Validator Collection Documentation

Release 1.3.1

Insight Industry Inc.

Feb 09, 2019

Contents:

1 Validator Reference **3**
- 1.1 Using Validators . . . 3
- 1.2 Core . . . 5
- 1.3 Date / Time . . . 9
- 1.4 Numbers . . . 12
- 1.5 File-related . . . 14
- 1.6 Internet-related . . . 19

2 Checker Reference **25**
- 2.1 Using Checkers . . . 25
- 2.2 Core . . . 27
- 2.3 Date / Time . . . 32
- 2.4 Numbers . . . 33
- 2.5 File-related . . . 35
- 2.6 Internet-related . . . 39

3 Error Reference **41**
- 3.1 Handling Errors . . . 42
- 3.2 Standard Errors . . . 44
- 3.3 Core . . . 45
- 3.4 Date / Time . . . 47
- 3.5 Numbers . . . 47
- 3.6 File-related . . . 48
- 3.7 Internet-related . . . 49

4 Contributing to the Validator Collection **51**
- 4.1 Design Philosophy . . . 52
- 4.2 Style Guide . . . 52
- 4.3 Dependencies . . . 55
- 4.4 Preparing Your Development Environment . . . 55
- 4.5 Ideas and Feature Requests . . . 55
- 4.6 Testing . . . 55
- 4.7 Submitting Pull Requests . . . 56
- 4.8 Building Documentation . . . 56
- 4.9 References . . . 56

5 Testing the Validator Collection **57**

5.1 Testing Philosophy . . . 57
5.2 Test Organization . . . 58
5.3 Configuring & Running Tests . . . 58
5.4 Skipping Tests . . . 59
5.5 Incremental Tests . . . 59

6 Release History **61**
6.1 Release 1.3.1 (released November 30, 2018) . . . 61
6.2 Release 1.3.0 (released November 12, 2018) . . . 62
6.3 Release 1.2.0 (released August 4, 2018) . . . 62
6.4 Release 1.1.0 (released April 23, 2018) . . . 62
6.5 Release 1.0.0 (released April 16, 2018) . . . 63

7 Glossary **65**

8 Installation **67**
8.1 Dependencies . . . 67

9 Available Validators and Checkers **69**

10 Hello, World and Standard Usage **71**
10.1 Using Validators . . . 72
10.2 Using Checkers . . . 74

11 Best Practices **77**
11.1 Defensive Approach: Check, then Convert if Necessary . . . 77
11.2 Confident Approach: try ... except . . . 78

12 Questions and Issues **81**

13 Contributing **83**

14 Testing **85**

15 License **87**

16 Indices and tables **89**

Python Module Index **91**

Python library of 60+ commonly-used validator functions

Version Compatability

The **Validator Collection** is designed to be compatible with Python 2.7 and Python 3.4 or higher.

Branch	Unit Tests
latest	
v. 1.3	
v. 1.2	
v. 1.1	
v. 1.0.0	
develop	

CHAPTER 1

Validator Reference

Core	Date/Time	Numbers	File-related	Internet-related
dict	*date*	*numeric*	*bytesIO*	*email*
json	*datetime*	*integer*	*stringIO*	*url*
string	*time*	*float*	*path*	*domain*
iterable	*timezone*	*fraction*	*path_exists*	*ip_address*
none		*decimal*	*file_exists*	*ipv4*
not_empty			*directory_exists*	*ipv6*
uuid			*readable*	*mac_address*
variable_name			*writeable*	
			executable	

1.1 Using Validators

A validator does what it says on the tin: It validates that an input value is what you think it should be, and returns its valid form.

Each validator is expressed as the name of the thing being validated, for example *email()*.

Each validator accepts a value as its first argument, and an optional `allow_empty` boolean as its second argument. For example:

```
email_address = validators.email(value, allow_empty = True)
```

If the value you're validating validates successfully, it will be returned. If the value you're validating needs to be coerced to a different type, the validator will try to do that. So for example:

```
validators.integer(1)
validators.integer('1')
```

will both return an `int` of `1`.

If the value you're validating is empty/falsey and `allow_empty` is `False`, then the validator will raise a *EmptyValueError* exception (which inherits from the built-in `ValueError`). If `allow_empty` is `True`, then an empty/falsey input value will be converted to a `None` value.

Caution: By default, `allow_empty` is always set to `False`.

Hint: Some validators (particularly numeric ones like *integer*) have additional options which are used to make sure the value meets criteria that you set for it. These options are always included as keyword arguments *after* the `allow_empty` argument, and are documented for each validator below.

1.1.1 When Validation Fails

Validators raise exceptions when validation fails. All exceptions raised inherit from built-in exceptions like `ValueError`, `TypeError`, and `IOError`.

If the value you're validating fails its validation for some reason, the validator may raise different exceptions depending on the reason. In most cases, this will be a descendent of `ValueError` though it can sometimes be a `TypeError`, or an `IOError`, etc.

For specifics on each validator's likely exceptions and what can cause them, please review the *Validator Reference*.

Hint: While validators will always raise built-in exceptions from the standard library, to give you greater programmatic control over how to respond when validation fails, we have defined a set of custom exceptions that inherit from those built-ins.

Our custom exceptions provide you with very specific, fine-grained information as to *why* validation for a given value failed. In general, most validators will raise `ValueError` or `TypeError` exceptions, and you can safely catch those and be fine. But if you want to handle specific types of situations with greater control, then you can instead catch *EmptyValueError*, *CannotCoerceError*, *MaximumValueError*, and the like.

For more detailed information, please see: *Error Reference* and *Validator Reference*.

1.1.2 Disabling Validation

Caution: If you are disabling validators using the `VALIDATORS_DISABLED` environment variable, their related *checkers* will **also** be disabled (meaning they will always return `True`).

Validation can at times be an expensive (in terms of performance) operation. As a result, there are times when you want to disable certain kinds of validation when running in production. Using the **Validator-Collection** this is simple:

> Just add the name of the validator you want disabled to the `VALIDATORS_DISABLED` environment variable, and validation will automatically be skipped.

Caution: `VALIDATORS_DISABLED` expects a comma-separated list of values. If it isn't comma-separated, it won't work properly.

Here's how it works in practice. Let's say we define the following environment variable:

```
$ export VALIDATORS_DISABLED = "variable_name, email, ipv4"
```

This disables the *variable_name()*, *email()*, and *ipv4()* validators respectively.

Now if we run:

```
from validator_collection import validators, errors

try:
    result = validators.variable_name('this is an invalid variable name')
except ValueError:
    # handle the error
```

The validator will return the `value` supplied to it un-changed. So that means `result` will be equal to `this is an invalid variable name`.

However, if we run:

```
from validator_collection import validators, errors

try:
    result = validators.integer('this is an invalid variable name')
except errors.NotAnIntegerError:
    # handle the error
```

the validator will run and raise *NotAnIntegerError*.

We can force validators to run (even if disabled using the environment variable) by passing a `force_run = True` keyword argument. For example:

```
from validator_collection import validators, errors

try:
    result = validators.variable_name('this is an invalid variable name',
                                      force_run = True)
except ValueError:
    # handle the error
```

will produce a *InvalidVariableNameError* (which is a type of `ValueError`).

1.2 Core

1.2.1 dict

dict(*value*, *allow_empty=False*, *json_serializer=None*, ***kwargs*)
Validate that `value` is a `dict`.

> **Hint:** If `value` is a string, this validator will assume it is a JSON object and try to convert it into a `dict`
>
> You can override the JSON serializer used by passing it to the `json_serializer` property. By default, will utilize the Python *json* encoder/decoder.

Parameters

- **value** – The value to validate.
- **allow_empty** (bool) – If True, returns None if value is empty. If False, raises a *EmptyValueError* if value is empty. Defaults to False.
- **json_serializer** (*callable*) – The JSON encoder/decoder to use to deserialize a string passed in value. If not supplied, will default to the Python json encoder/decoder.

Returns value / None

Return type dict / None

Raises

- ***EmptyValueError*** – if value is empty and allow_empty is False
- ***CannotCoerceError*** – if value cannot be coerced to a dict
- ***NotADictError*** – if value is not a dict

1.2.2 json

json(*value*, *schema=None*, *allow_empty=False*, *json_serializer=None*, ***kwargs*)

Validate that value conforms to the supplied JSON Schema.

Note: schema supports JSON Schema Drafts 3 - 7. Unless the JSON Schema indicates the meta-schema using a $schema property, the schema will be assumed to conform to Draft 7.

Hint: If either value or schema is a string, this validator will assume it is a JSON object and try to convert it into a dict.

You can override the JSON serializer used by passing it to the json_serializer property. By default, will utilize the Python *json* encoder/decoder.

Parameters

- **value** – The value to validate.
- **schema** – An optional JSON Schema against which value will be validated.
- **allow_empty** (bool) – If True, returns None if value is empty. If False, raises a *EmptyValueError* if value is empty. Defaults to False.
- **json_serializer** (*callable*) – The JSON encoder/decoder to use to deserialize a string passed in value. If not supplied, will default to the Python json encoder/decoder.

Returns value / None

Return type dict / list of dict / None

Raises

- ***EmptyValueError*** – if value is empty and allow_empty is False
- ***CannotCoerceError*** – if value cannot be coerced to a dict
- ***NotJSONError*** – if value cannot be deserialized from JSON
- ***NotJSONSchemaError*** – if schema is not a valid JSON Schema object
- ***JSONValidationError*** – if value does not validate against the JSON Schema

1.2.3 string

string(*value*, *allow_empty=False*, *coerce_value=False*, *minimum_length=None*, *maximum_length=None*, *whitespace_padding=False*, ***kwargs*)

Validate that `value` is a valid string.

Parameters

- **value** (`str` / `None`) – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *`EmptyValueError`* if `value` is empty. Defaults to `False`.
- **coerce_value** (`bool`) – If `True`, will attempt to coerce `value` to a string if it is not already. If `False`, will raise a `ValueError` if `value` is not a string. Defaults to `False`.
- **minimum_length** (`int`) – If supplied, indicates the minimum number of characters needed to be valid.
- **maximum_length** (`int`) – If supplied, indicates the minimum number of characters needed to be valid.
- **whitespace_padding** (`bool`) – If `True` and the value is below the `minimum_length`, pad the value with spaces. Defaults to `False`.

