: 'http'>, <Scheme.HTTPS: 'https'>)`

requires_connection: `ClassVar[bool] = True`

session: `CachedSession | None = None`

nitpick.style.fetchers.pypackage module

Support for py schemes.

class nitpick.style.fetchers.pypackage.**BuiltinStyle**(*, *py_url: furl*, *py_url_without_ext: furl*, *path_from_repo_root: str*, *path_from_resources_root: str*)

Bases: `object`

A built-in style file in TOML format.

Method generated by attrs for class BuiltinStyle.

files: `list[str]`

classmethod from_path(*resource_path: Path*) → *BuiltinStyle*

Create a built-in style from a resource path.

identify_tag: `str`

name: `str`

path_from_repo_root: str

path_from_resources_root: str

py_url: furl

py_url_without_ext: furl

pypackage_url: *PythonPackageURL*

url: str

class nitpick.style.fetchers.pypackage.**PythonPackageFetcher**(*session: CachedSession | None = None*, *protocols: tuple[str, ...] = (<Scheme.PY: 'py'>, <Scheme.PYPACKAGE: 'pypackage'>)*, *domains: tuple[str, ...] = ()*)

Bases: *StyleFetcher*

Fetch a style from an installed Python package.

URL schemes: `- py://import/path/of/style/file/<style_file_name> - pypackage://import/path/of/style/file/<style_file_name>`

E.g. `py://some_package/path/nitpick.toml`.

domains: tuple[str, ...] = ()

fetch(*url: furl*) → str

Fetch the style from a Python package.

normalize(*url: furl*) → furl

Normalize a URL.

Produces a canonical URL, meant to be used to uniquely identify a style resource.

- The base name has .toml appended if not already ending in that extension
- Individual fetchers can further normalize the path and scheme.

preprocess_relative_url(*url: str*) → str

Preprocess a relative URL.

Only called for urls that lack a scheme (at the very least), being resolved against a base URL that matches this specific fetcher.

protocols: tuple[str, ...] = (<Scheme.PY: 'py'>, <Scheme.PYPACKAGE: 'pypackage'>)

requires_connection: ClassVar[bool] = False

session: CachedSession | None = None

class nitpick.style.fetchers.pypackage.**PythonPackageURL**(*import_path: str*, *resource_name: str*)

Bases: object

Represent a resource file in installed Python package.

property content_path: Path

Raw path of resource file.

classmethod from_furl(*url: furl*) → *PythonPackageURL*

Create an instance from a parsed URL in any accepted format.

See the code for `test_parsing_python_package_urls()` for more examples.

import_path: `str`

resource_name: `str`

nitpick.style.fetchers.pypackage.**builtin_resources_root**() → Path

Built-in resources root.

nitpick.style.fetchers.pypackage.**builtin_styles**() → Iterable[Path]

List the built-in styles.

nitpick.style.fetchers.pypackage.**repo_root**() → Path

Repository root, 3 levels up from the resources root.

Submodules

nitpick.style.cache module

Cache functions and configuration for styles.

nitpick.style.cache.**parse_cache_option**(*cache_option: str*) → tuple[*CachingEnum*, timedelta | int]

Parse the cache option provided on pyproject.toml.

If no cache if provided or is invalid, the default is *one hour*.

nitpick.style.config module

Config validator.

class nitpick.style.config.**ConfigValidator**(*project:* Project)

Bases: `object`

Validate a nitpick configuration.

project: *Project*

validate(*config_dict: Dict*) → Tuple[Dict, Dict]

Validate an already parsed toml file.

nitpick.style.core module

Style files.

class nitpick.style.core.**StyleManager**(*project:* Project, *offline: bool*, *cache_option: str*)

Bases: `object`

Include styles recursively from one another.

property cache_dir: `Path`

Clear the cache directory (on the project root or on the current directory).

cache_option: *str*

static file_field_pair(*filename: str*, *base_file_class: type[NitpickPlugin]*) → dict[str, *fields.Field*]

Return a schema field with info from a config file class.

find_initial_styles(*configured_styles: Sequence[str]*, *base: str | None = None*) → Iterator[*Fuss*]

Find the initial style(s) and include them.

base is the URL for the source of the initial styles, and is used to resolve relative references. If omitted, defaults to the project root.

static get_default_style_url(*github=False*) → furl

Return the URL of the default style/preset.

include_multiple_styles(*chosen_styles: Iterable[furl]*) → Iterator[*Fuss*]

Include a list of styles (or just one) into this style tree.

load_fixed_name_plugins() → Set[Type[*NitpickPlugin*]]

Separate classes with fixed file names from classes with dynamic files names.

merge_toml_dict() → JsonDict

Merge all included styles into a TOML (actually JSON) dictionary.

offline: *bool*

project: *Project*

rebuild_dynamic_schema() → None

Rebuild the dynamic Marshmallow schema when needed, adding new fields that were found on the style.

12.1.2 Submodules

nitpick.blender module

Dictionary blender and configuration file formats.

class nitpick.blender.**BaseDoc**(*, *path: PathOrStr | None = None*, *string: str | None = None*, *obj: JsonDict | None = None*)

Bases: object

Base class for configuration file formats.

Parameters

- **path** – Path of the config file to be loaded.
- **string** – Config in string format.
- **obj** – Config object (Python dict, YamlDoc, TomlDoc instances).

property as_object: *dict*

String content converted to a Python object (dict, YAML object instance, etc.).

property as_string: *str*

Contents of the file or the original string provided when the instance was created.

abstract load() → bool

Load the configuration from a file, a string or a dict.

property reformatted: str

Reformat the configuration dict as a new string (it might not match the original string/file contents).

class nitpick.blender.**Comparison**(*actual: TBaseDoc*, *expected: JsonDict*, *special_config:* SpecialConfig)

Bases: object

A comparison between two dictionaries, computing missing items and differences.

property diff: TBaseDoc | None

Different data.

property has_changes: bool

Return True is there is a difference or something missing.

property missing: TBaseDoc | None

Missing data.

property replace: TBaseDoc | None

Data to be replaced.

class nitpick.blender.**ElementDetail**(*data: ElementData*, *key: str | list[str]*, *index: int*, *scalar: bool*, *compact: str*)

Bases: object

Detailed information about an element of a list.

Method generated by attrs for class ElementDetail.

property cast_to_dict: Dict[str, Any]

Data cast to dict, for mypy.

compact: str

data: ElementData

classmethod from_data(*index: int*, *data: Dict[str, Any] | str | int | float | CommentedMap | List[Any]*, *jmes_key: str*) → *ElementDetail*

Create an element detail from dict data.

index: int

key: str | list[str]

scalar: bool

class nitpick.blender.**InlineTableTomlDecoder**(*_dict=<class 'dict'>*)

Bases: TomlDecoder

A hacky decoder to work around some bug (or unfinished work) in the Python TOML package.

https://github.com/uiri/toml/issues/362.

bounded_string(*s*)

embed_comments(*idx*, *currentlevel*)

get_empty_inline_table()

Hackity hack for a crappy unmaintained package.

Total lack of respect, the guy doesn't even reply: https://github.com/uiri/toml/issues/361

get_empty_table()

load_array(*a*)

load_inline_object(*line*, *currentlevel*, *multikey=False*, *multibackslash=False*)

load_line(*line*, *currentlevel*, *multikey*, *multibackslash*)

load_value(*v*, *strictly_valid=True*)

preserve_comment(*line_no*, *key*, *comment*, *beginline*)

class nitpick.blender.**JsonDoc**(*, *path: PathOrStr | None = None*, *string: str | None = None*, *obj: JsonDict | None = None*)

Bases: *BaseDoc*

JSON configuration format.

property as_object: dict

String content converted to a Python object (dict, YAML object instance, etc.).

property as_string: str

Contents of the file or the original string provided when the instance was created.

load() → bool

Load a JSON file by its path, a string or a dict.

property reformatted: str

Reformat the configuration dict as a new string (it might not match the original string/file contents).

class nitpick.blender.**ListDetail**(*data: ListOrCommentedSeq*, *elements: list[*ElementDetail*]*)

Bases: object

Detailed info about a list.

