# nitpick Documentation

***Release 0.28.0***

**W. Augusto Andreoli**

**Oct 27, 2021**

# 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 style 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: v0.28.0
    hooks:
      - id: 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 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 *some pre-configured examples*, and copy/paste/change configuration from them.

## 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 Breaking style 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.

### 2.5.1 `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:

```
[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@v0.28.0/nitpick-style.toml"
# or
style = "gh://andreoliwa/nitpick@v0.28.0/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`):

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

(continues on next page)

(continued from previous page)

```
# 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.

```
[tool.nitpick]
style = "https://github.com/andreoliwa/nitpick/blob/v0.28.0/nitpick-style.toml"
```

Or use the raw GitHub URL directly:

```
[tool.nitpick]
style = "https://raw.githubusercontent.com/andreoliwa/nitpick/v0.28.0/nitpick-style.toml"
```

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

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

## 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://`:

```
[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:

```
[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",
]
```

CHAPTER

FOUR

# EXAMPLES

If you don't *configure your own style*, those are some of the defaults that will be applied.

All TOML configs below are taken from the default style file.

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

## 4.1 Absent files

Contents of styles/absent-files.toml:

```toml
[nitpick.files.absent]
"requirements.txt" = "Install poetry, run 'poetry init' to create pyproject.toml, and␣
↪move dependencies to it"
".isort.cfg" = "Move values to setup.cfg, section [isort]"
"Pipfile" = "Use pyproject.toml instead"
"Pipfile.lock" = "Use pyproject.toml instead"
".venv" = ""
".pyup.yml" = "Configure safety instead: https://github.com/pyupio/safety#using-safety-
↪with-a-ci-service"
```

## 4.2 black

Contents of styles/black.toml:

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

[[".pre-commit-config.yaml".repos]]
yaml = """
  - repo: https://github.com/psf/black
    hooks:
      - id: black
        args: [--safe, --quiet]
  - repo: https://github.com/asottile/blacken-docs
    hooks:
      - id: blacken-docs
        additional_dependencies: [black==21.5b2]
```

(continues on next page)

(continued from previous page)

```
"""
# TODO The toml library has issues loading arrays with multiline strings:
#  https://github.com/uiri/toml/issues/123
#  https://github.com/uiri/toml/issues/230
#  If they are fixed one day, remove this 'yaml' key and use only a 'repos' list with a␣
↪single element:
#[".pre-commit-config.yaml"]
#repos = ["""
#<YAML goes here>
#"""]
```

## 4.3 EditorConfig

Contents of styles/editorconfig.toml:

```
# http://editorconfig.org/

[".editorconfig"]
# top-most EditorConfig file
root = true

[".editorconfig"."*"]
# Unix-style newlines with a newline ending every file
end_of_line = "lf"
insert_final_newline = true
indent_style = "space"
indent_size = 4

# Whitespace at the end of lines
trim_trailing_whitespace = true

# Matches multiple files with brace expansion notation
# Set default charset
[".editorconfig"."*.{js,json}"]
charset = "utf-8"
indent_size = 2

[".editorconfig"."*.py"]
charset = "utf-8"

[".editorconfig"."*.{yml,yaml,md,rb}"]
indent_size = 2

[".editorconfig".Makefile]
indent_style = "tab"
```

## 4.4 flake8

Contents of styles/flake8.toml:

```
["setup.cfg".flake8]
# http://www.pydocstyle.org/en/2.1.1/error_codes.html
# Ignoring W503: https://github.com/python/black#line-breaks--binary-operators
# Ignoring "D202 No blank lines allowed after function docstring": black inserts a blank␣
↪line.
ignore = "D107,D202,D203,D401,E203,E402,E501,W503"
max-line-length = 120
inline-quotes = "double"
exclude = ".tox,build"

# Nitpick recommends those plugins as part of the style, but doesn't install them␣
↪automatically as before.
# This way, the developer has the choice of overriding this style, instead of having␣
↪lots of plugins installed
# without being asked.
[[".pre-commit-config.yaml".repos]]
yaml = """
  - repo: https://github.com/PyCQA/flake8
    hooks:
      - id: flake8
        additional_dependencies:
          [
            flake8-blind-except,
            flake8-bugbear,
            flake8-comprehensions,
            flake8-debugger,
            flake8-docstrings,
            flake8-isort,
            flake8-polyfill,
            flake8-pytest,
            flake8-quotes,
            flake8-typing-imports,
            yesqa,
          ]
"""
# TODO suggest nitpick for external repos
```

## 4.5 IPython

Contents of styles/ipython.toml:

```
["pyproject.toml".tool.poetry.dev-dependencies]
ipython = "*"
ipdb = "*"
```

## 4.6 isort

Contents of styles/isort.toml:

```
["setup.cfg".isort]
line_length = 120
skip = ".tox,build"
known_first_party = "tests"

# The configuration below is needed for compatibility with black.
# https://github.com/python/black#how-black-wraps-lines
# https://github.com/PyCQA/isort#multi-line-output-modes
multi_line_output = 3
include_trailing_comma = true
force_grid_wrap = 0
combine_as_imports = true

[[".pre-commit-config.yaml".repos]]
yaml = """
  - repo: https://github.com/PyCQA/isort
    hooks:
      - id: isort
"""
```

## 4.7 mypy

Contents of styles/mypy.toml:

```
# https://mypy.readthedocs.io/en/latest/config_file.html
["setup.cfg".mypy]
ignore_missing_imports = true

# Do not follow imports (except for ones found in typeshed)
follow_imports = "skip"

# Treat Optional per PEP 484
strict_optional = true

# Ensure all execution paths are returning
warn_no_return = true

# Lint-style cleanliness for typing
warn_redundant_casts = true
warn_unused_ignores = true

[[".pre-commit-config.yaml".repos]]
yaml = """
  - repo: https://github.com/pre-commit/mirrors-mypy
    hooks:
      - id: mypy
"""
```

## 4.8 package.json

Contents of styles/package-json.toml:

```
["package.json"]
contains_keys = ["name", "version", "repository.type", "repository.url", "release.plugins
↪"]

["package.json".contains_json]
commitlint = """
  {
    "extends": [
      "@commitlint/config-conventional"
    ]
  }
"""
```

## 4.9 Poetry

Contents of styles/poetry.toml:

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

## 4.10 Bash

Contents of styles/pre-commit/bash.toml:

```
[[".pre-commit-config.yaml".repos]]
yaml = """
  - repo: https://github.com/openstack/bashate
    hooks:
      - id: bashate
"""
```

## 4.11 commitlint

Contents of styles/pre-commit/commitlint.toml:

```
[[".pre-commit-config.yaml".repos]]
yaml = """
  - repo: https://github.com/alessandrojcm/commitlint-pre-commit-hook
    rev: v5.0.0
    hooks:
      - id: commitlint
        stages: [commit-msg]
```

(continues on next page)

(continued from previous page)

```
        additional_dependencies: ['@commitlint/config-conventional']
"""
```

## 4.12 pre-commit (hooks)

Contents of styles/pre-commit/general.toml:

```
[[".pre-commit-config.yaml".repos]]
yaml = """
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.0.1
    hooks:
      - id: end-of-file-fixer
      - id: trailing-whitespace
"""
```

## 4.13 pre-commit (main)

Contents of styles/pre-commit/main.toml:

```
# See https://pre-commit.com for more information
# See https://pre-commit.com/hooks.html for more hooks

[nitpick.files.present]
".pre-commit-config.yaml" = "Create the file with the contents below, then run 'pre-
↪commit install'"
```