Returns `value` / `None`

Return type `str` / `None`

Raises

- ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`
- ***CannotCoerceError*** – if `value` is not a valid string and `coerce_value` is `False`
- ***MinimumLengthError*** – if `minimum_length` is supplied and the length of `value` is less than `minimum_length` and `whitespace_padding` is `False`
- ***MaximumLengthError*** – if `maximum_length` is supplied and the length of `value` is more than the `maximum_length`

1.2.4 iterable

iterable(*value*, *allow_empty=False*, *forbid_literals=(<class 'str'>, <class 'bytes'>)*, *minimum_length=None*, *maximum_length=None*, ***kwargs*)

Validate that `value` is a valid iterable.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *`EmptyValueError`* if `value` is empty. Defaults to `False`.
- **forbid_literals** (*`iterable`*) – A collection of literals that will be considered invalid even if they are (actually) iterable. Defaults to `str` and `bytes`.
- **minimum_length** (`int`) – If supplied, indicates the minimum number of members needed to be valid.
- **maximum_length** (`int`) – If supplied, indicates the minimum number of members needed to be valid.

Returns `value` / `None`

Return type iterable / `None`

Raises

- ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`
- ***NotAnIterableError*** – if `value` is not a valid iterable or `None`
- ***MinimumLengthError*** – if `minimum_length` is supplied and the length of `value` is less than `minimum_length` and `whitespace_padding` is `False`
- ***MaximumLengthError*** – if `maximum_length` is supplied and the length of `value` is more than the `maximum_length`

1.2.5 none

none(*value*, *allow_empty=False*, ***kwargs*)

Validate that `value` is `None`.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty but **not** `None`. If `False`, raises a `NotNoneError` if `value` is empty but **not** `None`. Defaults to `False`.

Returns `None`

Raises ***NotNoneError*** – if `allow_empty` is `False` and `value` is empty but **not** `None` and

1.2.6 not_empty

not_empty(*value*, *allow_empty=False*, ***kwargs*)

Validate that `value` is not empty.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *`EmptyValueError`* if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Raises ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`

1.2.7 uuid

uuid(*value*, *allow_empty=False*, ***kwargs*)

Validate that `value` is a valid `UUID`.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *`EmptyValueError`* if `value` is empty. Defaults to `False`.

Returns `value` coerced to a `UUID` object / `None`

Return type `UUID` / `None`

Raises

- ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`
- ***CannotCoerceError*** – if `value` cannot be coerced to a `UUID`

1.2.8 variable_name

variable_name(*value*, *allow_empty=False*, ***kwargs*)

Validate that the value is a valid Python variable name.

> **Caution:** This function does **NOT** check whether the variable exists. It only checks that the `value` would work as a Python variable (or class, or function, etc.) name.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *`EmptyValueError`* if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Return type `str` or `None`

Raises ***EmptyValueError*** – if `allow_empty` is `False` and `value` is empty

---

1.3 Date / Time

1.3.1 date

date(*value*, *allow_empty=False*, *minimum=None*, *maximum=None*, *coerce_value=True*, ***kwargs*)

Validate that `value` is a valid date.

Parameters

- **value** (`str` / `datetime` / `date` / `None`) – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *`EmptyValueError`* if `value` is empty. Defaults to `False`.
- **minimum** (`datetime` / `date` / compliant `str` / `None`) – If supplied, will make sure that `value` is on or after this value.
- **maximum** (`datetime` / `date` / compliant `str` / `None`) – If supplied, will make sure that `value` is on or before this value.
- **coerce_value** (`bool`) – If `True`, will attempt to coerce `value` to a `date` if it is a timestamp value. If `False`, will not.

Returns `value` / `None`

Return type `date` / `None`

Raises

- ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`
- ***CannotCoerceError*** – if `value` cannot be coerced to a `date` and is not `None`
- ***MinimumValueError*** – if `minimum` is supplied but `value` occurs before `minimum`
- ***MaximumValueError*** – if `maximum` is supplied but `value` occurs after `maximum`

1.3.2 datetime

datetime(*value*, *allow_empty=False*, *minimum=None*, *maximum=None*, *coerce_value=True*, ***kwargs*)

Validate that `value` is a valid datetime.

> **Caution:** If supplying a string, the string needs to be in an ISO 8601-format to pass validation. If it is not in an ISO 8601-format, validation will fail.

Parameters

- **value** (`str` / `datetime` / `date` / `None`) – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.
- **minimum** (`datetime` / `date` / compliant `str` / `None`) – If supplied, will make sure that `value` is on or after this value.
- **maximum** (`datetime` / `date` / compliant `str` / `None`) – If supplied, will make sure that `value` is on or before this value.
- **coerce_value** (`bool`) – If `True`, will coerce dates to `datetime` objects with times of 00:00:00. If `False`, will error if `value` is not an unambiguous timestamp. Defaults to `True`.

Returns `value` / `None`

Return type `datetime` / `None`

Raises

- ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`
- ***CannotCoerceError*** – if `value` cannot be coerced to a `datetime` value and is not `None`
- ***MinimumValueError*** – if `minimum` is supplied but `value` occurs before `minimum`
- ***MaximumValueError*** – if `maximum` is supplied but `value` occurs after `minimum`

1.3.3 time

time(*value*, *allow_empty=False*, *minimum=None*, *maximum=None*, *coerce_value=True*, ***kwargs*)

Validate that `value` is a valid `time`.

> **Caution:** This validator will **always** return the time as timezone naive (effectively UTC). If `value` has a timezone / UTC offset applied, the validator will coerce the value returned back to UTC.

Parameters

- **value** (*datetime* or *time*-compliant str / datetime / time) – The value to validate.
- **allow_empty** (bool) – If True, returns None if value is empty. If False, raises a *EmptyValueError* if value is empty. Defaults to False.
- **minimum** (*datetime* or *time*-compliant str / datetime / time) – If supplied, will make sure that value is on or after this value.
- **maximum** (*datetime* or *time*-compliant str / datetime / time) – If supplied, will make sure that value is on or before this value.
- **coerce_value** (bool) – If True, will attempt to coerce/extract a time from value. If False, will only respect direct representations of time. Defaults to True.

Returns value in UTC time / None

Return type time / None

Raises

- ***EmptyValueError*** – if value is empty and allow_empty is False
- ***CannotCoerceError*** – if value cannot be coerced to a time and is not None
- ***MinimumValueError*** – if minimum is supplied but value occurs before minimum
- ***MaximumValueError*** – if maximum is supplied but value occurs after minimum

1.3.4 timezone

timezone(*value*, *allow_empty=False*, *positive=True*, ***kwargs*)

Validate that value is a valid tzinfo.

> **Caution:** This does **not** verify whether the value is a timezone that actually exists, nor can it resolve timezone names (e.g. 'Eastern' or 'CET').
>
> For that kind of functionality, we recommend you utilize: pytz

Parameters

- **value** (str / tzinfo / numeric / None) – The value to validate.
- **allow_empty** (bool) – If True, returns None if value is empty. If False, raises a *EmptyValueError* if value is empty. Defaults to False.
- **positive** (bool) – Indicates whether the value is positive or negative (only has meaning if value is a string). Defaults to True.

Returns value / None

Return type tzinfo / None

Raises

- ***EmptyValueError*** – if value is empty and allow_empty is False
- ***CannotCoerceError*** – if value cannot be coerced to tzinfo and is not None
- ***PositiveOffsetMismatchError*** – if positive is True, but the offset indicated by value is actually negative
- ***NegativeOffsetMismatchError*** – if positive is False, but the offset indicated by value is actually positive

1.4 Numbers

Note: Because Python's `None` is implemented as an integer value, numeric validators do not check "falsiness". Doing so would find false positives if `value` were set to `0`.

Instead, all numeric validators explicitly check for the Python global singleton `None`.

1.4.1 numeric

numeric(*value*, *allow_empty=False*, *minimum=None*, *maximum=None*, ***kwargs*)

Validate that `value` is a numeric value.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is `None`. If `False`, raises an *`EmptyValueError`* if `value` is `None`. Defaults to `False`.
- **minimum** (*`numeric`*) – If supplied, will make sure that `value` is greater than or equal to this value.
- **maximum** (*`numeric`*) – If supplied, will make sure that `value` is less than or equal to this value.

Returns `value` / `None`

Raises

- ***EmptyValueError*** – if `value` is `None` and `allow_empty` is `False`
- ***MinimumValueError*** – if `minimum` is supplied and `value` is less than the `minimum`
- ***MaximumValueError*** – if `maximum` is supplied and `value` is more than the `maximum`
- ***CannotCoerceError*** – if `value` cannot be coerced to a numeric form

1.4.2 integer

integer(*value*, *allow_empty=False*, *coerce_value=False*, *minimum=None*, *maximum=None*, *base=10*, ***kwargs*)

Validate that `value` is an `int`.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is `None`. If `False`, raises a *`EmptyValueError`* if `value` is `None`. Defaults to `False`.
- **coerce_value** (`bool`) – If `True`, will force any numeric `value` to an integer (always rounding up). If `False`, will raise an error if `value` is numeric but not a whole number. Defaults to `False`.

- **minimum** (*numeric*) – If supplied, will make sure that `value` is greater than or equal to this value.
- **maximum** (*numeric*) – If supplied, will make sure that `value` is less than or equal to this value.
- **base** – Indicates the base that is used to determine the integer value. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with `0b/0B`, `0o/0O/0`, or `0x/0X`, as with integer literals in code. Base 0 means to interpret the string exactly as an integer literal, so that the actual base is 2, 8, 10, or 16. Defaults to `10`.

Returns `value` / `None`

Raises

- ***EmptyValueError*** – if `value` is `None` and `allow_empty` is `False`
- ***MinimumValueError*** – if `minimum` is supplied and `value` is less than the `minimum`
- ***MaximumValueError*** – if `maximum` is supplied and `value` is more than the `maximum`
- ***NotAnIntegerError*** – if `coerce_value` is `False`, and `value` is not an integer
- ***CannotCoerceError*** – if `value` cannot be coerced to an `int`

1.4.3 float

float(*value*, *allow_empty=False*, *minimum=None*, *maximum=None*, ***kwargs*)

Validate that `value` is a `float`.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is `None`. If `False`, raises a *EmptyValueError* if `value` is `None`. Defaults to `False`.