Method generated by attrs for class ListDetail.

data: ListOrCommentedSeq

elements: list[*ElementDetail*]

find_by_key(*desired:* ElementDetail) → *ElementDetail* | None

Find an element by key.

classmethod from_data(*data: List[Any] | CommentedSeq*, *jmes_key: str*) → *ListDetail*

Create a list detail from list data.

nitpick.blender.**SEPARATOR_QUOTED_SPLIT = '#$@'**

Special unique separator for *nitpick.blender.quoted_split()*.

class nitpick.blender.**SensibleYAML**

Bases: YAML

YAML with sensible defaults but an inefficient dump to string.

Output of dump() as a string.

typ: 'rt'/None -> RoundTripLoader/RoundTripDumper, (default)
'safe' -> SafeLoader/SafeDumper, 'unsafe' -> normal/unsafe Loader/Dumper 'base' -> baseloader

pure: if True only use Python modules input/output: needed to work as context manager plug_ins: a list of plug-in files

Xdump_all(*documents: Any*, *stream: Any*, *, *transform: Any | None = None*) → Any

Serialize a sequence of Python objects into a YAML stream.

property block_seq_indent: Any

compact(*seq_seq: Any | None = None*, *seq_map: Any | None = None*) → None

compose(*stream: Path | Any*) → Any

Parse the first YAML document in a stream and produce the corresponding representation tree.

compose_all(*stream: Path | Any*) → Any

Parse all YAML documents in a stream and produce corresponding representation trees.

property composer: Any

property constructor: Any

dump(*data: Path | Any*, *stream: Any | None = None*, *, *transform: Any | None = None*) → Any

dump_all(*documents: Any*, *stream: Path | Any*, *, *transform: Any | None = None*) → Any

dumps(*data*) → str

Dump to a string… who would want such a thing? One can dump to a file or stdout.

emit(*events: Any*, *stream: Any*) → None

Emit YAML parsing events into a stream. If stream is None, return the produced string instead.

property emitter: Any

get_constructor_parser(*stream: Any*) → Any

the old cyaml needs special setup, and therefore the stream

get_serializer_representer_emitter(*stream: Any*, *tlca: Any*) → Any

property indent: Any

load(*stream: Path | Any*) → Any

at this point you either have the non-pure Parser (which has its own reader and scanner) or you have the pure Parser. If the pure Parser is set, then set the Reader and Scanner, if not already set. If either the Scanner or Reader are set, you cannot use the non-pure Parser,

so reset it to the pure parser and set the Reader resp. Scanner if necessary

load_all(*stream: Path | Any*) → Any

loads(*string: str*)

Load YAML from a string… that unusual use case in a world of files only.

map(***kw: Any*) → Any

official_plug_ins() → Any

search for list of subdirs that are plug-ins, if __file__ is not available, e.g. single file installers that are not properly emulating a file-system (issue 324) no plug-ins will be found. If any are packaged, you know which file that are and you can explicitly provide it during instantiation:

yaml = ruamel.yaml.YAML(plug_ins=[‘ruamel/yaml/jinja2/__plug_in__’])

parse(*stream: Any*) → Any

Parse a YAML stream and produce parsing events.

property parser: Any

property reader: Any

register_class(*cls: Any*) → Any

register a class for dumping/loading - if it has attribute yaml_tag use that to register, else use class name - if it has methods to_yaml/from_yaml use those to dump/load else dump attributes

as mapping

property representer: Any

property resolver: Any

scan(*stream: Any*) → Any

Scan a YAML stream and produce scanning tokens.

property scanner: Any

seq(**args: Any*) → Any

serialize(*node: Any*, *stream: Any | None*) → Any

Serialize a representation tree into a YAML stream. If stream is None, return the produced string instead.

serialize_all(*nodes: Any*, *stream: Any | None*) → Any

Serialize a sequence of representation trees into a YAML stream. If stream is None, return the produced string instead.

property serializer: Any

property version: Any | None

class nitpick.blender.**TomlDoc**(**, path: PathOrStr | None = None, string: str | None = None, obj: JsonDict | None = None, use_tomlkit=False*)

Bases: BaseDoc

TOML configuration format.

property as_object: dict

String content converted to a Python object (dict, YAML object instance, etc.).

property as_string: str

Contents of the file or the original string provided when the instance was created.

load() → bool

Load a TOML file by its path, a string or a dict.

property reformatted: str

Reformat the configuration dict as a new string (it might not match the original string/file contents).

class nitpick.blender.**YamlDoc**(**, path: PathOrStr | None = None, string: str | None = None, obj: JsonDict | None = None*)

Bases: BaseDoc

YAML configuration format.

property as_object: dict

String content converted to a Python object (dict, YAML object instance, etc.).

property as_string: str

Contents of the file or the original string provided when the instance was created.

load() → bool

Load a YAML file by its path, a string or a dict.

property reformatted: str

Reformat the configuration dict as a new string (it might not match the original string/file contents).

updater: *SensibleYAML*

nitpick.blender.**compare_lists_with_dictdiffer**(*actual: list | dict*, *expected: list | dict*, ***, *return_list: bool = True*) → list | dict

Compare two lists using dictdiffer.

nitpick.blender.**custom_reducer**(*separator: str*) → Callable

Custom reducer for `flatten_dict.flatten_dict.flatten()` accepting a separator.

nitpick.blender.**custom_splitter**(*separator: str*) → Callable

Custom splitter for `flatten_dict.flatten_dict.unflatten()` accepting a separator.

nitpick.blender.**flatten_quotes**(*dict_: Dict[str, Any]*, *separator='.'*) → Dict[str, Any]

Flatten a dict keeping quotes in keys.

nitpick.blender.**is_scalar**(*value: Dict[str, Any] | List[Any] | str | float*) → bool

Return True if the value is NOT a dict or a list.

nitpick.blender.**quote_if_dotted**(*key: str*) → str

Quote the key if it has a dot.

nitpick.blender.**quote_reducer**(*separator: str*) → Callable

Reducer used to unflatten dicts. Quote keys when they have dots.

nitpick.blender.**quoted_split**(*string_: str*, *separator='.'*) → list[str]

Split a string by a separator, but considering quoted parts (single or double quotes).

```
>>> quoted_split("my.key.without.quotes")
['my', 'key', 'without', 'quotes']
>>> quoted_split('"double.quoted.string"')
['double.quoted.string']
>>> quoted_split('"double.quoted.string".and.after')
['double.quoted.string', 'and', 'after']
>>> quoted_split('something.before."double.quoted.string"')
['something', 'before', 'double.quoted.string']
>>> quoted_split("'single.quoted.string'")
['single.quoted.string']
>>> quoted_split("'single.quoted.string'.and.after")
['single.quoted.string', 'and', 'after']
>>> quoted_split("something.before.'single.quoted.string'")
['something', 'before', 'single.quoted.string']
```

nitpick.blender.**quotes_splitter**(*flat_key: str*) → tuple[str, ...]