## 4.14 pre-commit (Python hooks)

Contents of styles/pre-commit/python.toml:

```
[[".pre-commit-config.yaml".repos]]
yaml = """
  - repo: https://github.com/pre-commit/pygrep-hooks
    hooks:
      - id: python-check-blanket-noqa
      - id: python-check-mock-methods
      - id: python-no-eval
      - id: python-no-log-warn
      - id: rst-backticks
  - repo: https://github.com/pre-commit/pre-commit-hooks
    hooks:
      - id: debug-statements
  - repo: https://github.com/asottile/pyupgrade
    hooks:
      - id: pyupgrade
"""
```

## 4.15 Pylint

Contents of styles/pylint.toml:

```
["pyproject.toml".tool.poetry.dependencies]
pylint = {version = "*", optional = true}

["pyproject.toml".tool.poetry.extras]
lint = ["pylint"]

[".pylintrc".MASTER]
# Use multiple processes to speed up Pylint.
jobs = 1

[".pylintrc".REPORTS]
# Set the output format. Available formats are text, parseable, colorized, msvs (visual␣
↪studio) and html.
# You can also give a reporter class, eg mypackage.mymodule.MyReporterClass.
output-format = "colorized"

[".pylintrc"."MESSAGES CONTROL"]
# TODO: deal with character separated INI options in https://github.com/andreoliwa/
↪nitpick/issues/271
#  The "nitpick.files" section doesn't work out of the box for .pylintrc:
#  [nitpick.files.".pylintrc"]
#  comma_separated_values = ["MESSAGES CONTROL.disable"]
#  This syntax will be deprecated anyway, so I won't make it work now
# Configurations for the black formatter
#disable = "bad-continuation,bad-whitespace,fixme,cyclic-import"

[".pylintrc".BASIC]
# List of builtins function names that should not be used, separated by a comma
bad-functions = "map,filter"
# Good variable names which should always be accepted, separated by a comma
good-names = "i,j,k,e,ex,Run,_,id,rv"

[".pylintrc".FORMAT]
# Maximum number of characters on a single line.
max-line-length = 120
# Maximum number of lines in a module
max-module-lines = 1000
# TODO: deal with empty options (strings with spaces and quotes); maybe it's a␣
↪ConfigParser/ConfigUpdater thing
# String used as indentation unit. This is usually " " (4 spaces) or "\t" (1 tab).
#indent-string = "    "
# Number of spaces of indent required inside a hanging or continued line.
indent-after-paren = 4

[".pylintrc".SIMILARITIES]
# Minimum lines number of a similarity.
min-similarity-lines = 4
# Ignore comments when computing similarities.
ignore-comments = "yes"
```

(continues on next page)

(continued from previous page)

```
# Ignore docstrings when computing similarities.
ignore-docstrings = "yes"
# Ignore imports when computing similarities.
ignore-imports = "no"

[".pylintrc".VARIABLES]
# A regular expression matching the name of dummy variables (i.e. expectedly not used).
dummy-variables-rgx = "_$|dummy"
```

## 4.16 Python 3.6

Contents of styles/python36.toml:

```
["pyproject.toml".tool.poetry.dependencies]
python = "^3.6"
```

## 4.17 Python 3.7

Contents of styles/python37.toml:

```
["pyproject.toml".tool.poetry.dependencies]
python = "^3.7"
```

## 4.18 Python 3.8

Contents of styles/python38.toml:

```
["pyproject.toml".tool.poetry.dependencies]
python = "^3.8"
```

## 4.19 Python 3.9

Contents of styles/python39.toml:

```
["pyproject.toml".tool.poetry.dependencies]
python = "^3.9"
```

## 4.20 tox

Contents of styles/tox.toml:

```
["tox.ini".tox]
# https://tox.readthedocs.io/en/latest/config.html
isolated_build = true

["tox.ini".testenv]
description = "Run tests with pytest and coverage"
extras = "test"

["tox.ini"."coverage:run"]
# https://coverage.readthedocs.io/en/latest/config.html#run
branch = true
parallel = true
source = "src/"
# TODO: deal with multiline INI values in https://github.com/andreoliwa/nitpick/issues/
↪271
#omit = """tests/*
#.tox/*
#*/pypoetry/virtualenvs/*
#"""
# This config is needed by https://github.com/marketplace/actions/coveralls-python#usage
relative_files = true

["tox.ini"."coverage:report"]
# https://coverage.readthedocs.io/en/latest/config.html#report
show_missing = true
precision = 2
skip_covered = true
skip_empty = true
sort = "Cover"
```

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:

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

The styles will be merged following the sequence in the list.

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)
  --help                   Show this message and exit.

Commands:
  check  Don't modify files, just print the differences.
  fix    Fix files, modifying them directly.
  init   Create a configuration file if it doesn't exist already.
  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]

  Create a 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

Nitpick tries to find the root dir of the project using some hardcoded assumptions.

1. Starting from the current working directory, it will search for files that are usually in the root of a project:

- `.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)

1. If none of these root files were found, search for `manage.py`. On Django projects, it can be in another dir inside the root dir (issue 21).
2. If multiple roots are found, get the top one in the dir tree.

## 7.3 Main Python file

After finding 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 .pre-commit-config.yaml

Enforce configuration for .pre-commit-config.yaml.

Style example: *the default pre-commit hooks*.

## 8.2 INI files

Enforce config on 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*.

## 8.3 JSON files

Enforce configurations for any JSON file.

