configconfig

Release 0.6.2

Load and validate YAML configuration files.

Dominic Davis-Foster

Apr 20, 2023

Contents

Chapter

ONE

Installation

1.1 from PyPI

```
$ python3 -m pip install configconfig --user
```

1.2 from Anaconda

First add the required channels

```
$ conda config --add channels https://conda.anaconda.org/conda-forge
$ conda config --add channels https://conda.anaconda.org/domdfcoding
```

Then install

```
$ conda install configconfig
```

1.3 from GitHub

```
$ python3 -m pip install git+https://github.com/repo-helper/configconfig@master --user
```

Chapter

TWO

configconfig.autoconfig

A Sphinx directive for documenting YAML configuration values.

Provides the *autoconfig* directive to document configuration values automatically, the *conf* directive to document them manually, and the *conf* role to link to a *conf* directive.

Attention: This module has the following additional requirements:

```
docutils
sphinx<3.4.0,>=3.0.3
sphinx-toolbox
```

These can be installed as follows:

```
$ python -m pip install configconfig[sphinx]
```

2.1 Usage

.. autoconfig::
Directive to automatically document an YAML configuration value.

Takes a single argument, either the fully qualified name of the *ConfigVar* object, or the name of the module if the `:category:` option is given.

:category: (string)
(optional) The category of options to document.

.. conf::
Directive to document an YAML configuration value.

:conf:
Role to add a cross-reference to a *conf* or *autoconfig* directive.

2.2 API Reference

Classes:

AutoConfigDirective(name, arguments, ...)	Sphinx directive to automatically document an YAML configuration value.

Functions:

parse_conf_node(env, text, node)	Parse the content of a *conf* directive.
setup(app)	Setup Sphinx Extension.

class AutoConfigDirective(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: SphinxDirective

Sphinx directive to automatically document an YAML configuration value.

Methods:

document_config_var(var_obj)	Document the given configuration value.
run()	Process the content of the directive.

document_config_var(*var_obj*)

Document the given configuration value.

Parameters **var_obj** (Type[*ConfigVar*])

Return type paragraph

run()

Process the content of the directive.

Return type Sequence[Node]

parse_conf_node(*env*, *text*, *node*)

Parse the content of a *conf* directive.

Parameters

- **env** (BuildEnvironment) – The Sphinx build environment.
- **text** (str) – The content of the directive.
- **node** (desc_signature) – The docutils node class.

Return type str

setup(*app*)

Setup Sphinx Extension.

Parameters **app** (Sphinx) – The Sphinx app

Return type Dict[str, Any]

Chapter

THREE

configconfig.configvar

class ConfigVar(*raw_config_vars: Dict[str, Any]*)

Base class for `YAML` configuration values.

The class docstring should be the description of the config var, with an example, and the name of the class should be the variable name.

Alternatively, for a more Pythonic naming approach, the variable name can be set with the `name` class variable.

Example:

```
class platforms(ConfigVar):
    """
    A case-insensitive list of platforms to perform tests for.

    Example:

    .. code-block:: yaml

        platforms:
          - Windows
          - Linux

    These values determine the GitHub test workflows to enable,
    and the Trove classifiers used on PyPI.
    """

    dtype = List[Literal["Windows", "macOS", "Linux"]]
    default: List[str] = ["Windows", "macOS", "Linux"]
    category: str = "packaging"
```

Attributes:

category	The category the *ConfigVar* is listed under in the documentation.
default	The default value of the configuration value if it is optional.
dtype	The allowed type or types in the `YAML` configuration file.
required	Flag to indicate whether the configuration value is required.
rtype	The variable type passed to Jinja2.

Methods:

get([raw_config_vars])	Returns the value of this *ConfigVar*.
make_documentation()	Returns the reStructuredText documentation for the *ConfigVar*.
validate([raw_config_vars])	Validate the value obtained from the `YAML` file and coerce into the appropriate return type.
validator(value)	Function to call to validate the values.

category = 'other'
Type: str

The category the *ConfigVar* is listed under in the documentation.

default = ''
Type: Union[Callable[[Dict[str, Any]], Any], Any]

The default value of the configuration value if it is optional. Defaults to '' if unset.

May also be set to a callable which returns a dynamic or mutable default value.

dtype = typing.Any
Type: Type

The allowed type or types in the YAML configuration file.

classmethod get(*raw_config_vars=None*)
Returns the value of this *ConfigVar*.