Returns `value` / `None`

Return type `float` / `None`

Raises

- ***EmptyValueError*** – if `value` is `None` and `allow_empty` is `False`
- ***MinimumValueError*** – if `minimum` is supplied and `value` is less than the `minimum`
- ***MaximumValueError*** – if `maximum` is supplied and `value` is more than the `maximum`
- ***CannotCoerceError*** – if unable to coerce `value` to a `float`

1.4.4 fraction

fraction(*value*, *allow_empty=False*, *minimum=None*, *maximum=None*, ***kwargs*)

Validate that `value` is a `Fraction`.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is `None`. If `False`, raises a *EmptyValueError* if `value` is `None`. Defaults to `False`.

Returns `value` / `None`

Return type Fraction / None

Raises

- ***EmptyValueError*** – if value is None and allow_empty is False
- ***MinimumValueError*** – if minimum is supplied and value is less than the minimum
- ***MaximumValueError*** – if maximum is supplied and value is more than the maximum
- ***CannotCoerceError*** – if unable to coerce value to a Fraction

1.4.5 decimal

decimal(*value*, *allow_empty=False*, *minimum=None*, *maximum=None*, ***kwargs*)

Validate that value is a Decimal.

Parameters

- **value** – The value to validate.
- **allow_empty** (bool) – If True, returns None if value is None. If False, raises a *EmptyValueError* if value is None. Defaults to False.
- **minimum** (*numeric*) – If supplied, will make sure that value is greater than or equal to this value.
- **maximum** (*numeric*) – If supplied, will make sure that value is less than or equal to this value.

Returns value / None

Return type Decimal / None

Raises

- ***EmptyValueError*** – if value is None and allow_empty is False
- ***MinimumValueError*** – if minimum is supplied and value is less than the minimum
- ***MaximumValueError*** – if maximum is supplied and value is more than the maximum
- ***CannotCoerceError*** – if unable to coerce value to a Decimal

---

1.5 File-related

1.5.1 bytesIO

bytesIO(*value*, *allow_empty=False*, ***kwargs*)

Validate that value is a BytesIO object.

Parameters

- **value** – The value to validate.
- **allow_empty** (bool) – If True, returns None if value is empty. If False, raises a *EmptyValueError* if value is empty. Defaults to False.

Returns value / None

Return type BytesIO / None

Raises

- ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`
- ***NotBytesIOError*** – if `value` is not a `BytesIO` object.

1.5.2 stringIO

stringIO(*value*, *allow_empty=False*, ***kwargs*)
Validate that `value` is a `StringIO` object.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Return type `StringIO` / `None`

Raises

- ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`
- ***NotStringIOError*** – if `value` is not a `StringIO` object

1.5.3 path

path(*value*, *allow_empty=False*, ***kwargs*)
Validate that `value` is a valid path-like object.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.

Returns The path represented by `value`.

Return type Path-like object / `None`

Raises

- ***EmptyValueError*** – if `allow_empty` is `False` and `value` is empty
- ***NotPathlikeError*** – if `value` is not a valid path-like object

1.5.4 path_exists

path_exists(*value*, *allow_empty=False*, ***kwargs*)
Validate that `value` is a path-like object that exists on the local filesystem.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.

Returns The file name represented by `value`.

Return type Path-like object / `None`

Raises

- ***EmptyValueError*** – if `allow_empty` is `False` and `value` is empty
- ***NotPathlikeError*** – if `value` is not a path-like object
- ***PathExistsError*** – if `value` does not exist

1.5.5 file_exists

file_exists(*value*, *allow_empty=False*, ***kwargs*)

Validate that `value` is a valid file that exists on the local filesystem.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.

Returns The file name represented by `value`.

Return type Path-like object / `None`

Raises

- ***EmptyValueError*** – if `allow_empty` is `False` and `value` is empty
- ***NotPathlikeError*** – if `value` is not a path-like object
- ***PathExistsError*** – if `value` does not exist on the local filesystem
- ***NotAFileError*** – if `value` is not a valid file

1.5.6 directory_exists

directory_exists(*value*, *allow_empty=False*, ***kwargs*)

Validate that `value` is a valid directory that exists on the local filesystem.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.

Returns The file name represented by `value`.

Return type Path-like object / `None`

Raises

- ***EmptyValueError*** – if `allow_empty` is `False` and `value` is empty
- ***NotPathlikeError*** – if `value` is not a path-like object
- ***PathExistsError*** – if `value` does not exist on the local filesystem
- ***NotADirectoryError*** – if `value` is not a valid directory

1.5.7 readable

readable(*value*, *allow_empty=False*, ***kwargs*)
Validate that `value` is a path to a readable file.

> **Caution: Use of this validator is an anti-pattern and should be used with caution.**
>
> Validating the readability of a file *before* attempting to read it exposes your code to a bug called TOCTOU.
>
> This particular class of bug can expose your code to **security vulnerabilities** and so this validator should only be used if you are an advanced user.
>
> A better pattern to use when reading from a file is to apply the principle of EAFP ("easier to ask forgiveness than permission"), and simply attempt to write to the file using a `try ... except` block:

```
try:
    with open('path/to/filename.txt', mode = 'r') as file_object:
        # read from file here
except (OSError, IOError) as error:
    # Handle an error if unable to write.
```

Parameters

- **value** (*Path-like object*) – The path to a file on the local filesystem whose readability is to be validated.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.

Returns Validated path-like object or `None`

Return type Path-like object or `None`

Raises

- ***EmptyValueError*** – if `allow_empty` is `False` and `value` is empty
- ***NotPathlikeError*** – if `value` is not a path-like object
- ***PathExistsError*** – if `value` does not exist on the local filesystem
- ***NotAFileError*** – if `value` is not a valid file
- ***NotReadableError*** – if `value` cannot be opened for reading

1.5.8 writeable

writeable(*value*, *allow_empty=False*, ***kwargs*)
Validate that `value` is a path to a writeable file.

> **Caution:** This validator does **NOT** work correctly on a Windows file system. This is due to the vagaries of how Windows manages its file system and the various ways in which it can manage file permission.
>
> If called on a Windows file system, this validator will raise `NotImplementedError()`.

Caution: Use of this validator is an anti-pattern and should be used with caution.

Validating the writability of a file *before* attempting to write to it exposes your code to a bug called TOCTOU.

This particular class of bug can expose your code to **security vulnerabilities** and so this validator should only be used if you are an advanced user.

A better pattern to use when writing to file is to apply the principle of EAFP ("easier to ask forgiveness than permission"), and simply attempt to write to the file using a `try ... except` block:

```
try:
    with open('path/to/filename.txt', mode = 'a') as file_object:
        # write to file here
except (OSError, IOError) as error:
    # Handle an error if unable to write.
```

Note: This validator relies on `os.access()` to check whether `value` is writeable. This function has certain limitations, most especially that:

- It will **ignore** file-locking (yielding a false-positive) if the file is locked.
- It focuses on *local operating system permissions*, which means if trying to access a path over a network you might get a false positive or false negative (because network paths may have more complicated authentication methods).

Parameters

- **value** (*Path-like object*) – The path to a file on the local filesystem whose writeability is to be validated.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.

Returns Validated absolute path or `None`

Return type Path-like object or `None`

Raises

- ***EmptyValueError*** – if `allow_empty` is `False` and `value` is empty
- **NotImplementedError** – if used on a Windows system
- ***NotPathlikeError*** – if `value` is not a path-like object
- ***NotWriteableError*** – if `value` cannot be opened for writing

1.5.9 executable

executable(*value*, *allow_empty=False*, ***kwargs*)

Validate that `value` is a path to an executable file.

Caution: This validator does **NOT** work correctly on a Windows file system. This is due to the vagaries of how Windows manages its file system and the various ways in which it can manage file permission.

If called on a Windows file system, this validator will raise `NotImplementedError()`.

Caution: Use of this validator is an anti-pattern and should be used with caution.

Validating the executability of a file *before* attempting to execute it exposes your code to a bug called TOCTOU.

This particular class of bug can expose your code to **security vulnerabilities** and so this validator should only be used if you are an advanced user.

A better pattern to use when writing to file is to apply the principle of EAFP ("easier to ask forgiveness than permission"), and simply attempt to execute the file using a `try ... except` block.

Note: This validator relies on `os.access()` to check whether `value` is executable. This function has certain limitations, most especially that:

- It will **ignore** file-locking (yielding a false-positive) if the file is locked.
- It focuses on *local operating system permissions*, which means if trying to access a path over a network you might get a false positive or false negative (because network paths may have more complicated authentication methods).

Parameters

- **value** (*Path-like object*) – The path to a file on the local filesystem whose writeability is to be validated.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.

Returns Validated absolute path or `None`

Return type Path-like object or `None`

Raises

- ***EmptyValueError*** – if `allow_empty` is `False` and `value` is empty
- **NotImplementedError** – if used on a Windows system
- ***NotPathlikeError*** – if `value` is not a path-like object
- ***NotAFileError*** – if `value` does not exist on the local file system
- ***NotExecutableError*** – if `value` cannot be executed

1.6 Internet-related

1.6.1 email

email(*value*, *allow_empty=False*, ***kwargs*)

Validate that `value` is a valid email address.

Note: Email address validation is…complicated. The methodology that we have adopted here is *generally* compliant with RFC 5322 and uses a combination of string parsing and regular expressions.

String parsing in particular is used to validate certain *highly unusual* but still valid email patterns, including the use of escaped text and comments within an email address' local address (the user name part).

This approach ensures more complete coverage for unusual edge cases, while still letting us use regular expressions that perform quickly.

Parameters

- **value** (str / None) – The value to validate.
- **allow_empty** (bool) – If True, returns None if value is empty. If False, raises a *EmptyValueError* if value is empty. Defaults to False.

Returns value / None

Return type str / None

Raises

- ***EmptyValueError*** – if value is empty and allow_empty is False
- ***CannotCoerceError*** – if value is not a str or None
- ***InvalidEmailError*** – if value is not a valid email address or empty with allow_empty set to True

1.6.2 url

url(*value*, *allow_empty=False*, ***kwargs*)

Validate that value is a valid URL.

Parameters

- **value** (str / None) – The value to validate.
- **allow_empty** (bool) – If True, returns None if value is empty. If False, raises a *EmptyValueError* if value is empty. Defaults to False.

Returns value / None

Return type str / None

Raises

- ***EmptyValueError*** – if value is empty and allow_empty is False
- ***CannotCoerceError*** – if value is not a str or None
- ***InvalidURLError*** – if value is not a valid URL or empty with allow_empty set to True

1.6.3 domain

domain(*value*, *allow_empty=False*, ***kwargs*)

Validate that value is a valid domain name.

> **Caution:** This validator does not verify that value **exists** as a domain. It merely verifies that its contents *might* exist as a domain.