Split keys keeping quoted strings together.

nitpick.blender.**replace_or_add_list_element**(*yaml_obj: CommentedSeq | CommentedMap*, *element: Any*, *key: str*, *index: int*) → None

Replace or add a new element in a YAML sequence of mappings.

nitpick.blender.**search_json**(*json_data: ElementData*, *jmespath_expression: ParsedResult | str*, *default: Any | None = None*) → Any

Search a dictionary or list using a JMESPath expression. Return a default value if not found.

```
>>> data = {"root": {"app": [1, 2], "test": "something"}}
>>> search_json(data, "root.app", None)
[1, 2]
>>> search_json(data, "root.test", None)
'something'
>>> search_json(data, "root.unknown", "")
''
>>> search_json(data, "root.unknown", None)
```

```
>>> search_json(data, "root.unknown")
```

```
>>> search_json(data, jmespath.compile("root.app"), [])
[1, 2]
>>> search_json(data, jmespath.compile("root.whatever"), "xxx")
'xxx'
>>> search_json(data, "")
```

```
>>> search_json(data, None)
```

Parameters

- **jmespath_expression** – A compiled JMESPath expression or a string with an expression.
- **json_data** – The dictionary to be searched.
- **default** – Default value in case nothing is found.

Returns

The object that was found or the default value.

nitpick.blender.**set_key_if_not_empty**(*dict_: Dict[str, Any]*, *key: str*, *value: Any*) → None

Update the dict if the value is valid.

nitpick.blender.**traverse_toml_tree**(*document: TOMLDocument*, *dictionary*)

Traverse a TOML document recursively and change values, keeping its formatting and comments.

nitpick.blender.**traverse_yaml_tree**(*yaml_obj: CommentedSeq | CommentedMap*, *change: Dict[str, Any]*)

Traverse a YAML document recursively and change values, keeping its formatting and comments.

nitpick.cli module

Module that contains the command line app.

Why does this file exist, and why not put this in __main__?

You might be tempted to import things from __main__ later, but that will cause problems: the code will get executed twice:

- **When you run *python -mnitpick* python will execute __main__.py as a script.**
  That means there won't be any nitpick.__main__ in sys.modules.

- **When you import __main__ it will get executed again (as a module) because** there's no `nitpick.__main__` in `sys.modules`.

Also see (1) from https://click.palletsprojects.com/en/5.x/setuptools/#setuptools-integration

`nitpick.cli.`**`common_fix_or_check`**(*context*, *verbose: int*, *files*, *check_only: bool*) → None

Common CLI code for both "fix" and "check" commands.

`nitpick.cli.`**`get_nitpick`**(*context: Context*) → *Nitpick*

Create a Nitpick instance from the click context parameters.

nitpick.compat module

Handle import compatibility issues.

nitpick.config module

Special configurations.

class `nitpick.config.`**`OverridableConfig`**(*from_plugin: Dict[str, Any] = _Nothing.NOTHING*, *from_style: Dict[str, Any] = _Nothing.NOTHING*, *value: Dict[str, Any] = _Nothing.NOTHING*)

Bases: `object`

Configs formed from defaults from the plugin that can be overridden by the style.

Method generated by attrs for class OverridableConfig.

`from_plugin:` `Dict[str, Any]`

`from_style:` `Dict[str, Any]`

`value:` `Dict[str, Any]`

class `nitpick.config.`**`SpecialConfig`**(*list_keys: OverridableConfig = _Nothing.NOTHING*)

Bases: `object`

Special configurations for plugins.

Method generated by attrs for class SpecialConfig.

`list_keys:` *`OverridableConfig`*

nitpick.constants module

Constants.

nitpick.core module

The Nitpick application.

class nitpick.core.**Nitpick**

Bases: object

The Nitpick API.

configured_files(**partial_names: str*) → List[Path]

List of files configured in the Nitpick style. Filter only the selected partial names.

echo(*message: str*)

Echo a message on the terminal, with the relative path at the beginning.

enforce_present_absent(**partial_names: str*) → Iterator[*Fuss*]

Enforce files that should be present or absent.

Parameters
partial_names – Names of the files to enforce configs for.

Returns
Fuss generator.

enforce_style(**partial_names: str*, *autofix=True*) → Iterator[*Fuss*]

Read the merged style and enforce the rules in it.

1. Get all root keys from the merged style (every key is a filename, except "nitpick").
2. For each file name, find the plugin(s) that can handle the file.

Parameters

- **partial_names** – Names of the files to enforce configs for.
- **autofix** – Flag to modify files, if the plugin supports it (default: True).

Returns
Fuss generator.

init(*project_root: Path | str | None = None*, *offline: bool | None = None*) → *Nitpick*

Initialize attributes of the singleton.

offline: bool

project: *Project*

run(**partial_names: str*, *autofix=False*) → Iterator[*Fuss*]

Run Nitpick.

Parameters

- **partial_names** – Names of the files to enforce configs for.
- **autofix** – Flag to modify files, if the plugin supports it (default: True).

Returns
Fuss generator.

classmethod singleton() → *Nitpick*

Return a single instance of the class.

nitpick.enums module

Enums.

class nitpick.enums.**CachingEnum**(*value*)

Bases: IntEnum

Caching modes for styles.

EXPIRES = 3

The cache expires after the configured amount of time (minutes/hours/days).

FOREVER = 2

Once the style(s) are cached, they never expire.

NEVER = 1

Never cache, the style file(s) are always looked-up.

class nitpick.enums.**OptionEnum**(*value*)

Bases: _OptionMixin, Enum

Options to be used with the CLI.

OFFLINE = 'Offline mode: no style will be downloaded (no HTTP requests at all)'

name: str

nitpick.exceptions module

Nitpick exceptions.

class nitpick.exceptions.**Deprecation**

Bases: object

All deprecation messages in a single class.

When it's time to break compatibility, remove a method/warning below, and older config files will trigger validation errors on Nitpick.

static jsonfile_section(*style_errors: dict[str, Any]*) → bool

The [nitpick.JSONFile] is not needed anymore; JSON files are now detected by the extension.

static pre_commit_repos_with_yaml_key() → bool

The pre-commit config should not have the "repos.yaml" key anymore; this is the old style.

Slight breaking change in the TOML config format: ditching the old TOML config.

static pre_commit_without_dash(*path_from_root: str*) → bool

The pre-commit config should start with a dot on the config file.

exception nitpick.exceptions.**QuitComplainingError**(*violations:* Fuss | *list[*Fuss*]*)

Bases: Exception

Quit complaining and exit the application.

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

nitpick.exceptions.**pretty_exception**(*err: Exception*, *message: str = ''*)

Return a pretty error message with the full path of the Exception.

nitpick.fields module

Custom Marshmallow fields and validators.

class nitpick.fields.**Dict**(*keys: Field | type | None = None*, *values: Field | type | None = None*, ***kwargs*)

Bases: Mapping

A dict field. Supports dicts and dict-like objects. Extends Mapping with dict as the mapping_type.

Example:

```
numbers = fields.Dict(keys=fields.Str(), values=fields.Float())
```

Parameters

kwargs – The same keyword arguments that Mapping receives.

New in version 2.1.0.

property context

The context dictionary for the parent Schema.

property default

default_error_messages = {'invalid': 'Not a valid mapping type.'}

Default error messages.

deserialize(*value: Any*, *attr: str | None = None*, *data: Mapping[str, Any] | None = None*, ***kwargs*)

Deserialize value.

Parameters

- **value** – The value to deserialize.
- **attr** – The attribute/key in *data* to deserialize.
- **data** – The raw input data passed to *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises

ValidationError – If an invalid value is passed or if a required value is missing.

fail(*key: str*, ***kwargs*)

Helper method that raises a *ValidationError* with an error message from self.error_messages.