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

## 8.4 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"
```

## 8.5 TOML files

Enforce config on 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*.

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-py3.6/lib/python3.6/
↪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 there.

## 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[pathlib.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[*nitpick.violations.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*, *fix=True*) → Iterator[*nitpick.violations.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.
- **fix** – Flag to modify files, if the plugin supports it (default: True).

**Returns** Fuss generator.

**init**(*project_root: Optional[Union[pathlib.Path, str]] = None*, *offline: Optional[bool] = None*) → *nitpick.core.Nitpick*

Initialize attributes of the singleton.

**project:** *nitpick.project.Project*

**run**(**partial_names: str*, *fix=False*) → Iterator[*nitpick.violations.Fuss*]

Run Nitpick.

**Parameters**

- **partial_names** – Names of the files to enforce configs for.

- **fix** – Flag to modify files, if the plugin supports it (default: True).

**Returns** Fuss generator.

**classmethod singleton**() → *nitpick.core.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:* nitpick.plugins.info.FileInfo, *expected_config: Dict[str, Any]*, *fix=False*)

Bases: object

Base class for Nitpick plugins.

**Parameters**

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

**can_fix: bool = False**
Can this plugin modify files directly?

**abstract enforce_rules**() → Iterator[*nitpick.violations.Fuss*]
Enforce rules for this file. It must be overridden by inherited classes if needed.

**entry_point**() → Iterator[*nitpick.violations.Fuss*]
Entry point of the Nitpick plugin.

**filename = ''**

**classmethod get_compiled_jmespath_filenames**()
Return a compiled JMESPath expression for file names, using the class name as part of the key.

**identify_tags: Set[str] = {}**
Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

**init**()
Hook for plugin initialization after the instance was created.

**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.

**skip_empty_suggestion = False**

**validation_schema: Optional[marshmallow.schema.Schema] = 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**

**warn_missing_different**(*comparison:* nitpick.formats.Comparison, *prefix: str = ''*) → Iterator[*nitpick.violations.Fuss*]
Warn about missing and different keys.

**write_file**(*file_exists: bool*) → Optional[*nitpick.violations.Fuss*]
Hook to write the new file when fix mode is on. Should be used by inherited classes.

### nitpick.plugins.info module

Info needed by the plugins.

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

File information needed by the plugin.

**classmethod create**(*project:* nitpick.project.Project, *path_from_root: str*) → *nitpick.plugins.info.FileInfo*
Clean the file name and get its tags.

**path_from_root: str**

**project: *nitpick.project.Project***

**tags: Set[str]**

### nitpick.plugins.ini module

INI files.

**class** nitpick.plugins.ini.**IniPlugin**(*info:* nitpick.plugins.info.FileInfo, *expected_config: Dict[str, Any]*, *fix=False*)
Bases: *nitpick.plugins.base.NitpickPlugin*

Enforce config on 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: collections.OrderedDict*) → None
Add new options before a blank line in the end of the section.

**can_fix:** bool = True
Can this plugin modify files directly?

**comma_separated_values:** Set[str]

**compare_different_keys**(*section*, *key*, *raw_actual: Any*, *raw_expected: Any*) → Iterator[*nitpick.violations.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[*nitpick.violations.Fuss*]
Enforce sections and keys with comma-separated values. The values might contain spaces.

**enforce_missing_sections**() → Iterator[*nitpick.violations.Fuss*]
Enforce missing sections.

**enforce_rules**() → Iterator[*nitpick.violations.Fuss*]
Enforce rules on missing sections and missing key/value pairs in an INI file.

**enforce_section**(*section: str*) → Iterator[*nitpick.violations.Fuss*]
Enforce rules for a section.

**entry_point**() → Iterator[*nitpick.violations.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** = ''

**classmethod get_compiled_jmespath_filenames**()
Return a compiled JMESPath expression for file names, using the class name as part of the key.

**static get_example_cfg**(*parser: configparser.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.

**init**()
Post initialization after the instance was created.

**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.

**show_missing_keys**(*section: str*, *values: List[Tuple[str, Any]]*) → Iterator[*nitpick.violations.Fuss*]
Show the keys that are not present in a section.

**skip_empty_suggestion = False**

**updater: configupdater.configupdater.ConfigUpdater**

**validation_schema: Optional[Schema] = 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**

**warn_missing_different**(*comparison:* nitpick.formats.Comparison, *prefix: str = ''*) → Iterator[*nitpick.violations.Fuss*]
Warn about missing and different keys.

**write_file**(*file_exists: bool*) → Optional[*nitpick.violations.Fuss*]
Write the new file.

class nitpick.plugins.ini.**Violations**(*value*)
Bases: *nitpick.violations.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:* nitpick.plugins.info.FileInfo) → Optional[Type[*nitpick.plugins.base.NitpickPlugin*]]
Handle INI files.

nitpick.plugins.ini.**plugin_class**() → Type[*nitpick.plugins.base.NitpickPlugin*]
Handle INI files.

### nitpick.plugins.json module

JSON files.

class nitpick.plugins.json.**JSONFileSchema**(*, *only: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Optional[Dict] = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: Optional[str] = None*)

Bases: *nitpick.schemas.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 marshmallow.schema.SchemaOpts

**TYPE_MAPPING: typing.Dict[type, typing.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: Optional[bool] = 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: Optional[bool] = 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: typing.Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/nitpick_section.html.'}**
Overrides for default schema-level error messages

**fields: typing.Dict[str, ma_fields.Field]**
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.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: marshmallow.exceptions.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:* marshmallow.fields.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, *, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = 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:* nitpick.plugins.info.FileInfo, *expected_config: Dict[str, Any]*, *fix=False*)
Bases: *nitpick.plugins.base.NitpickPlugin*

Enforce configurations for any JSON file.

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

**SOME_VALUE_PLACEHOLDER = '<some value here>'**

**can_fix: bool = False**
Can this plugin modify files directly?

**dirty:** bool

**enforce_rules**() → Iterator[*nitpick.violations.Fuss*]
Enforce rules for missing keys and JSON content.

**entry_point**() → Iterator[*nitpick.violations.Fuss*]
Entry point of the Nitpick plugin.

**expected_config:** JsonDict

**file_path:** Path

**filename** = ''

**classmethod get_compiled_jmespath_filenames**()
Return a compiled JMESPath expression for file names, using the class name as part of the key.

**get_suggested_json**(*raw_actual: Optional[Dict[str, Any]] = None*) → Dict[str, Any]
Return the suggested JSON based on actual values.

**identify_tags:** Set[str] = {'json'}
Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

**init**()
Hook for plugin initialization after the instance was created.

**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.

**skip_empty_suggestion** = False

**validation_schema**
alias of *nitpick.plugins.json.JSONFileSchema*

**violation_base_code:** int = 340

**warn_missing_different**(*comparison:* nitpick.formats.Comparison, *prefix: str = ''*) → Iterator[*nitpick.violations.Fuss*]
Warn about missing and different keys.

**write_file**(*file_exists: bool*) → Optional[*nitpick.violations.Fuss*]
Hook to write the new file when fix mode is on. Should be used by inherited classes.

**class** nitpick.plugins.json.**Violations**(*value*)
Bases: *nitpick.violations.ViolationEnum*

Violations for this plugin.

**MISSING_KEYS** = (348, ' has missing keys:')

nitpick.plugins.json.**can_handle**(*info:* nitpick.plugins.info.FileInfo) → Optional[Type[*nitpick.plugins.base.NitpickPlugin*]]
Handle JSON files.

nitpick.plugins.json.**plugin_class**() → Type[*nitpick.plugins.base.NitpickPlugin*]
Handle JSON files.

### nitpick.plugins.pre_commit module

Enforce configuration for .pre-commit-config.yaml.

**class** nitpick.plugins.pre_commit.**PreCommitHook**(*repo: str*, *hook_id: str*, *yaml:* nitpick.formats.YAMLFormat)

Bases: object

A pre-commit hook.

Method generated by attrs for class PreCommitHook.

**classmethod get_all_hooks_from**(*str_or_yaml: Union[str, ruamel.yaml.comments.CommentedSeq, ruamel.yaml.comments.CommentedMap]*)
Get all hooks from a YAML string. Split the string in hooks and copy the repo info for each.

**property key_value_pair: Tuple[str, *nitpick.plugins.pre_commit.PreCommitHook*]**
Key/value pair to be used as a dict item.

**property single_hook: Dict[str, Any]**
Return only a single hook instead of a list.

**property unique_key: str**
Unique key of this hook, to be used in a dict.

**class** nitpick.plugins.pre_commit.**PreCommitPlugin**(*info:* nitpick.plugins.info.FileInfo, *expected_config: Dict[str, Any]*, *fix=False*)

Bases: *nitpick.plugins.base.NitpickPlugin*

Enforce configuration for .pre-commit-config.yaml.

Style example: *the default pre-commit hooks*.

**actual_hooks: Dict[str, *nitpick.plugins.pre_commit.PreCommitHook*] = {}**

**actual_hooks_by_index: List[str] = []**

**actual_hooks_by_key: Dict[str, int] = {}**

**actual_yaml: *nitpick.formats.YAMLFormat***

**can_fix: bool = False**
Can this plugin modify files directly?

**dirty: bool**

**enforce_hooks**() → Iterator[*nitpick.violations.Fuss*]
Enforce the repositories configured in pre-commit.

**enforce_repo_block**(*expected_repo_block: collections.OrderedDict*) → Iterator[*nitpick.violations.Fuss*]
Enforce a repo with a YAML string configuration.

**enforce_repo_old_format**(*index: int*, *repo_data: collections.OrderedDict*) → Iterator[*nitpick.violations.Fuss*]
Enforce repos using the old deprecated format with `hooks` and `repo` keys.

**enforce_rules**() → Iterator[*nitpick.violations.Fuss*]
Enforce rules for the pre-commit hooks.

**entry_point**() → Iterator[*nitpick.violations.Fuss*]
Entry point of the Nitpick plugin.

**expected_config: JsonDict**

**file_path: Path**

**filename = '.pre-commit-config.yaml'**

**static format_hook**(*expected_dict*) → str
Format the hook so it's easy to copy and paste it to the .yaml file: ID goes first, indent with spaces.

**classmethod get_compiled_jmespath_filenames**()
Return a compiled JMESPath expression for file names, using the class name as part of the key.

**identify_tags: Set[str] = {}**
Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

**init**()
Hook for plugin initialization after the instance was created.

**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.

**skip_empty_suggestion = False**

**validation_schema: Optional[Schema] = 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 = 330**

**warn_missing_different**(*comparison:* nitpick.formats.Comparison, *prefix: str = ''*) → Iterator[*nitpick.violations.Fuss*]
Warn about missing and different keys.

**write_file**(*file_exists: bool*) → Optional[*nitpick.violations.Fuss*]
Hook to write the new file when fix mode is on. Should be used by inherited classes.

**class** nitpick.plugins.pre_commit.**Violations**(*value*)
Bases: *nitpick.violations.ViolationEnum*

Violations for this plugin.

**HOOK_NOT_FOUND = (332, ': hook {id!r} not found. Use this:')**

**MISSING_HOOK_WITH_ID = (337, ': missing hook with id {id!r}:')**

**MISSING_KEY_IN_HOOK = (336, ': style file is missing {key!r} in hook:')**

**MISSING_KEY_IN_REPO = (334, ': missing {key!r} in repo {repo!r}')**

**NO_ROOT_KEY = (331, " doesn't have the 'repos' root key")**

**REPO_DOES_NOT_EXIST = (333, ': repo {repo!r} does not exist under {key!r}')**

**STYLE_FILE_MISSING_NAME = (335, ': style file is missing {key!r} in repo {repo!r}')**

**STYLE_MISSING_INDEX = (332, ': style file is missing {key!r} key in repo #{index}')**

nitpick.plugins.pre_commit.**can_handle**(*info:* nitpick.plugins.info.FileInfo) → Optional[Type[*nitpick.plugins.base.NitpickPlugin*]]
Handle pre-commit config file.

nitpick.plugins.pre_commit.**plugin_class**() → Type[*nitpick.plugins.base.NitpickPlugin*]
Handle pre-commit config file.

### nitpick.plugins.text module

Text files.

class nitpick.plugins.text.**TextItemSchema**(*, *only: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Optional[Dict] = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: Optional[str] = None*)

Bases: marshmallow.schema.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 marshmallow.schema.SchemaOpts

**TYPE_MAPPING: typing.Dict[type, typing.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: Optional[bool] = 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: Optional[bool] = 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: typing.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: typing.Dict[str, ma_fields.Field]**
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.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: marshmallow.exceptions.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:* marshmallow.fields.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, *, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = 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:* nitpick.plugins.info.FileInfo, *expected_config: Dict[str, Any]*, *fix=False*)
Bases: *nitpick.plugins.base.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"
```