Note: This validator checks to validate that `value` resembles a valid domain name. It is - generally - compliant with RFC 1035, however it diverges in a number of key ways:

- Including authentication (e.g. `username:password@domain.dev`) will fail validation.
- Including a path (e.g. `domain.dev/path/to/file`) will fail validation.
- Including a port (e.g. `domain.dev:8080`) will fail validation.

If you are hoping to validate a more complete URL, we recommend that you see *`url`*.

Hint: Leading and trailing whitespace will be automatically stripped.

Parameters

- **`value`** (`str` / `None`) – The value to validate.
- **`allow_empty`** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *`EmptyValueError`* if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Return type `str` / `None`

Raises

- ***`EmptyValueError`*** – if `value` is empty and `allow_empty` is `False`
- ***`CannotCoerceError`*** – if `value` is not a `str` or `None`
- ***`InvalidDomainError`*** – if `value` is not a valid domain name or empty with `allow_empty` set to `True`
- ***`SlashInDomainError`*** – if `value` contains a slash or backslash
- ***`AtInDomainError`*** – if `value` contains an `@` symbol
- ***`ColonInDomainError`*** – if `value` contains a `:` symbol
- ***`WhitespaceInDomainError`*** – if `value` contains whitespace

1.6.4 ip_address

`ip_address`(*value*, *allow_empty=False*, ***kwargs*)

Validate that `value` is a valid IP address.

Note: First, the validator will check if the address is a valid IPv6 address. If that doesn't work, the validator will check if the address is a valid IPv4 address.

If neither works, the validator will raise an error (as always).

Parameters

- **`value`** – The value to validate.
- **`allow_empty`** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *`EmptyValueError`* if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Raises

- ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`
- ***InvalidIPAddressError*** – if `value` is not a valid IP address or empty with `allow_empty` set to `True`

1.6.5 ipv4

ipv4(*value*, *allow_empty=False*)

Validate that `value` is a valid IP version 4 address.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Raises

- ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`
- ***InvalidIPAddressError*** – if `value` is not a valid IP version 4 address or empty with `allow_empty` set to `True`

1.6.6 ipv6

ipv6(*value*, *allow_empty=False*, ***kwargs*)

Validate that `value` is a valid IP address version 6.

Parameters

- **value** – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a *EmptyValueError* if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Raises

- ***EmptyValueError*** – if `value` is empty and `allow_empty` is `False`
- ***InvalidIPAddressError*** – if `value` is not a valid IP version 6 address or empty with `allow_empty` is not set to `True`

1.6.7 mac_address

mac_address(*value*, *allow_empty=False*, ***kwargs*)

Validate that `value` is a valid MAC address.

Parameters

- **value** (`str` / `None`) – The value to validate.

- **allow_empty** (bool) – If True, returns None if value is empty. If False, raises a *EmptyValueError* if value is empty. Defaults to False.

Returns value / None

Return type str / None

Raises

- ***EmptyValueError*** – if value is empty and allow_empty is False
- ***CannotCoerceError*** – if value is not a valid str or string-like object
- ***InvalidMACAddressError*** – if value is not a valid MAC address or empty with allow_empty set to True

CHAPTER 2

Checker Reference

Core	Date/Time	Numbers	File-related	Internet-related
is_type	*is_date*	*is_numeric*	*is_bytesIO*	*is_email*
is_between	*is_datetime*	*is_integer*	*is_stringIO*	*is_url*
has_length	*is_time*	*is_float*	*is_pathlike*	*is_domain*
are_equivalent	*is_timezone*	*is_fraction*	*is_on_filesystem*	*is_ip_address*
are_dicts_equivalent		*is_decimal*	*is_file*	*is_ipv4*
is_dict			*is_directory*	*is_ipv6*
is_json			*is_readable*	*is_mac_address*
is_string			*is_writeable*	
is_iterable			*is_executable*	
is_not_empty				
is_none				
is_callable				
is_uuid				
is_variable_name				

2.1 Using Checkers

A *checker* is what it sounds like: It checks that an input value is what you expect it to be, and tells you `True`/`False` whether it is or not.

Important: Checkers do *not* verify or convert object types. You can think of a checker as a tool that tells you whether its corresponding *validator* would fail. See *Best Practices* for tips and tricks on using the two together.

Each checker is expressed as the name of the thing being validated, prefixed by `is_`. So the checker for an email address is *`is_email()`* and the checker for an integer is *`is_integer()`*.

Checkers take the input value you want to check as their first (and often only) positional argumet. If the input value

validates, they will return `True`. Unlike *validators*, checkers will not raise an exception if validation fails. They will instead return `False`.

Hint: If you need to know *why* a given value failed to validate, use the validator instead.

Hint: Some checkers (particularly numeric ones like *is_integer*) have additional options which are used to make sure the value meets criteria that you set for it. These options are always *optional* and are included as keyword arguments *after* the input value argument. For details, please see the *Checker Reference*.

2.1.1 Disabling Checking

Caution: If you are disabling validators using the `VALIDATORS_DISABLED` environment variable, their related checkers will **also** be disabled. This means they will always return `True` unless you call them using `force_run = True`.

Checking can at times be an expensive (in terms of performance) operation. As a result, there are times when you want to disable certain kinds of checking when running in production. Using the **Validator-Collection** this is simple:

> Just add the name of the checker you want disabled to the `CHECKERS_DISABLED` environment variable, and validation will automatically be skipped.

Caution: `CHECKERS_DISABLED` expects a comma-separated list of values. If it isn't comma-separated, it won't work properly.

Here's how it works in practice. Let's say we define the following environment variable:

```
$ export CHECKERS_DISABLED = "is_variable_name, is_email, is_ipv4"
```

This disables the *is_variable_name()*, *is_email()*, and *is_ipv4()* validators respectively.

Now if we run:

```
from validator_collection import checkers, errors

result = checkers.is_variable_name('this is an invalid variable name')
# result will be True
```

The checker will return `True`.

However, if we run:

```
from validator_collection import checkers

result = validators.is_integer('this is an invalid variable name')
# result will be False
```

the checker will return `False`

We can force checkers to run (even if disabled using the environment variable) by passing a `force_run = True` keyword argument. For example:

```
from validator_collection import checkers, errors

result = checkers.is_variable_name('this is an invalid variable name',
                                   force_run = True)
# result will be False
```

will return `False`.

2.2 Core

2.2.1 is_type

is_type(*obj*, *type_*, ***kwargs*)
Indicate if `obj` is a type in `type_`.

Hint: This checker is particularly useful when you want to evaluate whether `obj` is of a particular type, but importing that type directly to use in `isinstance()` would cause a circular import error.

To use this checker in that kind of situation, you can instead pass the *name* of the type you want to check as a string in `type_`. The checker will evaluate it and see whether `obj` is of a type or inherits from a type whose name matches the string you passed.

Parameters

- **obj** (`object`) – The object whose type should be checked.
- **type** (`type` / iterable of `type` / `str` with type name / iterable of `str` with type name) – The type(s) to check against.

Returns `True` if `obj` is a type in `type_`. Otherwise, `False`.

Return type `bool`

2.2.2 are_equivalent

are_equivalent(**args*, ***kwargs*)
Indicate if arguments passed to this function are equivalent.

Hint: This checker operates recursively on the members contained within iterables and `dict` objects.

Caution: If you only pass one argument to this checker - even if it is an iterable - the checker will *always* return `True`.

To evaluate members of an iterable for equivalence, you should instead unpack the iterable into the function like so:

```
obj = [1, 1, 1, 2]

result = are_equivalent(*obj)
# Will return ``False`` by unpacking and evaluating the iterable's members

result = are_equivalent(obj)
# Will always return True
```

Parameters **args** – One or more values, passed as positional arguments.

Returns `True` if `args` are equivalent, and `False` if not.

Return type `bool`

2.2.3 are_dicts_equivalent

are_dicts_equivalent(**args*, ***kwargs*)

Indicate if dicts passed to this function have identical keys and values.

Parameters **args** – One or more values, passed as positional arguments.

Returns `True` if `args` have identical keys/values, and `False` if not.

Return type `bool`

2.2.4 is_between

is_between(*value*, *minimum=None*, *maximum=None*, ***kwargs*)

Indicate whether `value` is greater than or equal to a supplied `minimum` and/or less than or equal to `maximum`.

Note: This function works on any `value` that support comparison operators, whether they are numbers or not. Technically, this means that `value`, `minimum`, or `maximum` need to implement the Python magic methods `__lte__` and `__gte__`.

If `value`, `minimum`, or `maximum` do not support comparison operators, they will raise `NotImplemented`.

Parameters

- **value** (*anything that supports comparison operators*) – The `value` to check.
- **minimum** (anything that supports comparison operators / `None`) – If supplied, will return `True` if `value` is greater than or equal to this value.
- **maximum** (anything that supports comparison operators / `None`) – If supplied, will return `True` if `value` is less than or equal to this value.

Returns `True` if `value` is greater than or equal to a supplied `minimum` and less than or equal to a supplied `maximum`. Otherwise, returns `False`.

Return type `bool`

Raises

- **NotImplemented** – if `value`, `minimum`, or `maximum` do not support comparison operators
- **ValueError** – if both `minimum` and `maximum` are `None`

2.2.5 has_length

has_length(*value*, *minimum=None*, *maximum=None*, ***kwargs*)
Indicate whether `value` has a length greater than or equal to a supplied `minimum` and/or less than or equal to `maximum`.

Note: This function works on any `value` that supports the `len()` operation. This means that `value` must implement the `__len__` magic method.

If `value` does not support length evaluation, the checker will raise `NotImplemented`.

Parameters

- **value** (*anything that supports length evaluation*) – The `value` to check.
- **minimum** (*numeric*) – If supplied, will return `True` if `value` is greater than or equal to this value.
- **maximum** (*numeric*) – If supplied, will return `True` if `value` is less than or equal to this value.

Returns `True` if `value` has length greater than or equal to a supplied `minimum` and less than or equal to a supplied `maximum`. Otherwise, returns `False`.

Return type `bool`

Raises

- **TypeError** – if `value` does not support length evaluation
- **ValueError** – if both `minimum` and `maximum` are `None`

2.2.6 is_dict

is_dict(*value*, ***kwargs*)
Indicate whether `value` is a valid `dict`

Note: This will return `True` even if `value` is an empty `dict`.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.2.7 is_json

is_json(*value*, *schema=None*, *json_serializer=None*, ***kwargs*)
Indicate whether `value` is a valid JSON object.