Deprecated since version 3.0.0: Use *make_error <marshmallow.fields.Field.make_error>* instead.

get_value(*obj*, *attr*, *accessor=None*, *default=<marshmallow.missing>*)

Return the value for a given key from an object.

Parameters

- **obj** (*object*) – The object to get the value from.
- **attr** (*str*) – The attribute/key in *obj* to get the value from.
- **accessor** (*callable*) – A callable used to retrieve the value of *attr* from the object *obj*. Defaults to *marshmallow.utils.get_value*.

make_error(*key: str*, ***kwargs*) → ValidationError

Helper method to make a *ValidationError* with an error message from `self.error_messages`.

mapping_type

alias of `dict`

property missing

name = None

parent = None

root = None

serialize(*attr: str*, *obj: Any*, *accessor: Callable[[Any, str, Any], Any] | None = None*, ***kwargs*)

Pulls the value for the given key from the object, applies the field's formatting and returns the result.

Parameters

- **attr** – The attribute/key to get from the object.
- **obj** – The object to access the attribute/key from.
- **accessor** – Function used to access values from `obj`.
- **kwargs** – Field-specific keyword arguments.

class nitpick.fields.**Field**(**, load_default: typing.Any = <marshmallow.missing>, missing: typing.Any = <marshmallow.missing>, dump_default: typing.Any = <marshmallow.missing>, default: typing.Any = <marshmallow.missing>, data_key: str | None = None, attribute: str | None = None, validate: None | (typing.Callable[[typing.Any], typing.Any] | typing.Iterable[typing.Callable[[typing.Any], typing.Any]]) = None, required: bool = False, allow_none: bool | None = None, load_only: bool = False, dump_only: bool = False, error_messages: dict[str, str] | None = None, metadata: typing.Mapping[str, typing.Any] | None = None, **additional_metadata*)

Bases: `FieldABC`

Basic field from which other fields should extend. It applies no formatting by default, and should only be used in cases where data does not need to be formatted before being serialized or deserialized. On error, the name of the field will be returned.

Parameters

- **dump_default** – If set, this value will be used during serialization if the input value is missing. If not set, the field will be excluded from the serialized output if the input value is missing. May be a value or a callable.
- **load_default** – Default deserialization value for the field if the field is not found in the input data. May be a value or a callable.
- **data_key** – The name of the dict key in the external representation, i.e. the input of *load* and the output of *dump*. If *None*, the key will match the name of the field.
- **attribute** – The name of the attribute to get the value from when serializing. If *None*, assumes the attribute has the same name as the field. Note: This should only be used for very specific use cases such as outputting multiple fields for a single attribute. In most cases, you should use `data_key` instead.
- **validate** – Validator or collection of validators that are called during deserialization. Validator takes a field's input value as its only parameter and returns a boolean. If it returns *False*, an `ValidationError` is raised.

- **required** – Raise a `ValidationError` if the field value is not supplied during deserialization.
- **allow_none** – Set this to *True* if *None* should be considered a valid value during validation/deserialization. If `load_default=None` and `allow_none` is unset, will default to `True`. Otherwise, the default is `False`.
- **load_only** – If *True* skip this field during serialization, otherwise its value will be present in the serialized data.
- **dump_only** – If *True* skip this field during deserialization, otherwise its value will be present in the deserialized object. In the context of an HTTP API, this effectively marks the field as "read-only".
- **error_messages** (*dict*) – Overrides for *Field.default_error_messages*.
- **metadata** – Extra information to be stored as field metadata.

Changed in version 2.0.0: Removed *error* parameter. Use `error_messages` instead.

Changed in version 2.0.0: Added *allow_none* parameter, which makes validation/deserialization of *None* consistent across fields.

Changed in version 2.0.0: Added *load_only* and *dump_only* parameters, which allow field skipping during the (de)serialization process.

Changed in version 2.0.0: Added *missing* parameter, which indicates the value for a field if the field is not found during deserialization.

Changed in version 2.0.0: `default` value is only used if explicitly set. Otherwise, missing values inputs are excluded from serialized output.

Changed in version 3.0.0b8: Add `data_key` parameter for the specifying the key in the input and output data. This parameter replaced both `load_from` and `dump_to`.

property context

The context dictionary for the parent `Schema`.

property default

default_error_messages = {'null': 'Field may not be null.', 'required': 'Missing data for required field.', 'validator_failed': 'Invalid value.'}

Default error messages for various kinds of errors. The keys in this dictionary are passed to *Field.make_error*. The values are error messages passed to `marshmallow.exceptions.ValidationError`.

deserialize(*value: Any*, *attr: str | None = None*, *data: Mapping[str, Any] | None = None*, ***kwargs*)

Deserialize `value`.

Parameters

- **value** – The value to deserialize.
- **attr** – The attribute/key in *data* to deserialize.
- **data** – The raw input data passed to *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises

ValidationError – If an invalid value is passed or if a required value is missing.

fail(*key: str*, ***kwargs*)

Helper method that raises a *ValidationError* with an error message from `self.error_messages`.

Deprecated since version 3.0.0: Use *make_error <marshmallow.fields.Field.make_error>* instead.

get_value(*obj*, *attr*, *accessor=None*, *default=<marshmallow.missing>*)

Return the value for a given key from an object.

Parameters

- **obj** (*object*) – The object to get the value from.
- **attr** (*str*) – The attribute/key in *obj* to get the value from.
- **accessor** (*callable*) – A callable used to retrieve the value of *attr* from the object *obj*. Defaults to *marshmallow.utils.get_value*.

make_error(*key: str*, ***kwargs*) → ValidationError

Helper method to make a *ValidationError* with an error message from `self.error_messages`.

property missing

name = None

parent = None

root = None

serialize(*attr: str*, *obj: Any*, *accessor: Callable[[Any, str, Any], Any] | None = None*, ***kwargs*)

Pulls the value for the given key from the object, applies the field's formatting and returns the result.

Parameters

- **attr** – The attribute/key to get from the object.
- **obj** – The object to access the attribute/key from.
- **accessor** – Function used to access values from `obj`.
- **kwargs** – Field-specific keyword arguments.

class nitpick.fields.**List**(*cls_or_instance: Field | type*, ***kwargs*)

Bases: *Field*

A list field, composed with another *Field* class or instance.

Example:

```
numbers = fields.List(fields.Float())
```

Parameters

- **cls_or_instance** – A field class or instance.
- **kwargs** – The same keyword arguments that *Field* receives.

Changed in version 2.0.0: The `allow_none` parameter now applies to deserialization and has the same semantics as the other fields.

Changed in version 3.0.0rc9: Does not serialize scalar values to single-item lists.

property context

The context dictionary for the parent `Schema`.

property default

default_error_messages = {'invalid': 'Not a valid list.'}

Default error messages.

deserialize(*value: Any*, *attr: str | None = None*, *data: Mapping[str, Any] | None = None*, ***kwargs*)

Deserialize `value`.

Parameters

- **value** – The value to deserialize.
- **attr** – The attribute/key in *data* to deserialize.
- **data** – The raw input data passed to *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises

ValidationError – If an invalid value is passed or if a required value is missing.

fail(*key: str*, ***kwargs*)

Helper method that raises a *ValidationError* with an error message from `self.error_messages`.

Deprecated since version 3.0.0: Use *make_error <marshmallow.fields.Field.make_error>* instead.

get_value(*obj*, *attr*, *accessor=None*, *default=<marshmallow.missing>*)

Return the value for a given key from an object.