(continues on next page)

(continued from previous page)

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

**can_fix: bool = False**
Can this plugin modify files directly?

**dirty: bool**

**enforce_rules**() → Iterator[*nitpick.violations.Fuss*]
Enforce rules for missing lines.

**entry_point**() → Iterator[*nitpick.violations.Fuss*]
Entry point of the Nitpick plugin.

**expected_config: JsonDict**

**file_path: Path**

**filename = ''**

**classmethod get_compiled_jmespath_filenames**()
Return a compiled JMESPath expression for file names, using the class name as part of the key.

**identify_tags: Set[str] = {'text'}**
Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

**init**()
Hook for plugin initialization after the instance was created.

**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.

**skip_empty_suggestion = True**

**validation_schema**
alias of *nitpick.plugins.text.TextSchema*

**violation_base_code: int = 350**

**warn_missing_different**(*comparison:* nitpick.formats.Comparison, *prefix: str = ''*) → Iterator[*nitpick.violations.Fuss*]
Warn about missing and different keys.

**write_file**(*file_exists: bool*) → Optional[*nitpick.violations.Fuss*]
Hook to write the new file when fix mode is on. Should be used by inherited classes.

**class** nitpick.plugins.text.**TextSchema**(*, *only: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Optional[Dict] = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: Optional[str] = None*)
Bases: marshmallow.schema.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 `marshmallow.schema.SchemaOpts`

**TYPE_MAPPING: typing.Dict[type, typing.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: Optional[bool] = 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: Optional[bool] = 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: typing.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: typing.Dict[str, ma_fields.Field]**
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.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: marshmallow.exceptions.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:* marshmallow.fields.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, **, many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = 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: *nitpick.violations.ViolationEnum*

Violations for this plugin.

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

nitpick.plugins.text.**can_handle**(*info:* nitpick.plugins.info.FileInfo) → Optional[Type[*nitpick.plugins.base.NitpickPlugin*]]
Handle text files.

nitpick.plugins.text.**plugin_class**() → Type[*nitpick.plugins.base.NitpickPlugin*]
Handle text files.

### nitpick.plugins.toml module

TOML files.

**class** nitpick.plugins.toml.**TomlPlugin**(*info:* nitpick.plugins.info.FileInfo, *expected_config: Dict[str, Any]*, *fix=False*)
Bases: *nitpick.plugins.base.NitpickPlugin*

Enforce config on 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*.

**can_fix: bool = True**
Can this plugin modify files directly?

**dirty: bool**

**enforce_rules**() → Iterator[*nitpick.violations.Fuss*]
Enforce rules for missing key/value pairs in the TOML file.

**entry_point**() → Iterator[*nitpick.violations.Fuss*]
Entry point of the Nitpick plugin.

**expected_config: JsonDict**

**file_path: Path**

**filename = ''**

**classmethod get_compiled_jmespath_filenames**()
Return a compiled JMESPath expression for file names, using the class name as part of the key.

**identify_tags: Set[str] = {'toml'}**
Which `identify` tags this *nitpick.plugins.base.NitpickPlugin* child recognises.

**init**()
Hook for plugin initialization after the instance was created.

**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.

**report**(*violation:* nitpick.violations.ViolationEnum, *document: Optional[tomlkit.toml_document.TOMLDocument]*, *change_dict: Optional[*nitpick.formats.BaseFormat*]*)
Report a violation while optionally modifying the TOML document.

**skip_empty_suggestion = False**

**validation_schema: Optional[marshmallow.schema.Schema] = 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**

**warn_missing_different**(*comparison:* nitpick.formats.Comparison, *prefix: str = ''*) → Iterator[*nitpick.violations.Fuss*]
Warn about missing and different keys.