Note: `schema` supports JSON Schema Drafts 3 - 7. Unless the JSON Schema indicates the meta-schema using a `$schema` property, the schema will be assumed to conform to Draft 7.

Parameters

- **value** – The value to evaluate.
- **schema** (dict / str / None) – An optional JSON schema against which value will be validated.

Returns True if value is valid, False if it is not.

Return type bool

2.2.8 is_string

is_string(*value*, *coerce_value=False*, *minimum_length=None*, *maximum_length=None*, *whitespace_padding=False*, ***kwargs*)

Indicate whether value is a string.

Parameters

- **value** – The value to evaluate.
- **coerce_value** (bool) – If True, will check whether value can be coerced to a string if it is not already. Defaults to False.
- **minimum_length** (int) – If supplied, indicates the minimum number of characters needed to be valid.
- **maximum_length** (int) – If supplied, indicates the minimum number of characters needed to be valid.
- **whitespace_padding** (bool) – If True and the value is below the minimum_length, pad the value with spaces. Defaults to False.

Returns True if value is valid, False if it is not.

Return type bool

2.2.9 is_iterable

is_iterable(*obj*, *forbid_literals=(<class 'str'>, <class 'bytes'>)*, *minimum_length=None*, *maximum_length=None*, ***kwargs*)

Indicate whether obj is iterable.

Parameters

- **forbid_literals** (*iterable*) – A collection of literals that will be considered invalid even if they are (actually) iterable. Defaults to a tuple containing str and bytes.
- **minimum_length** (int) – If supplied, indicates the minimum number of members needed to be valid.
- **maximum_length** (int) – If supplied, indicates the minimum number of members needed to be valid.

Returns True if obj is a valid iterable, False if not.

Return type bool

2.2.10 is_not_empty

is_not_empty(*value*, ***kwargs*)
Indicate whether `value` is empty.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is empty, `False` if it is not.

Return type `bool`

2.2.11 is_none

is_none(*value*, *allow_empty=False*, ***kwargs*)
Indicate whether `value` is `None`.

Parameters

- **value** – The value to evaluate.
- **allow_empty** (`bool`) – If `True`, accepts falsey values as equivalent to `None`. Defaults to `False`.

Returns `True` if `value` is `None`, `False` if it is not.

Return type `bool`

2.2.12 is_variable_name

is_variable_name(*value*, ***kwargs*)
Indicate whether `value` is a valid Python variable name.

> **Caution:** This function does **NOT** check whether the variable exists. It only checks that the `value` would work as a Python variable (or class, or function, etc.) name.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.2.13 is_callable

is_callable(*value*, ***kwargs*)
Indicate whether `value` is callable (like a function, method, or class).

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.2.14 is_uuid

is_uuid(*value*, ***kwargs*)
Indicate whether `value` contains a `UUID`

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.3 Date / Time

2.3.1 is_date

is_date(*value*, *minimum=None*, *maximum=None*, *coerce_value=False*, ***kwargs*)
Indicate whether `value` is a `date`.

Parameters

- **value** – The value to evaluate.
- **minimum** (`datetime` / `date` / compliant `str` / `None`) – If supplied, will make sure that `value` is on or after this value.
- **maximum** (`datetime` / `date` / compliant `str` / `None`) – If supplied, will make sure that `value` is on or before this value.
- **coerce_value** (`bool`) – If `True`, will return `True` if `value` can be coerced to a `date`. If `False`, will only return `True` if `value` is a date value only. Defaults to `False`.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.3.2 is_datetime

is_datetime(*value*, *minimum=None*, *maximum=None*, *coerce_value=False*, ***kwargs*)
Indicate whether `value` is a `datetime`.

Parameters

- **value** – The value to evaluate.
- **minimum** (`datetime` / `date` / compliant `str` / `None`) – If supplied, will make sure that `value` is on or after this value.
- **maximum** (`datetime` / `date` / compliant `str` / `None`) – If supplied, will make sure that `value` is on or before this value.
- **coerce_value** (`bool`) – If `True`, will return `True` if `value` can be coerced to a `datetime`. If `False`, will only return `True` if `value` is a complete timestamp. Defaults to `False`.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.3.3 is_time

is_time(*value*, *minimum=None*, *maximum=None*, *coerce_value=False*, ***kwargs*)
Indicate whether `value` is a `time`.

Parameters

- **value** – The value to evaluate.
- **minimum** (*datetime* or *time*-compliant `str` / `datetime` / `time`) – If supplied, will make sure that `value` is on or after this value.
- **maximum** (*datetime* or *time*-compliant `str` / `datetime` / `time`) – If supplied, will make sure that `value` is on or before this value.
- **coerce_value** (`bool`) – If `True`, will return `True` if `value` can be coerced to a `time`. If `False`, will only return `True` if `value` is a valid time. Defaults to `False`.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.3.4 is_timezone

is_timezone(*value*, *positive=True*, ***kwargs*)
Indicate whether `value` is a `tzinfo`.

> **Caution:** This does **not** validate whether the value is a timezone that actually exists, nor can it resolve timzone names (e.g. `'Eastern'` or `'CET'`).
>
> For that kind of functionality, we recommend you utilize: pytz

Parameters

- **value** – The value to evaluate.
- **positive** (`bool`) – Indicates whether the `value` is positive or negative (only has meaning if `value` is a string). Defaults to `True`.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.4 Numbers

2.4.1 is_numeric

is_numeric(*value*, *minimum=None*, *maximum=None*, ***kwargs*)
Indicate whether `value` is a numeric value.

Parameters

- **value** – The value to evaluate.

- **minimum** (*numeric*) – If supplied, will make sure that `value` is greater than or equal to this value.
- **maximum** (*numeric*) – If supplied, will make sure that `value` is less than or equal to this value.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.4.2 is_integer

is_integer(*value*, *coerce_value=False*, *minimum=None*, *maximum=None*, *base=10*, ***kwargs*)

Indicate whether `value` contains a whole number.

Parameters

- **value** – The value to evaluate.
- **coerce_value** (`bool`) – If `True`, will return `True` if `value` can be coerced to whole number. If `False`, will only return `True` if `value` is already a whole number (regardless of type). Defaults to `False`.
- **minimum** (*numeric*) – If supplied, will make sure that `value` is greater than or equal to this value.
- **maximum** (*numeric*) – If supplied, will make sure that `value` is less than or equal to this value.
- **base** (`int`) – Indicates the base that is used to determine the integer value. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with `0b/0B`, `0o/0O/0`, or `0x/0X`, as with integer literals in code. Base 0 means to interpret the string exactly as an integer literal, so that the actual base is 2, 8, 10, or 16. Defaults to `10`.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.4.3 is_float

is_float(*value*, *minimum=None*, *maximum=None*, ***kwargs*)

Indicate whether `value` is a `float`.

Parameters

- **value** – The value to evaluate.
- **minimum** (*numeric*) – If supplied, will make sure that `value` is greater than or equal to this value.
- **maximum** (*numeric*) – If supplied, will make sure that `value` is less than or equal to this value.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.4.4 is_fraction

is_fraction(*value*, *minimum=None*, *maximum=None*, ***kwargs*)
Indicate whether `value` is a `Fraction`.

Parameters

- **value** – The value to evaluate.
- **minimum** (*numeric*) – If supplied, will make sure that `value` is greater than or equal to this value.
- **maximum** (*numeric*) – If supplied, will make sure that `value` is less than or equal to this value.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.4.5 is_decimal

is_decimal(*value*, *minimum=None*, *maximum=None*, ***kwargs*)
Indicate whether `value` contains a `Decimal`.

Parameters

- **value** – The value to evaluate.
- **minimum** (*numeric*) – If supplied, will make sure that `value` is greater than or equal to this value.
- **maximum** (*numeric*) – If supplied, will make sure that `value` is less than or equal to this value.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.5 File-related

2.5.1 is_bytesIO

is_bytesIO(*value*, ***kwargs*)
Indicate whether `value` is a `BytesIO` object.

Note: This checker will return `True` even if `value` is empty, so long as its type is a `BytesIO`.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.5.2 is_stringIO

is_stringIO(*value*, ***kwargs*)
Indicate whether `value` is a `StringIO` object.

Note: This checker will return `True` even if `value` is empty, so long as its type is a `String`.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.5.3 is_pathlike

is_pathlike(*value*, ***kwargs*)
Indicate whether `value` is a path-like object.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.5.4 is_on_filesystem

is_on_filesystem(*value*, ***kwargs*)
Indicate whether `value` is a file or directory that exists on the local filesystem.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.5.5 is_file

is_file(*value*, ***kwargs*)
Indicate whether `value` is a file that exists on the local filesystem.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.5.6 is_directory

is_directory(*value*, ***kwargs*)
Indicate whether `value` is a directory that exists on the local filesystem.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.5.7 is_readable

is_readable(*value*, ***kwargs*)
Indicate whether `value` is a readable file.

> **Caution: Use of this validator is an anti-pattern and should be used with caution.**
>
> Validating the readability of a file *before* attempting to read it exposes your code to a bug called TOCTOU.
>
> This particular class of bug can expose your code to **security vulnerabilities** and so this validator should only be used if you are an advanced user.
>
> A better pattern to use when reading from a file is to apply the principle of EAFP ("easier to ask forgiveness than permission"), and simply attempt to write to the file using a `try ... except` block:
>
> ```
> try:
>     with open('path/to/filename.txt', mode = 'r') as file_object:
>         # read from file here
> except (OSError, IOError) as error:
>     # Handle an error if unable to write.
> ```

Parameters **value** (*Path-like object*) – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.5.8 is_writeable

is_writeable(*value*, ***kwargs*)
Indicate whether `value` is a writeable file.

> **Caution:** This validator does **NOT** work correctly on a Windows file system. This is due to the vagaries of how Windows manages its file system and the various ways in which it can manage file permission.
>
> If called on a Windows file system, this validator will raise `NotImplementedError()`.

> **Caution: Use of this validator is an anti-pattern and should be used with caution.**
>
> Validating the writability of a file *before* attempting to write to it exposes your code to a bug called TOCTOU.
>
> This particular class of bug can expose your code to **security vulnerabilities** and so this validator should only be used if you are an advanced user.
>
> A better pattern to use when writing to file is to apply the principle of EAFP ("easier to ask forgiveness than permission"), and simply attempt to write to the file using a `try ... except` block:
>
> ```
> try:
>     with open('path/to/filename.txt', mode = 'a') as file_object:
>         # write to file here
> except (OSError, IOError) as error:
>     # Handle an error if unable to write.
> ```

Note: This validator relies on `os.access()` to check whether `value` is writeable. This function has certain limitations, most especially that:

- It will **ignore** file-locking (yielding a false-positive) if the file is locked.
- It focuses on *local operating system permissions*, which means if trying to access a path over a network you might get a false positive or false negative (because network paths may have more complicated authentication methods).