Parameters **raw_config_vars** (Optional[Dict[str, Any]]) – Dictionary to obtain the value from. Default None.

Return type See the *rtype* attribute.

classmethod make_documentation()
Returns the reStructuredText documentation for the *ConfigVar*.

Return type str

required = False
Type: bool

Flag to indicate whether the configuration value is required. Default False.

rtype = typing.Any
Type: Type

The variable type passed to Jinja2. If None *dtype* is used. Ignored for dtype=bool.

classmethod validate(*raw_config_vars=None*)
Validate the value obtained from the YAML file and coerce into the appropriate return type.

Parameters **raw_config_vars** (Optional[Dict[str, Any]]) – Dictionary to obtain the value from. Default None.

Return type See the *rtype* attribute.

classmethod validator(*value*)
Function to call to validate the values.

- The callable must have a single required argument (the value).
- Should raise ValueError if values are invalid, and return the values if they are valid.
- May change the values (e.g. make lowercase) before returning.

Return type Any

Chapter

FOUR

configconfig.metaclass

class ConfigVarMeta(*name: str*, *bases*, *dct: Dict*)

Bases: type

Metaclass for configuration values.

Methods:

__call__(raw_config_vars)	Alias for *ConfigVar.get*.
__repr__()	Return a string representation of the *ConfigVarMeta* class.
get_schema_entry([schema])	Returns the JSON schema entry for this configuration value.

__call__(*raw_config_vars*)

Alias for *ConfigVar.get*.

Returns the value of the *ConfigVar*.

Parameters **raw_config_vars** (Dict[str, Any]) – Dictionary to obtain the value from.

Return type See the *ConfigVar.rtype* attribute.

__repr__()

Return a string representation of the *ConfigVarMeta* class.

Return type str

get_schema_entry(*schema=None*)

Returns the JSON schema entry for this configuration value.

Parameters **schema** (Optional[Dict]) – Default None.

Return type Dict[str, Any]

Returns Dictionary representation of the JSON schema.

Chapter

FIVE

configconfig.parser

class Parser(*allow_unknown_keys=False*)

Base class for YAML configuration parsers.

Custom parsing steps for each configuration variable can be implemented with methods in the form:

```
def visit_<configuration value name>(
        self,
        raw_config_vars: Dict[str, Any],
        ) -> Any: ...
```

The method must return the value to set the configuration variable to, or raise an error in the case of invalid input.

A final custom parsing step, useful when several values must be set at once, may be implemented in the `custom_parsing` method:

```
def custom_parsing(
        self,
        raw_config_vars: Mapping[str, Any],
        parsed_config_vars: MutableMapping[str, Any],
        filename: PathPlus,
        ) -> MutableMapping[str, Any]: ...
```

This takes the mapping of raw configuration variables, the mapping of parsed variables (those set with the `visit_<configuration value name>` method), and the configuration file name. The method must return the `parsed_config_vars`.

Methods:

custom_parsing(raw_config_vars, ...)	Custom parsing step.
run(filename)	Parse configuration from the given file.

custom_parsing(*raw_config_vars*, *parsed_config_vars*, *filename*)

Custom parsing step.

Parameters

- **raw_config_vars** (Mapping[str, Any]) – Mapping of raw configuration values loaded from the YAML configuration file.
- **parsed_config_vars** (MutableMapping[str, Any]) – Mapping of parsed configuration values.
- **filename** (PathPlus) – The filename of the YAML configuration file.

Return type MutableMapping[str, Any]

run (*filename*)
Parse configuration from the given file.

Parameters **filename** (Union[str, Path, PathLike]) – The filename of the YAML configuration file.

Return type MutableMapping[str, Any]

Chapter

SIX

configconfig.testing

Helpers for testing *ConfigVar*.

Attention: This module has the following additional requirement:

```
pytest
```

This can be installed as follows:

```
$ python -m pip install configconfig[testing]
```

New in version 0.2.0.