**write_file**(*file_exists: bool*) → Optional[*nitpick.violations.Fuss*]
Hook to write the new file when fix mode is on. Should be used by inherited classes.

nitpick.plugins.toml.**can_handle**(*info:* nitpick.plugins.info.FileInfo) → Optional[Type[*nitpick.plugins.base.NitpickPlugin*]]
Handle TOML files.

nitpick.plugins.toml.**change_toml**(*document: tomlkit.toml_document.TOMLDocument*, *dictionary*)
Traverse a TOML document recursively and change values, keeping its formatting and comments.

nitpick.plugins.toml.**plugin_class**() → Type[*nitpick.plugins.base.NitpickPlugin*]
Handle TOML files.

### nitpick.style package

Styles parsing and merging.

**class** nitpick.style.**Style**(*project:* nitpick.project.Project, *offline: bool*, *cache_option: str*, *_first_full_path: str* = '')

Bases: object

Include styles recursively from one another.

**property cache_dir:** pathlib.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[*nitpick.plugins.base.NitpickPlugin*]*) → Dict[str, *marshmallow.fields.Field*]
Return a schema field with info from a config file class.

**find_initial_styles**(*configured_styles: Union[str, Iterable[str]]*) → Iterator[*nitpick.violations.Fuss*]
Find the initial style(s) and include them.

**static get_default_style_url**()
Return the URL of the default style for the current version.

**get_style_path**(*style_uri: str*) → Tuple[Optional[pathlib.Path], str]
Get the style path from the URI. Add the .toml extension if it's missing.

**include_multiple_styles**(*chosen_styles: Union[str, Iterable[str]]*) → Iterator[*nitpick.violations.Fuss*]
Include a list of styles (or just one) into this style tree.

**load_fixed_name_plugins**() → Set[Type[*nitpick.plugins.base.NitpickPlugin*]]
Separate classes with fixed file names from classes with dynamic files names.

**merge_toml_dict**() → Dict[str, Any]
Merge all included styles into a TOML (actually JSON) dictionary.

**offline:** bool

**project:** *nitpick.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[*nitpick.enums.CachingEnum*, datetime.timedelta]
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.**StyleFetcherManager**(*offline: bool*, *cache_dir: str*, *cache_option: str*)

Bases: object

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

**cache_dir:** str

**cache_option:** str

**cache_repository:** cachy.repository.Repository

**fetch**(*url*) → Tuple[Optional[pathlib.Path], str]
Determine which fetcher to be used and fetch from it.

Try a fetcher by domain first, then by protocol scheme.

**fetchers:** FetchersType

**offline:** bool

### Submodules

### nitpick.style.fetchers.base module

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

**class** nitpick.style.fetchers.base.**StyleFetcher**(*cache_manager: cachy.cache_manager.CacheManager*, *cache_option: str*, *protocols: Tuple[str, ...] = ()*, *domains: Tuple[str, ...] = ()*)

Bases: object

Base class of all fetchers, it encapsulate get/fetch from cache.

**cache_manager:** cachy.cache_manager.CacheManager

**cache_option:** str

**domains:** Tuple[str, ...] = ()

**fetch**(*url*) → Tuple[Optional[pathlib.Path], str]
Fetch a style form cache or from a specific fetcher.

**protocols:** Tuple[str, ...] = ()

**requires_connection** = False

### nitpick.style.fetchers.file module

Basic local file fetcher.

**class** nitpick.style.fetchers.file.**FileFetcher**(*cache_manager: cachy.cache_manager.CacheManager*, *cache_option: str*, *protocols: Tuple[str, ...] = ('file', '')*, *domains: Tuple[str, ...] = ()*)

Bases: *nitpick.style.fetchers.base.StyleFetcher*

Fetch a style from a local file.

**cache_manager:** CacheManager

**cache_option:** str

**domains:** Tuple[str, ...] = ()

**fetch**(*url*) → Tuple[Optional[pathlib.Path], str]
Fetch a style form cache or from a specific fetcher.

**protocols:** Tuple[str, ...] = ('file', '')

**requires_connection = False**

### nitpick.style.fetchers.github module

Support for gh and github schemes.

**class** nitpick.style.fetchers.github.**GitHubFetcher**(*cache_manager: cachy.cache_manager.CacheManager*, *cache_option: str*, *protocols: Tuple[str, ...] = ('gh', 'github')*, *domains: Tuple[str, ...] = ('github.com',)*)

Bases: *nitpick.style.fetchers.http.HttpFetcher*

Fetch styles from GitHub repositories.

**cache_manager: CacheManager**

**cache_option: str**

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

**fetch**(*url*) → Tuple[Optional[pathlib.Path], str]

Fetch a style form cache or from a specific fetcher.

**protocols: Tuple[str, ...] = ('gh', 'github')**

**requires_connection = True**

**class** nitpick.style.fetchers.github.**GitHubProtocol**(*value*)

Bases: enum.Enum

Protocols for the GitHUb scheme.

**LONG = 'github'**

**SHORT = 'gh'**

**class** nitpick.style.fetchers.github.**GitHubURL**(*owner: str*, *repository: str*, *git_reference: str*, *path: str*)

Bases: object

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

**property api_url: str**

API URL for this repo.

**property default_branch: str**

Default GitHub branch.

This property performs a HTTP request and it's memoized with lru_cache().

**git_reference: str**

**property git_reference_or_default: str**

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

**property long_protocol_url: str**

Long protocol URL (github).

**owner: str**

**classmethod parse_url**(*url: str*) → *nitpick.style.fetchers.github.GitHubURL*

Create an instance by parsing a URL string in any accepted format.

See the code for test_parsing_github_urls() for more examples.

**path: str**

property raw_content_url: str
Raw content URL for this path.

repository: str

property short_protocol_url: str
Short protocol URL (gh).

property url: str
Default URL built from attributes.

nitpick.style.fetchers.github.**get_default_branch**(*api_url: str*) → str
Get the default branch from the GitHub repo using the API.

For now, 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.

Another option for the future: perform an authenticated request to GitHub. That would require a `requests.Session` and some user credentials.

### nitpick.style.fetchers.http module

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

class nitpick.style.fetchers.http.**HttpFetcher**(*cache_manager: cachy.cache_manager.CacheManager*, *cache_option: str*, *protocols: Tuple[str, ...] = ('http', 'https')*, *domains: Tuple[str, ...] = ()*)
Bases: *nitpick.style.fetchers.base.StyleFetcher*

Fetch a style from an http/https server.

cache_manager: CacheManager

cache_option: str

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

fetch(*url*) → Tuple[Optional[pathlib.Path], str]
Fetch a style form cache or from a specific fetcher.

protocols: Tuple[str, ...] = ('http', 'https')

requires_connection = True

### nitpick.style.fetchers.pypackage module

Support for py schemes.

class nitpick.style.fetchers.pypackage.**PythonPackageFetcher**(*cache_manager: cachy.cache_manager.CacheManager*, *cache_option: str*, *protocols: Tuple[str, ...] = ('py', 'pypackage')*, *domains: Tuple[str, ...] = ()*)
Bases: *nitpick.style.fetchers.base.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`.

**cache_manager:** `CacheManager`

**cache_option:** `str`

**domains:** `Tuple[str, ...] = ()`

**fetch**(*url*) → Tuple[Optional[pathlib.Path], str]
Fetch a style form cache or from a specific fetcher.

**protocols:** `Tuple[str, ...] = ('py', 'pypackage')`

**requires_connection** = False

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

Represent a resource file in installed Python package.

**import_path:** `str`

**classmethod parse_url**(*url: str*) → *nitpick.style.fetchers.pypackage.PythonPackageURL*
Create an instance by parsing a URL string in any accepted format.