Parameters **value** (*Path-like object*) – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

Raises **NotImplementedError** – if called on a Windows system

2.5.9 is_executable

is_executable(*value*, ***kwargs*)

Indicate whether `value` is an executable file.

Caution: This validator does **NOT** work correctly on a Windows file system. This is due to the vagaries of how Windows manages its file system and the various ways in which it can manage file permission.

If called on a Windows file system, this validator will raise `NotImplementedError()`.

Caution: Use of this validator is an anti-pattern and should be used with caution.

Validating the writability of a file *before* attempting to execute it exposes your code to a bug called TOCTOU.

This particular class of bug can expose your code to **security vulnerabilities** and so this validator should only be used if you are an advanced user.

A better pattern to use when writing to file is to apply the principle of EAFP ("easier to ask forgiveness than permission"), and simply attempt to execute the file using a `try ... except` block.

Note: This validator relies on `os.access()` to check whether `value` is writeable. This function has certain limitations, most especially that:

- It will **ignore** file-locking (yielding a false-positive) if the file is locked.
- It focuses on *local operating system permissions*, which means if trying to access a path over a network you might get a false positive or false negative (because network paths may have more complicated authentication methods).

Parameters **value** (*Path-like object*) – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

Raises **NotImplementedError** – if called on a Windows system

2.6 Internet-related

2.6.1 is_email

is_email(*value*, ***kwargs*)

Indicate whether `value` is an email address.

Note: Email address validation is…complicated. The methodology that we have adopted here is *generally* compliant with RFC 5322 and uses a combination of string parsing and regular expressions.

String parsing in particular is used to validate certain *highly unusual* but still valid email patterns, including the use of escaped text and comments within an email address' local address (the user name part).

This approach ensures more complete coverage for unusual edge cases, while still letting us use regular expressions that perform quickly.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.6.2 is_url

is_url(*value*, ***kwargs*)

Indicate whether `value` is a URL.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.6.3 is_domain

is_domain(*value*, ***kwargs*)

Indicate whether `value` is a valid domain.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.6.4 is_ip_address

is_ip_address(*value*, ***kwargs*)

Indicate whether `value` is a valid IP address (version 4 or version 6).

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.6.5 is_ipv4

is_ipv4(*value*, ***kwargs*)

Indicate whether `value` is a valid IP version 4 address.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.6.6 is_ipv6

is_ipv6(*value*, ***kwargs*)

Indicate whether `value` is a valid IP version 6 address.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

2.6.7 is_mac_address

is_mac_address(*value*, ***kwargs*)

Indicate whether `value` is a valid MAC address.

Parameters **value** – The value to evaluate.

Returns `True` if `value` is valid, `False` if it is not.

Return type `bool`

CHAPTER 3

Error Reference

- *Handling Errors*
  - *Validator Names/Types*
  - *Validator Messages*
  - *Stack Traces*
- *Standard Errors*
  - *EmptyValueError (from* `ValueError`*)*
  - *CannotCoerceError (from* `TypeError`*)*
  - *MinimumValueError (from* `ValueError`*)*
  - *MaximumValueError (from* `ValueError`*)*
  - *ValidatorUsageError (from* `ValueError`*)*
  - *CoercionFunctionEmptyError (from* `ValidatorUsageError`*)*
  - *CoercionFunctionError (from* `ValueError`*)*
- *Core*
  - *MinimumLengthError (from* `ValueError`*)*
  - *MaximumLengthError (from* `ValueError`*)*
  - *NotNoneError (from* `ValueError`*)*
  - *NotADictError (from* `ValueError`*)*
  - *NotJSONError (from* `ValueError`*)*
  - *NotJSONSchemaError (from* `ValueError`*)*
  - *JSONValidationError (from* `ValueError`*)*

    - *NotAnIterableError (from* `CannotCoerceError`*)*
    - *NotCallableError (from* `ValueError`*)*
    - *InvalidVariableNameError (from* `ValueError`*)*
- *Date / Time*
    - *UTCOffsetError (from* `ValueError`*)*
    - *NegativeOffsetMismatchError (from* `ValueError`*)*
    - *PositiveOffsetMismatchError (from* `ValueError`*)*
- *Numbers*
    - *NotAnIntegerError (from* `ValueError`*)*
- *File-related*
    - *NotPathlikeError (from* `ValueError`*)*
    - *PathExistsError (from* `IOError`*)*
    - *NotAFileError (from* `IOError`*)*
    - *NotADirectoryError (from* `IOError`*)*
    - *NotReadableError (from* `IOError`*)*
    - *NotWriteableError (from* `IOError`*)*
    - *NotExecutableError (from* `IOError`*)*
    - *NotBytesIOError (from* `ValueError`*)*
    - *NotStringIOError (from* `ValueError`*)*
- *Internet-related*
    - *InvalidEmailError (from* `ValueError`*)*
    - *InvalidURLError (from* `ValueError`*)*
    - *InvalidDomainError (from* `ValueError`*)*
    - *SlashInDomainError (from* `InvalidDomainError`*)*
    - *AtInDomainError (from* `InvalidDomainError`*)*
    - *ColonInDomainError (from* `InvalidDomainError`*)*
    - *WhitespaceInDomainError (from* `InvalidDomainError`*)*
    - *InvalidIPAddressError (from* `ValueError`*)*
    - *InvalidMACAddressError (from* `ValueError`*)*

3.1 Handling Errors

Tip: By design, *checkers* **never** raise exceptions. If a given value fails, a checker will just return `False`.

Validators **always** raise exceptions when validation fails.

When *validators* fail, they raise exceptions. There are three ways for exceptions to provide you with information that is useful in different circumstances:

1. **Exception Type**. The type of the exception itself (and the name of that type) tells you a lot about the nature of the error. On its own, this should be enough for you to understand "what went wrong" and "why validation failed". Most importantly, this is easy to catch in your code using `try ... except` blocks, giving you fine-grained control over how to handle exceptional situations.
2. **Message**. Each exception is raised when a human-readable message, a brief string that says "this is why this exception was raised". This is primarily useful in debugging your code, because at run-time we don't want to parse strings to make control flow decisions.
3. **Stack Trace**. Each exception is raised with a stacktrace of the exceptions and calls that preceded it. This helps to provide the context for the error, and is (typically) most useful for debugging and logging purposes. In rare circumstances, we might want to programmatically parse this information... but that's a pretty rare requirement.

We have designed the exceptions raised by the **Validator-Collection** to leverage all three of these types of information.

3.1.1 Validator Names/Types

By design, all exceptions raised by the **Validator-Collection** inherit from the built-in exceptions defined in the standard library. This makes it simple to plug the **Validator-Collection** into existing validation code you have which already catches `ValueError`, `TypeError`, and the like.

However, because we have sub-classed the built-in exceptions, you can easily apply more fine-grained control over your code.

For example, let us imagine a validation which will fail:

```
from validator_collection import validators

value = validators.decimal('123.45',
                           allow_empty = False,
                           minimum = 0,
                           maximum = 100)
```

By design, we know that this value will fail validation. We have specified a `maximum` of 100, and the value being passed in is (a string) with a value of `123.45`. This **will** fail.

We can catch this using a standard/built-in `ValueError` like so:

```
from validator_collection import validators

try:
    value = validators.decimal('123.45',
                               allow_empty = False,
                               minimum = 0,
                               maximum = 100)
except ValueError as error:
    # Handle the error
```

Looking at the documentation for *`validators.decimal()`*, we can see that this will catch all of the following situations:

- when an empty/false value is passed with `allow_empty = False`,
- when a value is less than the allowed minimum,
- when a value is more than the allowed maximum

But maybe we want to handle each of these situations a little differently? In that case, we can use the custom exceptions defined by the **Validator-Collection**:

```
from validator_collection import validators, errors

try:
    value = validators.decimal('123.45',
                               allow_empty = False,
                               minimum = 0,
                               maximum = 100)
except errors.EmptyValueError as error:
    # Handle the situation where an empty value was received.
except errors.MinimumValueError as error:
    # Handle the situation when a value is less than the allowed minimum.
except errors.MaximumValueError as error:
    # Handle the situation when a value is more than the allowed minimum.
```

Both approaches will work, but one gives you a little more precise control over how your code handles a failed validation.

Tip: We **strongly** recommend that you review the exceptions raised by each of the *Validator Reference* you use. Each validator precisely documents which exceptions it raises, and each exception's documentation shows what built-in exceptions it inherits from.

3.1.2 Validator Messages

Because the **Validator-Collection** produces exceptions which inherit from the standard library, we leverage the same API. This means they print to standard output with a human-readable message that provides an explanation for "what went wrong."

3.1.3 Stack Traces

Because the **Validator-Collection** produces exceptions which inherit from the standard library, it leverages the same API for handling stack trace information. This means that it will be handled just like a normal exception in unit test frameworks, logging solutions, and other tools that might need that information.

3.2 Standard Errors

3.2.1 EmptyValueError (from `ValueError`)

class EmptyValueError

Exception raised when an empty value is detected, but the validator does not allow for empty values.

Note: While in general, an "empty" value means a value that is falsey, for certain specific validators "empty" means explicitly `None`.

Please see: *Validator Reference*.

INHERITS FROM: `ValueError`

3.2.2 CannotCoerceError (from `TypeError`)

class CannotCoerceError
Exception raised when a value cannot be coerced to an expected type.

INHERITS FROM: `TypeError`

3.2.3 MinimumValueError (from `ValueError`)

class MinimumValueError
Exception raised when a value has a lower or earlier value than the minimum allowed.

INHERITS FROM: `ValueError`

3.2.4 MaximumValueError (from `ValueError`)

class MaximumValueError
Exception raised when a value exceeds a maximum allowed value.

INHERITS FROM: `ValueError`

3.2.5 ValidatorUsageError (from `ValueError`)

class ValidatorUsageError
Exception raised when the validator was used incorrectly.

INHERITS FROM: `ValueError`

3.2.6 CoercionFunctionEmptyError (from `ValidatorUsageError`)

class CoercionFunctionEmptyError
Exception raised when a coercion function was empty.