Classes:

BoolFalseTest()	Test for boolean configuration values which default to False.
BoolTrueTest()	Test for boolean configuration values which default to True.
ConfigVarTest()	Base class for tests of *ConfigVar*s.
DictTest()	Test for dictionary configuration values.
DirectoryTest()	Test for configuration values which represent directories.
EnumTest()	Test for Enum configuration values.
ListTest()	Test for list configuration values.
NotBoolTest()	Mixin to add tests for *ConfigVar*s that can't be boolean values.
NotIntTest()	Mixin to add tests for *ConfigVar*s that can't be integers.
NotStrTest()	Mixin to add tests for *ConfigVar*s that can't be strings.
OptionalStringTest()	Test for string configuration values which are optional.
RequiredStringTest()	Test for string configuration values which are required.

class ConfigVarTest

Bases: ABC

Base class for tests of *ConfigVar*s.

config_var

Type: Type[*ConfigVar*]

The *ConfigVar* under test.

class NotIntTest
Bases: *ConfigVarTest*

Mixin to add tests for *ConfigVar*s that can't be integers.

test_error_int()
Checks that the *ConfigVar* raises a `ValueError` when passed an `int`.

class NotBoolTest
Bases: *ConfigVarTest*

Mixin to add tests for *ConfigVar*s that can't be boolean values.

test_error_bool()
Checks that the *ConfigVar* raises a `ValueError` when passed a `bool`.

class NotStrTest
Bases: *ConfigVarTest*

Mixin to add tests for *ConfigVar*s that can't be strings.

test_error_str()
Checks that the *ConfigVar* raises a `ValueError` when passed a `str`.

class ListTest
Bases: *NotStrTest*, *NotBoolTest*, *NotIntTest*, *ConfigVarTest*

Test for list configuration values.

Attributes:

default_value	The default value that should be returned when no valid is given.
different_key_value	A dictionary containing one or more keys that are not the keys used by the *ConfigVar*
test_value	A value that is valid and should be returned unchanged.

Methods:

test_success()	Checks that the *ConfigVar* can correctly parse various `list` values.

default_value = []
Type: `List[str]`

The default value that should be returned when no valid is given.

different_key_value = {'username': 'domdfcoding'}
Type: `Dict[str, Any]`

A dictionary containing one or more keys that are not the keys used by the *ConfigVar*

test_success()
Checks that the *ConfigVar* can correctly parse various `list` values.

test_value
Type: List[str]

A value that is valid and should be returned unchanged.

class DirectoryTest
Bases: *NotBoolTest*, *NotIntTest*, *ConfigVarTest*

Test for configuration values which represent directories.

Attributes:

default_value	The default value that should be returned when no valid is given.
different_key_value	A dictionary containing one or more keys that are not the keys used by the *ConfigVar*
test_value	A value that is valid and should be returned unchanged.

Methods:

test_error_list_int()	Checks that the *ConfigVar* raises a ValueError when passed a str.
test_error_list_str()	Checks that the *ConfigVar* raises a ValueError when passed a str.
test_success()	Checks that the *ConfigVar* can correctly parse various directory values.

default_value
Type: str

The default value that should be returned when no valid is given.

different_key_value = {'username': 'domdfcoding'}
Type: Dict[str, Any]

A dictionary containing one or more keys that are not the keys used by the *ConfigVar*

test_error_list_int()
Checks that the *ConfigVar* raises a ValueError when passed a str.

test_error_list_str()
Checks that the *ConfigVar* raises a ValueError when passed a str.

test_success()
Checks that the *ConfigVar* can correctly parse various directory values.

test_value
Type: str

A value that is valid and should be returned unchanged.

class BoolTrueTest

Bases: *ConfigVarTest*

Test for boolean configuration values which default to `True`.

Attributes:

different_key_value	A dictionary containing one or more keys that are not the keys used by the *ConfigVar*
false_values	A list of values which should be considered `False` by the *ConfigVar*.
true_values	A list of values which should be considered `True` by the *ConfigVar*.
wrong_values	A list of values which should are of the wrong type.

Methods:

test_empty_get()
test_errors()
test_false()
test_true()

different_key_value = {'username': 'domdfcoding'}

Type: `Dict[str, Any]`

A dictionary containing one or more keys that are not the keys used by the *ConfigVar*

property false_values

A list of values which should be considered `False` by the *ConfigVar*.

Return type `List[Dict[str, Any]]`

test_empty_get()

test_errors()

test_false()

test_true()

property true_values

A list of values which should be considered `True` by the *ConfigVar*.