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

**property raw_content_url:** `importlib_resources.abc.Traversable`
Raw path of resource file.

**resource_name:** `str`

### Submodules

### nitpick.style.cache module

Cache functions and configuration for styles.

nitpick.style.cache.**parse_cache_option**(*cache_option: str*) → Tuple[*nitpick.enums.CachingEnum*, datetime.timedelta]
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:* nitpick.project.Project)
Bases: `object`

Validate a nitpick configuration.

**project:** *nitpick.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.**Style**(*project:* nitpick.project.Project, *offline:* *bool*, *cache_option:* *str*, *_first_full_path:* *str* = '')

Bases: object

Include styles recursively from one another.

**property cache_dir:** pathlib.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[*nitpick.plugins.base.NitpickPlugin*]*) → Dict[str, *marshmallow.fields.Field*]
Return a schema field with info from a config file class.

**find_initial_styles**(*configured_styles: Union[str, Iterable[str]]*) → Iterator[*nitpick.violations.Fuss*]
Find the initial style(s) and include them.

**static get_default_style_url**()
Return the URL of the default style for the current version.

**get_style_path**(*style_uri:* *str*) → Tuple[Optional[pathlib.Path], str]
Get the style path from the URI. Add the .toml extension if it's missing.

**include_multiple_styles**(*chosen_styles: Union[str, Iterable[str]]*) → Iterator[*nitpick.violations.Fuss*]
Include a list of styles (or just one) into this style tree.

**load_fixed_name_plugins**() → Set[Type[*nitpick.plugins.base.NitpickPlugin*]]
Separate classes with fixed file names from classes with dynamic files names.

**merge_toml_dict**() → Dict[str, Any]
Merge all included styles into a TOML (actually JSON) dictionary.

**offline:** bool

**project:** *nitpick.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.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: click.core.Context*) → *nitpick.core.Nitpick*
Create a Nitpick instance from the click context parameters.

### nitpick.constants module

Constants.

nitpick.constants.**SEPARATOR_QUOTED_SPLIT = '#$@'**
Special unique separator for *nitpick.generic.quoted_split()*.

nitpick.constants.**SLASH = '/'**
Dot/slash is used to indicate a local style file

### nitpick.core module

The Nitpick application.

**class** nitpick.core.**Nitpick**
Bases: object

The Nitpick API.

**configured_files**(**partial_names: str*) → List[pathlib.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[*nitpick.violations.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*, *fix=True*) → Iterator[*nitpick.violations.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.
- **fix** – Flag to modify files, if the plugin supports it (default: True).

**Returns** Fuss generator.

**init**(*project_root: Optional[Union[pathlib.Path, str]] = None*, *offline: Optional[bool] = None*) → *nitpick.core.Nitpick*
Initialize attributes of the singleton.

**offline:** bool

**project:** *nitpick.project.Project*

**run**(**partial_names: str*, *fix=False*) → Iterator[*nitpick.violations.Fuss*]
Run Nitpick.

**Parameters**

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

**Returns** Fuss generator.

**classmethod singleton**() → *nitpick.core.Nitpick*
Return a single instance of the class.

### nitpick.enums module

Enums.

**class** nitpick.enums.**CachingEnum**(*value*)
Bases: enum.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: nitpick.enums._OptionMixin, enum.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_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: Union[*nitpick.violations.Fuss, *List[*nitpick.violations.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: Optional[Union[*marshmallow.fields.Field*, type]] = None*, *values: Optional[Union[*marshmallow.fields.Field*, type]] = None*, ***kwargs*)
Bases: marshmallow.fields.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: Optional[str] = None*, *data: Optional[Mapping[str, Any]] = 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*) → marshmallow.exceptions.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: Optional[Callable[[Any, str, Any], Any]] = 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: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, data_key: Optional[str] = None, attribute: Optional[str] = None, validate: Optional[Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]]]] = None, required: bool = False, allow_none: Optional[bool] = None, load_only: bool = False, dump_only: bool = False, error_messages: Optional[Dict[str, str]] = None, metadata: Optional[Mapping[str, Any]] = None, **additional_metadata*)
Bases: `marshmallow.base.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 `missing=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: Optional[str] = None*, *data: Optional[Mapping[str, Any]] = 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*) → marshmallow.exceptions.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: Optional[Callable[[Any, str, Any], Any]] = 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: Union[*marshmallow.fields.Field*, type]*, ***kwargs*)
Bases: *marshmallow.fields.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: Optional[str] = None*, *data: Optional[Mapping[str, Any]] = 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*) → marshmallow.exceptions.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: Optional[Callable[[Any, str, Any], Any]] = 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: Union[marshmallow.base.SchemaABC, type, str, Callable[[], marshmallow.base.SchemaABC]], *, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, only: Optional[Union[Sequence[str], Set[str]]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, unknown: Optional[str] = None, **kwargs*)
Bases: *marshmallow.fields.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), or callable that returns a *Schema* instance.
- **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: Optional[str] = None*, *data: Optional[Mapping[str, Any]] = 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*) → marshmallow.exceptions.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: Optional[Callable[[Any, str, Any], Any]] = 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: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, data_key: Optional[str] = None, attribute: Optional[str] = None, validate: Optional[Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]]]] = None, required: bool = False, allow_none: Optional[bool] = None, load_only: bool = False, dump_only: bool = False, error_messages: Optional[Dict[str, str]] = None, metadata: Optional[Mapping[str, Any]] = None, **additional_metadata*)
Bases: *marshmallow.fields.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: Optional[str] = None*, *data: Optional[Mapping[str, Any]] = 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*) → marshmallow.exceptions.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: Optional[Callable[[Any, str, Any], Any]] = 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.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: flake8.options.manager.OptionManager*)
Add the offline option.

**build_flake8_error**(*obj:* nitpick.violations.Fuss) → Tuple[int, int, str, Type]
Return a flake8 error from a fuss.

**collect_errors**() → Iterator[*nitpick.violations.Fuss*]
Collect all possible Nitpick errors.

**name = 'nitpick'**

**static parse_options**(*option_manager: flake8.options.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 = '0.28.0'**

### nitpick.formats module

Configuration file formats.

class nitpick.formats.**BaseFormat**(*, *path: Optional[Union[pathlib.Path, str]] = None*, *string: Optional[str] = None*, *data: Optional[Union[Dict[str, Any], ruamel.yaml.comments.CommentedSeq, ruamel.yaml.comments.CommentedMap,* nitpick.formats.BaseFormat*]] = None*, *ignore_keys: Optional[List[str]] = None*)

Bases: object

Base class for configuration file formats.

**Parameters**

- **path** – Path of the config file to be loaded.
- **string** – Config in string format.
- **data** – Config data in Python format (dict, YAMLFormat, TOMLFormat instances).
- **ignore_keys** – List of keys to ignore when using the comparison methods.

**property as_data: Union[Dict[str, Any], ruamel.yaml.comments.CommentedSeq, ruamel.yaml.comments.CommentedMap]**
String content converted to a Python data structure (a dict, YAML data, etc.).

**property as_string: str**
Contents of the file or the original string provided when the instance was created.

**classmethod cleanup**(**args: List[Any]*) → List[Any]
Cleanup similar values according to the specific format. E.g.: YAMLFormat accepts 'True' or 'true'.

**compare_with_dictdiffer**(*expected: Optional[Union[Dict[str, Any], nitpick.formats.BaseFormat]] = None*, *transform_function: Optional[Callable] = None*) → *nitpick.formats.Comparison*

Compare two structures and compute missing and different items using `dictdiffer`.