Parameters

- **obj** (*object*) – The object to get the value from.
- **attr** (*str*) – The attribute/key in *obj* to get the value from.
- **accessor** (*callable*) – A callable used to retrieve the value of *attr* from the object *obj*. Defaults to *marshmallow.utils.get_value*.

make_error(*key: str*, ***kwargs*) → ValidationError

Helper method to make a *ValidationError* with an error message from `self.error_messages`.

property missing

name = None

parent = None

root = None

serialize(*attr: str*, *obj: Any*, *accessor: Callable[[Any, str, Any], Any] | None = None*, ***kwargs*)

Pulls the value for the given key from the object, applies the field's formatting and returns the result.

Parameters

- **attr** – The attribute/key to get from the object.
- **obj** – The object to access the attribute/key from.
- **accessor** – Function used to access values from `obj`.
- **kwargs** – Field-specific keyword arguments.

class nitpick.fields.**Nested**(*nested: SchemaABC | type | str | dict[str, Field | type] | typing.Callable[[], SchemaABC | dict[str, Field | type]], *, dump_default: typing.Any = <marshmallow.missing>, default: typing.Any = <marshmallow.missing>, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, unknown: str | None = None, **kwargs*)

Bases: *Field*

Allows you to nest a Schema inside a field.

Examples:

```
class ChildSchema(Schema):
    id = fields.Str()
    name = fields.Str()
    # Use lambda functions when you need two-way nesting or self-nesting
    parent = fields.Nested(lambda: ParentSchema(only=("id",)), dump_only=True)
    siblings = fields.List(fields.Nested(lambda: ChildSchema(only=("id", "name"))))

class ParentSchema(Schema):
    id = fields.Str()
    children = fields.List(
        fields.Nested(ChildSchema(only=("id", "parent", "siblings")))
    )
    spouse = fields.Nested(lambda: ParentSchema(only=("id",)))
```

When passing a *Schema <marshmallow.Schema>* instance as the first argument, the instance's `exclude`, `only`, and `many` attributes will be respected.

Therefore, when passing the `exclude`, `only`, or `many` arguments to *fields.Nested*, you should pass a *Schema <marshmallow.Schema>* class (not an instance) as the first argument.

```
# Yes
author = fields.Nested(UserSchema, only=('id', 'name'))

# No
author = fields.Nested(UserSchema(), only=('id', 'name'))
```

Parameters

- **nested** – *Schema* instance, class, class name (string), dictionary, or callable that returns a *Schema* or dictionary. Dictionaries are converted with *Schema.from_dict*.
- **exclude** – A list or tuple of fields to exclude.
- **only** – A list or tuple of fields to marshal. If *None*, all fields are marshalled. This parameter takes precedence over `exclude`.
- **many** – Whether the field is a collection of objects.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **kwargs** – The same keyword arguments that *Field* receives.

property context

The context dictionary for the parent Schema.

property default

default_error_messages = {'type': 'Invalid type.'}

Default error messages.

deserialize(*value: Any*, *attr: str | None = None*, *data: Mapping[str, Any] | None = None*, ***kwargs*)

Deserialize `value`.

Parameters

- **value** – The value to deserialize.
- **attr** – The attribute/key in *data* to deserialize.
- **data** – The raw input data passed to *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises

ValidationError – If an invalid value is passed or if a required value is missing.

fail(*key: str*, ***kwargs*)

Helper method that raises a *ValidationError* with an error message from `self.error_messages`.

Deprecated since version 3.0.0: Use *make_error <marshmallow.fields.Field.make_error>* instead.

get_value(*obj*, *attr*, *accessor=None*, *default=<marshmallow.missing>*)

Return the value for a given key from an object.

Parameters

- **obj** (*object*) – The object to get the value from.
- **attr** (*str*) – The attribute/key in *obj* to get the value from.
- **accessor** (*callable*) – A callable used to retrieve the value of *attr* from the object *obj*. Defaults to *marshmallow.utils.get_value*.

make_error(*key: str*, ***kwargs*) → ValidationError

Helper method to make a *ValidationError* with an error message from `self.error_messages`.

property missing

name = None

parent = None

root = None

property schema

The nested Schema object.

Changed in version 1.0.0: Renamed from *serializer* to *schema*.

serialize(*attr: str*, *obj: Any*, *accessor: Callable[[Any, str, Any], Any] | None = None*, ***kwargs*)

Pulls the value for the given key from the object, applies the field's formatting and returns the result.

Parameters

- **attr** – The attribute/key to get from the object.
- **obj** – The object to access the attribute/key from.
- **accessor** – Function used to access values from `obj`.

- **kwargs** – Field-specific keyword arguments.

class nitpick.fields.**String**(*, load_default: typing.Any = <marshmallow.missing>, missing: typing.Any = <marshmallow.missing>, dump_default: typing.Any = <marshmallow.missing>, default: typing.Any = <marshmallow.missing>, data_key: str | None = None, attribute: str | None = None, validate: None | (typing.Callable[[typing.Any], typing.Any] | typing.Iterable[typing.Callable[[typing.Any], typing.Any]]) = None, required: bool = False, allow_none: bool | None = None, load_only: bool = False, dump_only: bool = False, error_messages: dict[str, str] | None = None, metadata: typing.Mapping[str, typing.Any] | None = None, **additional_metadata)

Bases: *Field*

A string field.

Parameters
kwargs – The same keyword arguments that *Field* receives.

property context

The context dictionary for the parent `Schema`.

property default

default_error_messages = {'invalid': 'Not a valid string.', 'invalid_utf8': 'Not a valid utf-8 string.'}

Default error messages.

deserialize(*value: Any*, *attr: str | None = None*, *data: Mapping[str, Any] | None = None*, ***kwargs*)

Deserialize `value`.

Parameters

- **value** – The value to deserialize.
- **attr** – The attribute/key in *data* to deserialize.
- **data** – The raw input data passed to *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises
ValidationError – If an invalid value is passed or if a required value is missing.

fail(*key: str*, ***kwargs*)

Helper method that raises a *ValidationError* with an error message from `self.error_messages`.

Deprecated since version 3.0.0: Use *make_error <marshmallow.fields.Field.make_error>* instead.

get_value(*obj*, *attr*, *accessor=None*, *default=<marshmallow.missing>*)

Return the value for a given key from an object.

Parameters

- **obj** (*object*) – The object to get the value from.
- **attr** (*str*) – The attribute/key in *obj* to get the value from.
- **accessor** (*callable*) – A callable used to retrieve the value of *attr* from the object *obj*. Defaults to *marshmallow.utils.get_value*.

make_error(*key: str*, ***kwargs*) → ValidationError

Helper method to make a *ValidationError* with an error message from `self.error_messages`.

property missing

name = None

parent = None

root = None

serialize(*attr: str*, *obj: Any*, *accessor: Callable[[Any, str, Any], Any] | None = None*, ***kwargs*)

Pulls the value for the given key from the object, applies the field's formatting and returns the result.

Parameters

- **attr** – The attribute/key to get from the object.
- **obj** – The object to access the attribute/key from.
- **accessor** – Function used to access values from `obj`.
- **kwargs** – Field-specific keyword arguments.

nitpick.fields.**URL**

alias of `Url`

nitpick.flake8 module

Flake8 plugin to check files.

class nitpick.flake8.**NitpickFlake8Extension**(*tree=None*, *filename='(none)'*)

Bases: `object`

Main class for the flake8 extension.

Method generated by attrs for class NitpickFlake8Extension.

static add_options(*option_manager: OptionManager*)

Add the offline option.

build_flake8_error(*obj: Fuss*) → Tuple[int, int, str, Type]

Return a flake8 error from a fuss.

collect_errors() → Iterator[*Fuss*]

Collect all possible Nitpick errors.

name = 'nitpick'

static parse_options(*option_manager: OptionManager*, *options*, *args*)

Create the Nitpick app, set logging from the verbose flags, set offline mode.