INHERITS FROM: `ValueError` -> *ValidatorUsageError*

3.2.7 CoercionFunctionError (from `ValueError`)

class CoercionFunctionError
Exception raised when a Coercion Function produces an `Exception`.

INHERITS FROM: `ValueError`

3.3 Core

3.3.1 MinimumLengthError (from `ValueError`)

class MinimumLengthError
Exception raised when a value has a lower length than the minimum allowed.

INHERITS FROM: `ValueError`

3.3.2 MaximumLengthError (from `ValueError`)

class MaximumLengthError
Exception raised when a value exceeds a maximum allowed length.

INHERITS FROM: `ValueError`

3.3.3 NotNoneError (from `ValueError`)

class NotNoneError
Exception raised when a value of `None` is expected, but a different empty value was detected.

INHERITS FROM: `ValueError`

3.3.4 NotADictError (from `ValueError`)

class NotADictError
Exception raised when a value is not a `dict`.

INHERITS FROM: `ValueError`

3.3.5 NotJSONError (from `ValueError`)

class NotJSONError
Exception raised when a value cannot be serialized/de-serialized to a JSON object.

INHERITS FROM: `ValueError`

3.3.6 NotJSONSchemaError (from `ValueError`)

class NotJSONSchemaError
Exception raised when a schema supplied is not a valid JSON Schema.

INHERITS FROM: `ValueError`

3.3.7 JSONValidationError (from `ValueError`)

class JSONValidationError
Exception raised when a value fails validation against a JSON Schema.

INHERITS FROM: `ValueError`

3.3.8 NotAnIterableError (from `CannotCoerceError`)

class NotAnIterableError
Exception raised when a value is not an iterable.

INHERITS FROM: `TypeError` -> *`CannotCoerceError`*

3.3.9 NotCallableError (from `ValueError`)

class NotCallableError
Exception raised when a given value is not callable.

INHERITS FROM: `ValueError`

3.3.10 InvalidVariableNameError (from `ValueError`)

class InvalidVariableNameError
Exception raised when a value is not a valid Python variable name.

INHERITS FROM: `ValueError`

3.4 Date / Time

3.4.1 UTCOffsetError (from `ValueError`)

class UTCOffsetError
Exception raised when the UTC offset exceeds +/- 24 hours.

INHERITS FROM: `ValueError`

3.4.2 NegativeOffsetMismatchError (from `ValueError`)

class NegativeOffsetMismatchError
Exception raised when a negative offset is expected, but the value indicates a positive offset.

INHERITS FROM: `ValueError`

3.4.3 PositiveOffsetMismatchError (from `ValueError`)

class PositiveOffsetMismatchError
Exception raised when a positive offset is expected, but the value indicates a negative offset.

INHERITS FROM: `ValueError`

3.5 Numbers

3.5.1 NotAnIntegerError (from `ValueError`)

class NotAnIntegerError
Exception raised when a value is not being coerced and is not an integer type.

INHERITS FROM: `ValueError`

3.6 File-related

3.6.1 NotPathlikeError (from `ValueError`)

class NotPathlikeError
Exception raised when a given value is not a path-like object.

INHERITS FROM: `ValueError`

3.6.2 PathExistsError (from `IOError`)

class PathExistsError
Exception raised when a path does not exist.

INHERITS FROM: `IOError`

3.6.3 NotAFileError (from `IOError`)

class NotAFileError
Exception raised when a path is not a file.

INHERITS FROM: `IOError`

3.6.4 NotADirectoryError (from `IOError`)

class NotADirectoryError
Exception raised when a path is not a directory.

INHERITS FROM: `IOError`

3.6.5 NotReadableError (from `IOError`)

class NotReadableError
Exception raised when a path is not readable.

INHERITS FROM: `IOError`

3.6.6 NotWriteableError (from `IOError`)

class NotWriteableError
Exception raised when a path is not writeable.

INHERITS FROM: `IOError`

3.6.7 NotExecutableError (from `IOError`)

class NotExecutableError
Exception raised when a path is not executable.

INHERITS FROM: `IOError`

3.6.8 NotBytesIOError (from `ValueError`)

class NotBytesIOError
Exception raised when a given value is not a `BytesIO` object.

INHERITS FROM: `ValueError`

3.6.9 NotStringIOError (from `ValueError`)

class NotStringIOError
Exception raised when a given value is not a `StringIO` object.

INHERITS FROM: `ValueError`

3.7 Internet-related

3.7.1 InvalidEmailError (from `ValueError`)

class InvalidEmailError
Exception raised when an email fails validation.

INHERITS FROM: `ValueError`

3.7.2 InvalidURLError (from `ValueError`)

class InvalidURLError
Exception raised when a URL fails validation.

INHERITS FROM: `ValueError`

3.7.3 InvalidDomainError (from `ValueError`)

class InvalidDomainError
Exception raised when a domain fails validation.

INHERITS FROM: `ValueError`

3.7.4 SlashInDomainError (from `InvalidDomainError`)

class SlashInDomainError
Exception raised when a domain value contains a slash or backslash.

INHERITS FROM: `ValueError` -> *InvalidDomainError*

3.7.5 AtInDomainError (from `InvalidDomainError`)

class AtInDomainError
Exception raised when a domain value contains an `@` symbol.

INHERITS FROM: `ValueError` -> *InvalidDomainError*

3.7.6 ColonInDomainError (from `InvalidDomainError`)

class ColonInDomainError
Exception raised when a domain value contains a colon (`:`).

INHERITS FROM: `ValueError` -> *`InvalidDomainError`*

3.7.7 WhitespaceInDomainError (from `InvalidDomainError`)

class WhitespaceInDomainError
Exception raised when a domain value contains whitespace.

INHERITS FROM: `ValueError` -> *`InvalidDomainError`*

3.7.8 InvalidIPAddressError (from `ValueError`)

class InvalidIPAddressError
Exception raised when a value is not a valid IP address.

INHERITS FROM: `ValueError`

3.7.9 InvalidMACAddressError (from `ValueError`)

class InvalidMACAddressError
Exception raised when a value is not a valid MAC address.

INHERITS FROM: `ValueError`

CHAPTER 4

Contributing to the Validator Collection

Note: As a general rule of thumb, the **Validator Collection** applies **PEP 8** styling, with some important differences.

Branch	Unit Tests
latest	
v. 1.3	
v. 1.2	
v. 1.1	
v. 1.0.0	
develop	

What makes an API idiomatic?

One of my favorite ways of thinking about idiomatic design comes from a talk given by Luciano Ramalho at Pycon 2016[5] where he listed traits of a Pythonic API as being:

- don't force [the user] to write boilerplate code
- provide ready to use functions and objects
- don't force [the user] to subclass unless there's a *very good* reason
- include the batteries: make easy tasks easy
- are simple to use but not simplistic: make hard tasks possible
- leverage the Python data model to:
  - provide objects that behave as you expect
  - avoid boilerplate through introspection (reflection) and metaprogramming.

[5] https://www.youtube.com/watch?v=k55d3ZUF3ZQ

Contents:

- *Design Philosophy*
- *Style Guide*
  - *Basic Conventions*
  - *Naming Conventions*
  - *Design Conventions*
  - *Documentation Conventions*
    - *Sphinx*
    - *Docstrings*
- *Dependencies*
- *Preparing Your Development Environment*
- *Ideas and Feature Requests*
- *Testing*
- *Submitting Pull Requests*
- *Building Documentation*
- *References*

4.1 Design Philosophy

The **Validator Collection** is meant to be a "beautiful" and "usable" library. That means that it should offer an idiomatic API that:

- works out of the box as intended,
- minimizes "bootstrapping" to produce meaningful output, and
- does not force users to understand how it does what it does.

In other words:

> Users should simply be able to drive the car without looking at the engine.

4.2 Style Guide

4.2.1 Basic Conventions

- Do not terminate lines with semicolons.
- Line length should have a maximum of *approximately* 90 characters. If in doubt, make a longer line or break the line between clear concepts.
- Each class should be contained in its own file.
- If a file runs longer than 2,000 lines. . . it should probably be refactored and split.

- All imports should occur at the top of the file.
- Do not use single-line conditions:

```
# GOOD
if x:
  do_something()

# BAD
if x: do_something()
```

- When testing if an object has a value, be sure to use `if x is None:` or `if x is not None`. Do **not** confuse this with `if x:` and `if not x:`.
- Use the `if x:` construction for testing truthiness, and `if not x:` for testing falsiness. This is **different** from testing:
  - `if x is True:`
  - `if x is False:`
  - `if x is None:`
- As of right now, because we feel that it negatively impacts readability and is less-widely used in the community, we are **not** using type annotations.

4.2.2 Naming Conventions

- `variable_name` and not `variableName` or `VariableName`. Should be a noun that describes what information is contained in the variable. If a `bool`, preface with `is_` or `has_` or similar question-word that can be answered with a yes-or-no.
- `function_name` and not `function_name` or `functionName`. Should be an imperative that describes what the function does (e.g. `get_next_page`).
- `CONSTANT_NAME` and not `constant_name` or `ConstantName`.
- `ClassName` and not `class_name` or `Class_Name`.

4.2.3 Design Conventions

- Functions at the module level can only be aware of objects either at a higher scope or singletons (which effectively have a higher scope).
- Functions and methods can use **one** positional argument (other than `self` or `cls`) without a default value. Any other arguments must be keyword arguments with default value given.

```
def do_some_function(argument):
  # rest of function...

def do_some_function(first_arg,
                     second_arg = None,
                     third_arg = True):
  # rest of function ...
```

- Functions and methods that accept values should start by validating their input, throwing exceptions as appropriate.
- When defining a class, define all attributes in `__init__`.

- When defining a class, start by defining its attributes and methods as private using a single-underscore prefix. Then, only once they're implemented, decide if they should be public.
- Don't be afraid of the private attribute/public property/public setter pattern:

```
class SomeClass(object):
  def __init__(*args, **kwargs):
    self._private_attribute = None

  @property
  def private_attribute(self):
    # custom logic which  may override the default return

    return self._private_attribute

  @setter.private_attribute
  def private_attribute(self, value):
    # custom logic that creates modified_value

    self._private_attribute = modified_value
```

- Separate a function or method's final (or default) `return` from the rest of the code with a blank line (except for single-line functions/methods).