**compare_with_flatten**(*expected: Optional[Union[Dict[str, Any], nitpick.formats.BaseFormat]] = None*) → *nitpick.formats.Comparison*

Compare two flattened dictionaries and compute missing and different items.

**abstract load**()

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.formats.**Comparison**(*actual: Union[Dict[str, Any], ruamel.yaml.comments.CommentedSeq, ruamel.yaml.comments.CommentedMap, nitpick.formats.BaseFormat]*, *expected: Union[Dict[str, Any], ruamel.yaml.comments.CommentedSeq, ruamel.yaml.comments.CommentedMap, nitpick.formats.BaseFormat]*, *format_class: Type[nitpick.formats.BaseFormat]*)

Bases: `object`

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

**property has_changes: bool**

Return True is there is a difference or something missing.

**set_diff**(*diff_dict*)

Set the diff dict and corresponding format.

**set_missing**(*missing_dict*)

Set the missing dict and corresponding format.

**update_pair**(*key*, *raw_expected*)

Update a key on one of the comparison dicts, with its raw expected value.

**class** nitpick.formats.**JSONFormat**(*, *path: Optional[Union[pathlib.Path, str]] = None*, *string: Optional[str] = None*, *data: Optional[Union[Dict[str, Any], ruamel.yaml.comments.CommentedSeq, ruamel.yaml.comments.CommentedMap, nitpick.formats.BaseFormat]] = None*, *ignore_keys: Optional[List[str]] = None*)

Bases: *nitpick.formats.BaseFormat*

JSON configuration format.

**property as_data: Union[Dict[str, Any], ruamel.yaml.comments.CommentedSeq, ruamel.yaml.comments.CommentedMap]**

String content converted to a Python data structure (a dict, YAML data, etc.).

**property as_string: str**

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

**classmethod cleanup**(**args: List[Any]*) → List[Any]

Cleanup similar values according to the specific format. E.g.: YAMLFormat accepts 'True' or 'true'.

**compare_with_dictdiffer**(*expected: Optional[Union[Dict[str, Any], nitpick.formats.BaseFormat]] = None*, *transform_function: Optional[Callable] = None*) → *nitpick.formats.Comparison*

Compare two structures and compute missing and different items using `dictdiffer`.

**compare_with_flatten**(*expected: Optional[Union[Dict[str, Any], nitpick.formats.BaseFormat]] = None*) → *nitpick.formats.Comparison*

Compare two flattened dictionaries and compute missing and different items.

**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.formats.**TOMLFormat**(**, path: Optional[Union[pathlib.Path, str]] = None, string: Optional[str] = None, data: Optional[Union[Dict[str, Any], ruamel.yaml.comments.CommentedSeq, ruamel.yaml.comments.CommentedMap, nitpick.formats.BaseFormat]] = None, ignore_keys: Optional[List[str]] = None*)

Bases: *nitpick.formats.BaseFormat*

TOML configuration format.

**property as_data: Union[Dict[str, Any], ruamel.yaml.comments.CommentedSeq, ruamel.yaml.comments.CommentedMap]**

String content converted to a Python data structure (a dict, YAML data, etc.).

**property as_string: str**

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

**classmethod cleanup**(**args: List[Any]*) → List[Any]

Cleanup similar values according to the specific format. E.g.: YAMLFormat accepts 'True' or 'true'.

**compare_with_dictdiffer**(*expected: Optional[Union[Dict[str, Any], nitpick.formats.BaseFormat]] = None, transform_function: Optional[Callable] = None*) → *nitpick.formats.Comparison*

Compare two structures and compute missing and different items using `dictdiffer`.

**compare_with_flatten**(*expected: Optional[Union[Dict[str, Any], nitpick.formats.BaseFormat]] = None*) → *nitpick.formats.Comparison*

Compare two flattened dictionaries and compute missing and different items.

**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.formats.**YAMLFormat**(**, path: Optional[Union[pathlib.Path, str]] = None, string: Optional[str] = None, data: Optional[Union[Dict[str, Any], ruamel.yaml.comments.CommentedSeq, ruamel.yaml.comments.CommentedMap, nitpick.formats.BaseFormat]] = None, ignore_keys: Optional[List[str]] = None*)

Bases: *nitpick.formats.BaseFormat*

YAML configuration format.

**property as_data: ruamel.yaml.comments.CommentedMap**

On YAML, this dict is a special object with comments and ordered keys.

**property as_list: ruamel.yaml.comments.CommentedSeq**

A list of dicts. On YAML, `as_data` might contain a `list`. This property is just a proxy for typing.

**property as_string: str**

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

**classmethod cleanup**(**args: List[Any]*) → List[Any]
A boolean "True" or "true" might have the same effect on YAML.

**compare_with_dictdiffer**(*expected: Optional[Union[Dict[str, Any], nitpick.formats.BaseFormat]] = None*, *transform_function: Optional[Callable] = None*) → *nitpick.formats.Comparison*
Compare two structures and compute missing and different items using `dictdiffer`.

**compare_with_flatten**(*expected: Optional[Union[Dict[str, Any], nitpick.formats.BaseFormat]] = None*) → *nitpick.formats.Comparison*
Compare two flattened dictionaries and compute missing and different items.

**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).

### nitpick.generic module

Generic functions and classes.

**class** nitpick.generic.**MergeDict**(*original_dict: Optional[Dict[str, Any]] = None*)
Bases: `object`

A dictionary that can merge other dictionaries into it.

**add**(*other: Dict[str, Any]*) → None
Add another dictionary to the existing data.

**merge**(*sort=True*) → Dict[str, Any]
Merge the dictionaries, replacing values with identical keys and extending lists.

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.**find_object_by_key**(*list_: List[dict]*, *search_key: str*, *search_value: Any*) → dict
Find an object in a list, using a key/value pair to search.

```
>>> fruits = [{"id": 1, "fruit": "banana"}, {"id": 2, "fruit": "apple"}, {"id": 3,
↪"fruit": "mango"}]
>>> find_object_by_key(fruits, "id", 1) == {'id': 1, 'fruit': 'banana'}
True
>>> find_object_by_key(fruits, "fruit", "banana") == {'id': 1, 'fruit': 'banana'}
```

(continues on next page)

(continued from previous page)

```
True
>>> find_object_by_key(fruits, "fruit", "pear")
{}
>>> find_object_by_key(fruits, "fruit", "mango") == {'id': 3, 'fruit': 'mango'}
True
>>> find_object_by_key(None, "fruit", "pear")
{}
```

nitpick.generic.**flatten**(*dict_*, *parent_key=''*, *separator='.'*, *current_lists=None*) → Dict[str, Any]

Flatten a nested dict.

Use *unflatten()* to revert.

Adapted from this StackOverflow question.

nitpick.generic.**get_subclasses**(*cls*)

Recursively get subclasses of a parent class.

nitpick.generic.**is_url**(*url: str*) → bool

Return True if a string is a URL.

```
>>> is_url("")
False
>>> is_url("  ")
False
>>> is_url("http://example.com")
True
>>> is_url("github://andreoliwa/nitpick/styles/black")
True
```

nitpick.generic.**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.generic.**relative_to_current_dir**(*path_or_str: Optional[Union[pathlib.Path, str]]*) → str

Return a relative path to the current dir or an absolute path.

nitpick.generic.**search_dict**(*jmespath_expression: Union[jmespath.parser.ParsedResult, str]*, *data: MutableMapping[str, Any]*, *default: Any*) → Any

Search a dictionary using a JMESPath expression, and returning a default value.