Return type `List[Dict[str, Any]]`

property wrong_values

A list of values which should are of the wrong type.

Return type `List[Dict[str, Any]]`

class BoolFalseTest

Bases: *BoolTrueTest*

Test for boolean configuration values which default to `False`.

Attributes:

different_key_value	A dictionary containing one or more keys that are not the keys used by the *ConfigVar*
false_values	A list of values which should be considered `False` by the *ConfigVar*.
true_values	A list of values which should be considered `True` by the *ConfigVar*.

Methods:

test_empty_get()

different_key_value = {'username': 'domdfcoding'}

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

A dictionary containing one or more keys that are not the keys used by the *ConfigVar*

property false_values

A list of values which should be considered `False` by the *ConfigVar*.

Return type `List`[`Dict`[`str`, `Any`]]

test_empty_get()

property true_values

A list of values which should be considered `True` by the *ConfigVar*.

Return type `List`[`Dict`[`str`, `Any`]]

class RequiredStringTest

Bases: *ConfigVarTest*

Test for string configuration values which are required.

Methods:

test_empty_get()
test_errors()
test_success()

Attributes:

test_value	A value that is valid and should be returned unchanged.
wrong_values	A list of values which should are of the wrong type.

test_empty_get()

test_errors()

test_success()

test_value
Type: str

A value that is valid and should be returned unchanged.

property wrong_values
A list of values which should are of the wrong type.

Return type List[Dict[str, Any]]

class OptionalStringTest
Bases: *RequiredStringTest*

Test for string configuration values which are optional.

Attributes:

default_value	The default value that should be returned when no valid is given.
different_key_value	A dictionary containing one or more keys that are not the keys used by the *ConfigVar*
wrong_values	A list of values which should are of the wrong type.

Methods:

test_empty_get()
test_errors()
test_success()

default_value = ''
Type: str

The default value that should be returned when no valid is given.

different_key_value = {'sphinx_html_theme': 'alabaster'}
Type: Dict[str, Any]

A dictionary containing one or more keys that are not the keys used by the *ConfigVar*

test_empty_get()

test_errors()

test_success()

property wrong_values
A list of values which should are of the wrong type.

Return type List[Dict[str, Any]]

class EnumTest
Bases: *RequiredStringTest*

Test for Enum configuration values.

Attributes:

default_value	The default value that should be returned when no valid is given.
non_enum_values	A list of values which are of the correct type but are invalid.

Methods:

test_empty_get()
test_errors()
test_non_enum()

default_value
Type: str

The default value that should be returned when no valid is given.

non_enum_values
Type: List[Any]

A list of values which are of the correct type but are invalid.

test_empty_get()

test_errors()

test_non_enum()

class DictTest
Bases: *NotStrTest*, *NotBoolTest*, *NotIntTest*, *ConfigVarTest*

Test for dictionary configuration values.

Attributes:

default_value	The default value that should be returned when no valid is given.
different_key_value	A dictionary containing one or more keys that are not the keys used by the *ConfigVar*
test_value	A value that is valid and should be returned unchanged.

Methods:

test_error_list_int()
test_success()

default_value = {}
Type: Dict[str, Any]

The default value that should be returned when no valid is given.

different_key_value = {'sphinx_html_theme': 'alabaster'}
Type: Dict[str, Any]

A dictionary containing one or more keys that are not the keys used by the *ConfigVar*

test_error_list_int()

test_success()

test_value
Type: Dict[str, Any]

A value that is valid and should be returned unchanged.

Chapter

SEVEN

configconfig.utils

Utility functions.

Functions:

get_literal_values(literal)	Returns a tuple of permitted values for a `typing.Literal`.
optional_getter(raw_config_vars, cls, required)	Returns either the configuration value, the default, or raises an error if the value is required but wasn't supplied.
get_yaml_type(type_)	Get the YAML type that corresponds to the given Python type.
make_schema(*configuration_variables)	Create a `JSON` schema from a list of *ConfigVar* classes.
check_union(obj, dtype)	Check if the type of `obj` is one of the types in a `typing.Union`, `typing.List` etc.
get_json_type(type_)	Get the type for the JSON schema that corresponds to the given Python type.

get_literal_values(*literal*)

Returns a tuple of permitted values for a `typing.Literal`.

New in version 0.3.0.