This function is called only once by flake8, so it's a good place to create the app.

run() → Iterator[Tuple[int, int, str, Type]]

Run the check plugin.

version = '0.33.2'

nitpick.generic module

Generic functions and classes.

nitpick.generic.**filter_names**(*iterable: Iterable*, **partial_names: str*) → list[str]

Filter names and keep only the desired partial names.

Exclude the project name automatically.

```
>>> file_list = ['requirements.txt', 'tox.ini', 'setup.py', 'nitpick']
>>> filter_names(file_list)
['requirements.txt', 'tox.ini', 'setup.py']
>>> filter_names(file_list, 'ini', '.py')
['tox.ini', 'setup.py']
```

```
>>> mapping = {'requirements.txt': None, 'tox.ini': 1, 'setup.py': 2, 'nitpick': 3}
>>> filter_names(mapping)
['requirements.txt', 'tox.ini', 'setup.py']
>>> filter_names(file_list, 'x')
['requirements.txt', 'tox.ini']
```

nitpick.generic.**relative_to_current_dir**(*path_or_str: PathOrStr | None*) → str

Return a relative path to the current dir or an absolute path.

nitpick.generic.**url_to_python_path**(*url: furl*) → Path

Convert the segments of a file URL to a path.

nitpick.generic.**version_to_tuple**(*version: str | None = None*) → tuple[int, ...]

Transform a version number into a tuple of integers, for comparison.

```
>>> version_to_tuple("")
()
>>> version_to_tuple("  ")
()
>>> version_to_tuple(None)
()
>>> version_to_tuple("1.0.1")
(1, 0, 1)
>>> version_to_tuple(" 0.2 ")
(0, 2)
>>> version_to_tuple(" 2 ")
(2,)
```

Parameters

version – String with the version number. It must be integers split by dots.

Returns

Tuple with the version number.

nitpick.project module

A project to be nitpicked.

class nitpick.project.**Configuration**(*file: Path | None*, *styles: str | list[str]*, *cache: str*)

Bases: object

Configuration read from one of the CONFIG_FILES.

cache: str

file: Path | None

styles: str | list[str]

class nitpick.project.**Project**(*root: PathOrStr | None = None*)

Bases: object

A project to be nitpicked.

create_configuration(*config:* Configuration, **style_urls: str*) → None

Create a configuration file.

merge_styles(*offline: bool*) → Iterator[*Fuss*]

Merge one or multiple style files.

property plugin_manager: PluginManager

Load all defined plugins.

read_configuration() → *Configuration*

Search for a configuration file and validate it against a Marshmallow schema.

property root: Path

Root dir of the project.

class nitpick.project.**ToolNitpickSectionSchema**(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)

Bases: *BaseNitpickSchema*

Validation schema for the [tool.nitpick] section on pyproject.toml.

class Meta

Bases: object

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- fields: Tuple or list of fields to include in the serialized result.

- **additional: Tuple or list of fields to include *in addition* to the**
  explicitly declared fields. additional and fields are mutually-exclusive options.
- **include: Dictionary of additional fields to include in the schema. It is**
  usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **exclude: Tuple or list of fields to exclude in the serialized result.**
  Nested fields can be represented with dot delimiters.
- dateformat: Default format for *Date* <*fields.Date*> fields.
- datetimeformat: Default format for *DateTime* <*fields.DateTime*> fields.
- timeformat: Default format for *Time* <*fields.Time*> fields.
- **render_module: Module to use for *loads* <*Schema.loads*> and *dumps* <*Schema.dumps*>.**
  Defaults to *json* from the standard library.
- **ordered: If *True*, order serialization output according to the**
  order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **index_errors: If *True*, errors dictionaries will include the index**
  of invalid items in a collection.
- load_only: Tuple or list of fields to exclude from serialized results.
- dump_only: Tuple or list of fields to exclude from deserialization
- **unknown: Whether to exclude, include, or raise an error for unknown**
  fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **register: Whether to register the *Schema* with marshmallow's internal**
  class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

OPTIONS_CLASS

alias of SchemaOpts

TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}

property dict_class: type

dump(*obj: Any*, *, *many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/configuration.html.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, ***, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, *, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, *, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

loads(*json_data: str*, *, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: SchemaOpts = <marshmallow.schema.SchemaOpts object>

property set_class: type

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns

A dictionary of validation errors.

New in version 1.1.0.

nitpick.project.**confirm_project_root**(*dir_: PathOrStr | None = None*) → Path

Confirm this is the root dir of the project (the one that has one of the `ROOT_FILES`).

nitpick.project.**find_main_python_file**(*root_dir: Path*) → Path

Find the main Python file in the root dir, the one that will be used to report Flake8 warnings.

The search order is: 1. Python files that belong to the root dir of the project (e.g.: `setup.py`, `autoapp.py`). 2. `manage.py`: they can be on the root or on a subdir (Django projects). 3. Any other `*.py` Python file on the root dir and subdir. This avoid long recursions when there is a `node_modules` subdir for instance.

nitpick.project.**glob_files**(*dir_: Path*, *file_patterns: Iterable[str]*) → set[Path]

Search a directory looking for file patterns.

nitpick.schemas module

Marshmallow schemas.

class nitpick.schemas.**BaseNitpickSchema**(***, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: Schema

Base schema for all others, with default error messages.

class Meta

Bases: object

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **`additional`: Tuple or list of fields to include *in addition* to the** explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **`include`: Dictionary of additional fields to include in the schema. It is** usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **`exclude`: Tuple or list of fields to exclude in the serialized result.** Nested fields can be represented with dot delimiters.
- `dateformat`: Default format for *Date* <*fields.Date*> fields.
- `datetimeformat`: Default format for *DateTime* <*fields.DateTime*> fields.
- `timeformat`: Default format for *Time* <*fields.Time*> fields.
- **`render_module`: Module to use for *loads* <*Schema.loads*> and *dumps* <*Schema.dumps*>.** Defaults to *json* from the standard library.
- **`ordered`: If *True*, order serialization output according to the** order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **`index_errors`: If *True*, errors dictionaries will include the index** of invalid items in a collection.
- `load_only`: Tuple or list of fields to exclude from serialized results.
- `dump_only`: Tuple or list of fields to exclude from deserialization
- **`unknown`: Whether to exclude, include, or raise an error for unknown** fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **`register`: Whether to register the *Schema* with marshmallow's internal** class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

`OPTIONS_CLASS`

alias of `SchemaOpts`

`TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}`

property dict_class: type

dump(*obj: Any*, *, *many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/nitpick_section.html.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, *, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, ***, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

loads(*json_data: str*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: SchemaOpts = <marshmallow.schema.SchemaOpts object>

property set_class: type

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns
A dictionary of validation errors.

New in version 1.1.0.

class nitpick.schemas.**BaseStyleSchema**(***, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: Schema

Base validation schema for style files. Dynamic fields will be added to it later.

class Meta

Bases: `object`

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **additional: Tuple or list of fields to include *in addition* to the**
  explicitly declared fields. `additional` and `fields` are mutually-exclusive options.