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

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

**Parameters**

- **jmespath_expression** – A compiled JMESPath expression or a string with an expression.
- **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.generic.**unflatten**(*dict_*, *separator='.'*, *sort=True*) → collections.OrderedDict

Turn back a flattened dict created by *flatten()* into a nested dict.

```
>>> expected = {'my': {'sub': {'path': True}, 'home': 4}, 'another': {'path': 3}}
>>> unflatten({"my.sub.path": True, "another.path": 3, "my.home": 4}) == expected
True
>>> unflatten({"repo": "conflicted key", "repo.name": "?", "repo.path": "?"})
Traceback (most recent call last):
  ...
TypeError: 'str' object does not support item assignment
```

nitpick.generic.**version_to_tuple**(*version: Optional[str] = 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: Optional[pathlib.Path]*, *styles: Union[str, List[str]]*, *cache: str*)
Bases: object

Configuration read from one of the CONFIG_FILES.

**cache:** str

**file:** Optional[pathlib.Path]

**styles:** Union[str, List[str]]

**class** nitpick.project.**Project**(*root: Optional[Union[pathlib.Path, str]] = None*)
Bases: object

A project to be nitpicked.

**create_configuration**() → None
Create a configuration file.

**merge_styles**(*offline: bool*) → Iterator[*nitpick.violations.Fuss*]
Merge one or multiple style files.

**property plugin_manager:** pluggy._manager.PluginManager
Load all defined plugins.

**read_configuration**() → *nitpick.project.Configuration*
Search for a configuration file and validate it against a Marshmallow schema.

**property root:** pathlib.Path
Root dir of the project.

**class** nitpick.project.**ToolNitpickSectionSchema**(**, only: Optional[Union[Sequence[str], Set[str]]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Optional[Dict] = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: Optional[str] = None*)
Bases: *nitpick.schemas.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 marshmallow.schema.SchemaOpts

**TYPE_MAPPING: typing.Dict[type, typing.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: Optional[bool] = 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: Optional[bool] = 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: typing.Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/configuration.html.'}**
Overrides for default schema-level error messages

**fields: typing.Dict[str, ma_fields.Field]**
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.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: marshmallow.exceptions.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:* marshmallow.fields.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = 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.**climb_directory_tree**(*starting_path: Union[pathlib.Path, str]*, *file_patterns: Iterable[str]*) → Set[pathlib.Path]

Climb the directory tree looking for file patterns.

nitpick.project.**find_main_python_file**(*root_dir: pathlib.Path*) → pathlib.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.**find_root**(*current_dir: Optional[Union[pathlib.Path, str]] = None*) → pathlib.Path

Find the root dir of the Python project (the one that has one of the `ROOT_FILES`).

Start from the current working dir.

nitpick.project.**find_starting_dir**(*current_dir: Union[pathlib.Path, str]*) → pathlib.Path

Find the starting dir from the current dir.

### nitpick.schemas module

Marshmallow schemas.

class nitpick.schemas.**BaseNitpickSchema**(*, *only: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Optional[Dict] = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: Optional[str] = None*)

Bases: `marshmallow.schema.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 marshmallow.schema.SchemaOpts

**TYPE_MAPPING: typing.Dict[type, typing.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: Optional[bool] = 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: Optional[bool] = 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: typing.Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/nitpick_section.html.'}**
Overrides for default schema-level error messages

**fields: typing.Dict[str, ma_fields.Field]**
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.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: marshmallow.exceptions.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:* marshmallow.fields.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = 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: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Optional[Dict] = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: Optional[str] = None*)

Bases: marshmallow.schema.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 marshmallow.schema.SchemaOpts

TYPE_MAPPING: typing.Dict[type, typing.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: Optional[bool] = 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: Optional[bool] = 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**: typing.Dict[str, str] = {'unknown': 'Unknown file. See https://nitpick.rtfd.io/en/latest/plugins.html.'}
Overrides for default schema-level error messages

**fields**: typing.Dict[str, ma_fields.Field]
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.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: marshmallow.exceptions.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, *, *many: Optional[bool]* = *None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]]* = *None*, *unknown: Optional[str]* = *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: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:* marshmallow.fields.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = 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: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Optional[Dict] = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: Optional[str] = None*)

Bases: *nitpick.schemas.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 marshmallow.schema.SchemaOpts

**TYPE_MAPPING: typing.Dict[type, typing.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: Optional[bool] = 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: typing.Dict[str, ma_fields.Field]**

**dumps**(*obj: Any*, **args*, *many: Optional[bool] = 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: typing.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: typing.Dict[str, ma_fields.Field]**
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.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: marshmallow.exceptions.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:** `typing.Dict[str, ma_fields.Field]`

**loads**(*json_data: str*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:* marshmallow.fields.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = 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: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Optional[Dict] = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: Optional[str] = None*)
Bases: *nitpick.schemas.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 `marshmallow.schema.SchemaOpts`

**TYPE_MAPPING: typing.Dict[type, typing.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: Optional[bool] = 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: typing.Dict[str, ma_fields.Field]**

**dumps**(*obj: Any*, **args*, *many: Optional[bool] = 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: typing.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: typing.Dict[str, ma_fields.Field]**
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.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: marshmallow.exceptions.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, *, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:** `typing.Dict[str, ma_fields.Field]`

**loads**(*json_data: str*, *, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:* marshmallow.fields.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, **, many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = 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: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Optional[Dict] = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: Optional[str] = None*)

Bases: *nitpick.schemas.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 marshmallow.schema.SchemaOpts

TYPE_MAPPING: typing.Dict[type, typing.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: Optional[bool] = 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:** `typing.Dict[str, ma_fields.Field]`

**dumps**(*obj: Any*, **args*, *many: Optional[bool] = 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:** `typing.Dict[str, str] = {'unknown': 'Unknown configuration. See https://nitpick.rtfd.io/en/latest/nitpick_section.html.'}`
Overrides for default schema-level error messages

**fields:** `typing.Dict[str, ma_fields.Field]`
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.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: marshmallow.exceptions.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:** `typing.Dict[str, ma_fields.Field]`

**loads**(*json_data: str*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:* marshmallow.fields.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = 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: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Optional[Dict] = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: Optional[str] = None*)

Bases: *nitpick.schemas.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 marshmallow.schema.SchemaOpts

TYPE_MAPPING: typing.Dict[type, typing.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: Optional[bool] = 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: typing.Dict[str, ma_fields.Field]

dumps(*obj: Any*, **args*, *many: Optional[bool] = 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: typing.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: typing.Dict[str, ma_fields.Field]**
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.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: marshmallow.exceptions.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, *, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:** `typing.Dict[str, ma_fields.Field]`

**loads**(*json_data: str*, *, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = 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:* marshmallow.fields.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: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, *, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = 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: *nitpick.violations.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*, *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:* nitpick.violations.ViolationEnum, *suggestion: str = ''*, *fixed=False*, ***kwargs*) → *nitpick.violations.Fuss*
Make a fuss.

**manual: int = 0**

**classmethod reset**()
Reset the counters.

class nitpick.violations.**SharedViolations**(*value*)
Bases: *nitpick.violations.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: *nitpick.violations.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.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

## A



## B



## C

## D

## E

## F

## G

## H



## I



## J

## K



## L



## M

## N

## O

## P



## Q



## R

## S



## T

## U



## V



## W

## Y