Parameters **literal** (`Literal`[])

Return type `Tuple`[`Any`]

optional_getter(*raw_config_vars*, *cls*, *required*)

Returns either the configuration value, the default, or raises an error if the value is required but wasn't supplied.

Parameters

- **raw_config_vars** (`Dict`[`str`, `Any`])
- **cls** (*ConfigVarMeta*)
- **required** (`bool`)

Return type `Any`

get_yaml_type(*type_*)

Get the YAML type that corresponds to the given Python type.

Parameters **type_** (`Type`)

Return type `str`

make_schema(**configuration_variables*)
Create a JSON schema from a list of *ConfigVar* classes.

Parameters **configuration_variables**

Return type Dict[str, Any]

Returns Dictionary representation of the JSON schema.

check_union(*obj*, *dtype*)
Check if the type of obj is one of the types in a typing.Union, typing.List etc.

Parameters

- **obj** (Any)
- **dtype** (Union, List, etc.)

Return type bool

get_json_type(*type_*)
Get the type for the JSON schema that corresponds to the given Python type.

Parameters **type_** (Type)

Return type Dict[str, Union[str, List, Dict]]

Chapter

EIGHT

configconfig.validator

Validate values obtained from the YAML file and coerce into the appropriate return types.

Classes:

Validator(config_var)	Methods are named visit_<type>.

Functions:

validate_files(schemafile, *datafiles[, ...])	Validate the given datafiles against the given schema.

class Validator(*config_var*)

Bases: object

Methods are named visit_<type>.

Methods:

unknown_type()	Called when the desired type has no visitor.
validate([raw_config_vars])	Validate the configuration value.
visit_bool(raw_config_vars)	Used to validate and convert bool values.
visit_dict(raw_config_vars)	Used to validate and convert dict values.
visit_float(raw_config_vars)	Used to validate and convert float values.
visit_int(raw_config_vars)	Used to validate and convert int values.
visit_list(raw_config_vars)	Used to validate and convert list values.
visit_literal(raw_config_vars)	Used to validate and convert typing.Literal values.
visit_str(raw_config_vars)	Used to validate and convert str values.
visit_union(raw_config_vars)	Used to validate and convert typing.Union values.

unknown_type()

Called when the desired type has no visitor.

Return type NoReturn

validate(*raw_config_vars=None*)

Validate the configuration value.

Parameters **raw_config_vars** (Optional[Dict[str, Any]]) – Default None.

Return type Any

Returns The validated value.

visit_bool(*raw_config_vars*)
Used to validate and convert `bool` values.

Parameters **raw_config_vars** (`Dict[str, Any]`)

Return type `bool`

visit_dict(*raw_config_vars*)
Used to validate and convert `dict` values.

Parameters **raw_config_vars** (`Dict[str, Any]`)

Return type `Dict`

visit_float(*raw_config_vars*)
Used to validate and convert `float` values.

Parameters **raw_config_vars** (`Dict[str, Any]`)

Return type `float`

visit_int(*raw_config_vars*)
Used to validate and convert `int` values.

Parameters **raw_config_vars** (`Dict[str, Any]`)

Return type `int`

visit_list(*raw_config_vars*)
Used to validate and convert `list` values.

Parameters **raw_config_vars** (`Dict[str, Any]`)

Return type `List`

visit_literal(*raw_config_vars*)
Used to validate and convert `typing.Literal` values.

Parameters **raw_config_vars** (`Dict[str, Any]`)

Return type `Any`

visit_str(*raw_config_vars*)
Used to validate and convert `str` values.

Parameters **raw_config_vars** (`Dict[str, Any]`)

Return type `str`

visit_union(*raw_config_vars*)
Used to validate and convert `typing.Union` values.

Parameters **raw_config_vars** (`Dict[str, Any]`)

Return type `Any`

validate_files(*schemafile*, **datafiles*, *encoding='utf-8'*)

Validate the given datafiles against the given schema.

Parameters

- **schemafile** (Union[str, Path, PathLike]) – The `json` or `yaml` formatted schema to validate with.
- ***datafiles** (Union[str, Path, PathLike]) – The `json` or `yaml` files to validate.
- **encoding** (str) – Encoding to open the files with. Default `'utf-8'`.

New in version 0.4.0.

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

L

M

N

O

P

R

S

T

U

V

W