- **include: Dictionary of additional fields to include in the schema. It is**
  usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **exclude: Tuple or list of fields to exclude in the serialized result.**
  Nested fields can be represented with dot delimiters.
- dateformat: Default format for *Date* <*fields.Date*> fields.
- datetimeformat: Default format for *DateTime* <*fields.DateTime*> fields.
- timeformat: Default format for *Time* <*fields.Time*> fields.
- **render_module: Module to use for *loads* <*Schema.loads*> and *dumps* <*Schema.dumps*>.**
  Defaults to *json* from the standard library.
- **ordered: If *True*, order serialization output according to the**
  order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **index_errors: If *True*, errors dictionaries will include the index**
  of invalid items in a collection.
- load_only: Tuple or list of fields to exclude from serialized results.
- dump_only: Tuple or list of fields to exclude from deserialization
- **unknown: Whether to exclude, include, or raise an error for unknown**
  fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **register: Whether to register the *Schema* with marshmallow's internal**
  class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

OPTIONS_CLASS

alias of SchemaOpts

TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}

property dict_class: type

dump(*obj: Any*, *, *many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns

Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown file. See https://nitpick.rtfd.io/en/latest/plugins.html.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, ***, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, ***, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.

- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

loads(*json_data: str*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: SchemaOpts = <marshmallow.schema.SchemaOpts object>

property set_class: type

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, *, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns

A dictionary of validation errors.

New in version 1.1.0.

class nitpick.schemas.**IniSchema**(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: *BaseNitpickSchema*

Validation schema for INI files.

class Meta

Bases: object

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **additional: Tuple or list of fields to include *in addition* to the** explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **include: Dictionary of additional fields to include in the schema. It is** usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **exclude: Tuple or list of fields to exclude in the serialized result.** Nested fields can be represented with dot delimiters.
- `dateformat`: Default format for *Date* <*fields.Date*> fields.
- `datetimeformat`: Default format for *DateTime* <*fields.DateTime*> fields.
- `timeformat`: Default format for *Time* <*fields.Time*> fields.

- **render_module: Module to use for *loads* <*Schema.loads*> and *dumps* <*Schema.dumps*>.**
  Defaults to *json* from the standard library.
- **ordered: If *True*, order serialization output according to the**
  order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **index_errors: If *True*, errors dictionaries will include the index**
  of invalid items in a collection.
- load_only: Tuple or list of fields to exclude from serialized results.
- dump_only: Tuple or list of fields to exclude from deserialization
- **unknown: Whether to exclude, include, or raise an error for unknown**
  fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **register: Whether to register the *Schema* with marshmallow's internal**
  class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

OPTIONS_CLASS

alias of SchemaOpts

TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}

property dict_class: type

dump(*obj: Any*, *, *many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A ValidationError is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dump_fields: Dict[str, ma_fields.Field]

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/nitpick_section.html#comma-separated-values.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, *, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, *, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

load_fields: `Dict[str, ma_fields.Field]`

loads(*json_data: str*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: `SchemaOpts = <marshmallow.schema.SchemaOpts object>`

property set_class: `type`

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns
A dictionary of validation errors.

New in version 1.1.0.

class nitpick.schemas.**NitpickFilesSectionSchema**(***, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: *BaseNitpickSchema*

Validation schema for the `[nitpick.files]` section on the style file.

class Meta

Bases: `object`

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **additional: Tuple or list of fields to include *in addition* to the**
  explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **include: Dictionary of additional fields to include in the schema. It is**
  usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **exclude: Tuple or list of fields to exclude in the serialized result.**
  Nested fields can be represented with dot delimiters.
- `dateformat`: Default format for *Date <fields.Date>* fields.
- `datetimeformat`: Default format for *DateTime <fields.DateTime>* fields.
- `timeformat`: Default format for *Time <fields.Time>* fields.
- **render_module: Module to use for *loads <Schema.loads>* and *dumps <Schema.dumps>*.**
  Defaults to *json* from the standard library.

- **ordered: If *True*, order serialization output according to the**
  order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **index_errors: If *True*, errors dictionaries will include the index**
  of invalid items in a collection.
- `load_only`: Tuple or list of fields to exclude from serialized results.
- `dump_only`: Tuple or list of fields to exclude from deserialization
- **unknown: Whether to exclude, include, or raise an error for unknown**
  fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **register: Whether to register the *Schema* with marshmallow's internal**
  class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

OPTIONS_CLASS

alias of `SchemaOpts`

TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}

property dict_class: type

dump(*obj: Any*, *, *many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns

Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dump_fields: Dict[str, ma_fields.Field]

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.

- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown file. See https://nitpick.rtfd.io/en/latest/nitpick_section.html#nitpick-files.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, *, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, *, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

load_fields: `Dict[str, ma_fields.Field]`

loads(*json_data: str*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: `SchemaOpts = <marshmallow.schema.SchemaOpts object>`

property set_class: `type`

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns
A dictionary of validation errors.

New in version 1.1.0.

class nitpick.schemas.**NitpickMetaSchema**(***, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: *BaseNitpickSchema*

Meta info about a specific TOML style file.

class Meta

Bases: `object`

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **additional: Tuple or list of fields to include *in addition* to the** explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **include: Dictionary of additional fields to include in the schema. It is** usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **exclude: Tuple or list of fields to exclude in the serialized result.** Nested fields can be represented with dot delimiters.
- `dateformat`: Default format for *Date* <*fields.Date*> fields.
- `datetimeformat`: Default format for *DateTime* <*fields.DateTime*> fields.
- `timeformat`: Default format for *Time* <*fields.Time*> fields.
- **render_module: Module to use for *loads* <*Schema.loads*> and *dumps* <*Schema.dumps*>.** Defaults to *json* from the standard library.

- **ordered: If *True*, order serialization output according to the**
  order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **index_errors: If *True*, errors dictionaries will include the index**
  of invalid items in a collection.
- `load_only`: Tuple or list of fields to exclude from serialized results.
- `dump_only`: Tuple or list of fields to exclude from deserialization
- **unknown: Whether to exclude, include, or raise an error for unknown**
  fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **register: Whether to register the *Schema* with marshmallow's internal**
  class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

OPTIONS_CLASS

alias of `SchemaOpts`

TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}

property dict_class: type

dump(*obj: Any*, *, *many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns

Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dump_fields: Dict[str, ma_fields.Field]

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.

- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/nitpick_section.html.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, *, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, *, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

load_fields: `Dict[str, ma_fields.Field]`

loads(*json_data: str*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: `SchemaOpts = <marshmallow.schema.SchemaOpts object>`

property set_class: `type`

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns
A dictionary of validation errors.

New in version 1.1.0.

class nitpick.schemas.**NitpickSectionSchema**(***, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: *BaseNitpickSchema*

Validation schema for the `[nitpick]` section on the style file.

class Meta

Bases: `object`

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **additional: Tuple or list of fields to include *in addition* to the**
  explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **include: Dictionary of additional fields to include in the schema. It is**
  usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **exclude: Tuple or list of fields to exclude in the serialized result.**
  Nested fields can be represented with dot delimiters.
- `dateformat`: Default format for *Date* <*fields.Date*> fields.
- `datetimeformat`: Default format for *DateTime* <*fields.DateTime*> fields.
- `timeformat`: Default format for *Time* <*fields.Time*> fields.
- **render_module: Module to use for *loads* <*Schema.loads*> and *dumps* <*Schema.dumps*>.**
  Defaults to *json* from the standard library.

- **ordered: If *True*, order serialization output according to the**
  order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **index_errors: If *True*, errors dictionaries will include the index**
  of invalid items in a collection.
- `load_only`: Tuple or list of fields to exclude from serialized results.
- `dump_only`: Tuple or list of fields to exclude from deserialization
- **unknown: Whether to exclude, include, or raise an error for unknown**
  fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **register: Whether to register the *Schema* with marshmallow's internal**
  class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

OPTIONS_CLASS

alias of `SchemaOpts`

TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}

property dict_class: type

dump(*obj: Any*, *, *many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns

Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dump_fields: Dict[str, ma_fields.Field]

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.

- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/nitpick_section.html.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, *, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, *, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

load_fields: `Dict[str, ma_fields.Field]`

loads(*json_data: str*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: `SchemaOpts = <marshmallow.schema.SchemaOpts object>`

property set_class: `type`

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns
A dictionary of validation errors.

New in version 1.1.0.

class nitpick.schemas.**NitpickStylesSectionSchema**(***, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: *BaseNitpickSchema*

Validation schema for the `[nitpick.styles]` section on the style file.

class Meta

Bases: `object`

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **additional: Tuple or list of fields to include *in addition* to the** explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **include: Dictionary of additional fields to include in the schema. It is** usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **exclude: Tuple or list of fields to exclude in the serialized result.** Nested fields can be represented with dot delimiters.
- `dateformat`: Default format for *Date <fields.Date>* fields.
- `datetimeformat`: Default format for *DateTime <fields.DateTime>* fields.
- `timeformat`: Default format for *Time <fields.Time>* fields.
- **render_module: Module to use for *loads <Schema.loads>* and *dumps <Schema.dumps>*.** Defaults to *json* from the standard library.

- **ordered: If *True*, order serialization output according to the**
  order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **index_errors: If *True*, errors dictionaries will include the index**
  of invalid items in a collection.
- `load_only`: Tuple or list of fields to exclude from serialized results.
- `dump_only`: Tuple or list of fields to exclude from deserialization
- **unknown: Whether to exclude, include, or raise an error for unknown**
  fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **register: Whether to register the *Schema* with marshmallow's internal**
  class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

OPTIONS_CLASS

alias of `SchemaOpts`

TYPE_MAPPING: Dict[type, Type[ma_fields.Field]] = {<class 'str'>: <class 'marshmallow.fields.String'>, <class 'bytes'>: <class 'marshmallow.fields.String'>, <class 'datetime.datetime'>: <class 'marshmallow.fields.DateTime'>, <class 'float'>: <class 'marshmallow.fields.Float'>, <class 'bool'>: <class 'marshmallow.fields.Boolean'>, <class 'tuple'>: <class 'marshmallow.fields.Raw'>, <class 'list'>: <class 'marshmallow.fields.Raw'>, <class 'set'>: <class 'marshmallow.fields.Raw'>, <class 'int'>: <class 'marshmallow.fields.Integer'>, <class 'uuid.UUID'>: <class 'marshmallow.fields.UUID'>, <class 'datetime.time'>: <class 'marshmallow.fields.Time'>, <class 'datetime.date'>: <class 'marshmallow.fields.Date'>, <class 'datetime.timedelta'>: <class 'marshmallow.fields.TimeDelta'>, <class 'decimal.Decimal'>: <class 'marshmallow.fields.Decimal'>}

property dict_class: type

dump(*obj: Any*, *, *many: bool | None = None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns

Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dump_fields: Dict[str, ma_fields.Field]

dumps(*obj: Any*, **args*, *many: bool | None = None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.

- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns
A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

error_messages: Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/nitpick_section.html#nitpick-styles.'}

Overrides for default schema-level error messages

fields: Dict[str, ma_fields.Field]

Dictionary mapping field_names -> `Field` objects

classmethod from_dict(*fields: dict[str, ma_fields.Field | type]*, *, *name: str = 'GeneratedSchema'*) → type

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

Parameters

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

get_attribute(*obj: Any*, *attr: str*, *default: Any*)

Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

handle_error(*error: ValidationError*, *data: Any*, *, *many: bool*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.
- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

load_fields: `Dict[str, ma_fields.Field]`

loads(*json_data: str*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*, *unknown: str | None = None*, ***kwargs*)

Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns
Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

on_bind_field(*field_name: str*, *field_obj:* Field) → None

Hook to modify a field when it is bound to the *Schema*.

No-op by default.

opts: `SchemaOpts = <marshmallow.schema.SchemaOpts object>`

property set_class: `type`

validate(*data: Mapping[str, Any] | Iterable[Mapping[str, Any]]*, ***, *many: bool | None = None*, *partial: bool | types.StrSequenceOrSet | None = None*) → dict[str, list[str]]

Validate *data* against the schema, returning a dictionary of validation errors.

Parameters

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns
A dictionary of validation errors.

New in version 1.1.0.

nitpick.schemas.**flatten_marshmallow_errors**(*errors: Dict*) → str

Flatten Marshmallow errors to a string.

nitpick.schemas.**help_message**(*sentence: str*, *help_page: str*) → str

Show help with the documentation URL on validation errors.

nitpick.typedefs module

Type definitions.

nitpick.violations module

Violation codes.

Name inspired by flake8's violations.

class nitpick.violations.**Fuss**(*fixed: bool*, *filename: str*, *code: int*, *message: str*, *suggestion: str = ''*, *lineno: int = 1*)

Bases: `object`

Nitpick makes a fuss when configuration doesn't match.

Fields inspired on `SyntaxError` and pyflakes.messages.Message.

code: `int`

property colored_suggestion: `str`

Suggestion with color.

filename: `str`

fixed: `bool`

lineno: `int = 1`

message: `str`

property pretty: `str`

Message to be used on the CLI.

suggestion: *str* = ''

class nitpick.violations.**ProjectViolations**(*value*)

Bases: *ViolationEnum*

Project initialization violations.

FILE_SHOULD_BE_DELETED = (104, ' should be deleted{extra}')

MINIMUM_VERSION = (203, "The style file you're using requires {project}>={expected} (you have {actual}). Please upgrade")

MISSING_FILE = (103, ' should exist{extra}')

NO_PYTHON_FILE = (102, 'No Python file was found on the root dir and subdir of {root!r}')

NO_ROOT_DIR = (101, "No root directory detected. Create a configuration file (.nitpick.toml, pyproject.toml) manually, or run 'nitpick init'. See https://nitpick.rtfd.io/en/latest/configuration.html")

class nitpick.violations.**Reporter**(*info:* FileInfo | *None* = *None*, *violation_base_code:* *int* = *0*)

Bases: object

Error reporter.

fixed: *int* = 0

classmethod get_counts() → str

String representation with error counts and emojis.

classmethod increment(*fixed=False*)

Increment the fixed ou manual count.

make_fuss(*violation:* ViolationEnum, *suggestion:* *str* = '', *fixed=False*, ***kwargs*) → *Fuss*

Make a fuss.

manual: *int* = 0

classmethod reset()

Reset the counters.

class nitpick.violations.**SharedViolations**(*value*)

Bases: *ViolationEnum*

Shared violations used by all plugins.

CREATE_FILE = (1, ' was not found', True)

CREATE_FILE_WITH_SUGGESTION = (1, ' was not found. Create it with this content:', True)

DELETE_FILE = (2, ' should be deleted', True)

DIFFERENT_VALUES = (9, '{prefix} has different values. Use this:', True)

MISSING_VALUES = (8, '{prefix} has missing values:', True)

class nitpick.violations.**StyleViolations**(*value*)

Bases: *ViolationEnum*

Style violations.

INVALID_CONFIG = (1, ' has an incorrect style. Invalid config:')

INVALID_DATA_TOOL_NITPICK = (1, ' has an incorrect style. Invalid data in [{section}]:')

INVALID_TOML = (1, ' has an incorrect style. Invalid TOML{exception}')

class nitpick.violations.**ViolationEnum**(*value*)

Bases: Enum

Base enum with violation codes and messages.

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

FOURTEEN

TO DO LIST

PYTHON MODULE INDEX

n

INDEX

D

E

F

G

M

N

O

P

Q

R

S

T

U

V

W

X

Y