4.2.4 Documentation Conventions

We are very big believers in documentation (maybe you can tell). To document the **Validator Collection** we rely on several tools:

Sphinx[1]

Sphinx[1] is used to organize the library's documentation into this lovely readable format (which will also be published to ReadTheDocs[2]). This documentation is written in reStructuredText[3] files which are stored in `<project>/docs`.

Tip: As a general rule of thumb, we try to apply the ReadTheDocs[2] own Documentation Style Guide[4] to our RST documentation.

Hint: To build the HTML documentation locally:

1. In a terminal, navigate to `<project>/docs`.
2. Execute `make html`.

When built locally, the HTML output of the documentation will be available at `./docs/_build/index.html`.

---

[1] http://sphinx-doc.org
[2] https://readthedocs.org
[3] http://www.sphinx-doc.org/en/stable/rest.html
[4] http://documentation-style-guide-sphinx.readthedocs.io/en/latest/style-guide.html

Docstrings

- Docstrings are used to document the actual source code itself. When writing docstrings we adhere to the conventions outlined in **PEP 257**.

4.3 Dependencies

Python 3.x

- jsonschema for JSON Schema validation

Python 2.x

- jsonschema for JSON Schema validation
- The regex drop-in replacement for Python's (buggy) standard `re` module.

> **Note:** This conditional dependency will be automatically installed if you are installing to Python 2.x.

4.4 Preparing Your Development Environment

In order to prepare your local development environment, you should:

1. Fork the Git repository.
2. Clone your forked repository.
3. Set up a virtual environment (optional).
4. Install dependencies:

```
validator-collection/ $ pip install -r requirements.txt
```

And you should be good to go!

4.5 Ideas and Feature Requests

Check for open issues or create a new issue to start a discussion around a bug or feature idea.

4.6 Testing

If you've added a new feature, we recommend you:

- create local unit tests to verify that your feature works as expected, and
- run local unit tests before you submit the pull request to make sure nothing else got broken by accident.

See also:

For more information about the **Validator Collection** testing approach please see: *Testing the Validator Collection*

4.7 Submitting Pull Requests

After you have made changes that you think are ready to be included in the main library, submit a pull request on Github and one of our developers will review your changes. If they're ready (meaning they're well documented, pass unit tests, etc.) then they'll be merged back into the main repository and slated for inclusion in the next release.

4.8 Building Documentation

In order to build documentation locally, you can do so from the command line using:

```
validator-collection/ $ cd docs
validator-collection/docs $ make html
```

When the build process has finished, the HTML documentation will be locally available at:

```
validator-collection/docs/_build/html/index.html
```

Note: Built documentation (the HTML) is **not** included in the project's Git repository. If you need local documentation, you'll need to build it.

4.9 References

CHAPTER 5

Testing the Validator Collection

Contents

- *Testing the Validator Collection*
  - *Testing Philosophy*
  - *Test Organization*
  - *Configuring & Running Tests*
    - *Installing with the Test Suite*
    - *Command-line Options*
    - *Configuration File*
    - *Running Tests*
  - *Skipping Tests*
  - *Incremental Tests*

5.1 Testing Philosophy

Note: Unit tests for the **Validator Collection** are written using pytest[1] and a comprehensive set of test automation are provided by tox[2].

There are many schools of thought when it comes to test design. When building the **Validator Collection**, we decided to focus on practicality. That means:

[1] https://docs.pytest.org/en/latest/
[2] https://tox.readthedocs.io

- **DRY is good, KISS is better.** To avoid repetition, our test suite makes extensive use of fixtures, parametrization, and decorator-driven behavior. This minimizes the number of test functions that are nearly-identical. However, there are certain elements of code that are repeated in almost all test functions, as doing so will make future readability and maintenance of the test suite easier.
- **Coverage matters...kind of.** We have documented the primary intended behavior of every function in the **Validator Collection** library, and the most-likely failure modes that can be expected. At the time of writing, we have about 85% code coverage. Yes, yes: We know that is less than 100%. But there are edge cases which are almost impossible to bring about, based on confluences of factors in the wide world. Our goal is to test the key functionality, and as bugs are uncovered to add to the test functions as necessary.

5.2 Test Organization

Each individual test module (e.g. `test_validators.py`) corresponds to a conceptual grouping of functionality. For example:

- `test_validators.py` tests validator functions found in `validator_collection/_validators.py`

Certain test modules are tightly coupled, as the behavior in one test module may have implications on the execution of tests in another. These test modules use a numbering convention to ensure that they are executed in their required order, so that `test_1_NAME.py` is always executed before `test_2_NAME.py`.

5.3 Configuring & Running Tests

5.3.1 Installing with the Test Suite

Installing via pip

```
$ pip install validator-collection[tests]
```

From Local Development Environment

See also:

When you *create a local development environment*, all dependencies for running and extending the test suite are installed.

5.3.2 Command-line Options

The **Validator Collection** does not use any custom command-line options in its test suite.

Tip: For a full list of the CLI options, including the defaults available, try:

```
validator-collection $ cd tests/
validator-collection/tests/ $ pytest --help
```

5.3.3 Configuration File

Because the **Validator Collection** has a very simple test suite, we have not prepared a `pytest.ini` configuration file.

5.3.4 Running Tests

Entire Test Suite

```
tests/ $ pytest
```

Test Module

```
tests/ $ pytest tests/test_module.py
```

Test Function

```
tests/ $ pytest tests/test_module.py -k 'test_my_test_function'
```

5.4 Skipping Tests

Note: Because of the simplicity of the **Validator Collection**, the test suite does not currently support any test skipping.

5.5 Incremental Tests

Note: The **Validator Collection** test suite does support incremental testing using, however at the moment none of the tests designed rely on this functionality.

A variety of test functions are designed to test related functionality. As a result, they are designed to execute incrementally. In order to execute tests incrementally, they need to be defined as methods within a class that you decorate with the `@pytest.mark.incremental` decorator as shown below:

```
@pytest.mark.incremental
class TestIncremental(object):
    def test_function1(self):
        pass
    def test_modification(self):
        assert 0
    def test_modification2(self):
        pass
```

This class will execute the `TestIncremental.test_function1()` test, execute and fail on the `TestIncremental.test_modification()` test, and automatically fail `TestIncremental.test_modification2()` because of the `.test_modification()` failure.

To pass state between incremental tests, add a `state` argument to their method definitions. For example:

```
@pytest.mark.incremental
class TestIncremental(object):
    def test_function(self, state):
        state.is_logged_in = True
        assert state.is_logged_in = True
    def test_modification1(self, state):
        assert state.is_logged_in is True
        state.is_logged_in = False
        assert state.is_logged_in is False
    def test_modification2(self, state):
        assert state.is_logged_in is True
```

Given the example above, the third test (`test_modification2`) will fail because `test_modification` updated the value of `state.is_logged_in`.

Note: `state` is instantiated at the level of the entire test session (one run of the test suite). As a result, it can be affected by tests in other test modules.

CHAPTER 6

Release History

Contents

- *Release History*
  - *Release 1.3.1 (released November 30, 2018)*
  - *Release 1.3.0 (released November 12, 2018)*
  - *Release 1.2.0 (released August 4, 2018)*
    - *Features Added*
    - *Bugs Fixed*
  - *Release 1.1.0 (released April 23, 2018)*
    - *Features Added*
    - *Bugs Fixed*
    - *Testing*
    - *Documentation*
  - *Release 1.0.0 (released April 16, 2018)*

6.1 Release 1.3.1 (released November 30, 2018)

- #21: Fixed *validators.datetime()* handling of timezone offsets to conform to ISO-8601.

6.2 Release 1.3.0 (released November 12, 2018)

- #18: Upgraded `requests` requirement to 2.20.1
- #17: Added `validators.json()` with support for JSON Schema validation.
- #17: Added `checkers.is_json()` with support for checking against JSON Schema.
- Added Python 3.7 to the Travis CI Test Matrix.

6.3 Release 1.2.0 (released August 4, 2018)

6.3.1 Features Added

- #14: Added `coerce_value` argument to `validators.date()`, `validators.datetime()`, and `validators.time()`.

6.3.2 Bugs Fixed

- #11: Removed legacy print statements.
- #13: `checkers.is_time()`, `checkers.is_date()`, and `checkers.is_datetime()` no longer return false positives

6.4 Release 1.1.0 (released April 23, 2018)

6.4.1 Features Added

- Added `validators.domain()` and `checkers.is_domain()` support with unit tests.
- #8: Added more verbose exceptions while retaining backwards-compatability with standard library exceptions.
- #6: Made it possible to disable validators by adding the validator name to the `VALIDATORS_DISABLED` environment variable.
- #6: Made it possible to disable checkers by adding the checker name to the `CHECKERS_DISABLED` environment variable.
- #6: Made it possible to force a validator or checker to run (even if disabled) by passing it a `force_run = True` keyword argument.
- #5: Added `validators.readable()` and `checkers.is_readable()` support to validate whether a file (path) is readable.
- #4: Added `validators.writeable()` and `checkers.is_writeable()` support to validate whether a file (path) is writeable. Only works on Linux, by design.
- #9: Added `validators.executable()` and `checkers.is_executable()` support to validate whether a file is executable. Only works on Linux, by design.

6.4.2 Bugs Fixed

- #7: Refactored `validators.email()` to more-comprehensively validate email addresses in compliance with RFC 5322.

6.4.3 Testing

- #6: Added unit tests for disabling validators and checkers based on the `VALIDATORS_DISABLED` and `CHECKERS_DISABLED` environment variables, with support for the `force_run = True` override.
- #7: Added more extensive email address cases to test compliance with RFC 5322.
- Added unit tests for `validators.domain()` and `checkers.is_domain()`.
- #5: Added unit tests for `validators.readable()` and `checkers.is_readable()` that work on the Linux platform. Missing unit tests on Windows.
- #4: Added unit tests for `validators.writeable()` and `checkers.is_writeable()`.
- #9: Added unit tests for `validators.executable()` and `checkers.is_executable()`.

6.4.4 Documentation

- Added `CHANGES.rst`.
- #7: Added additional detail to `validators.email()` documentation.
- #8: Added detailed exception / error handling documentation.
- #8: Updated validator error documentation.
- #6: Added documentation on disabling validators and checkers.
- #5: Added documentation for `validators.readable()` and `checkers.is_readable()`.
- #4: Added documentation for `validators.writeable()` and `checkers.is_writeable()`.
- #9: Added documentation for `validators.executable()` and `checkers.is_executable()`.

6.5 Release 1.0.0 (released April 16, 2018)

- First public release

CHAPTER 7

Glossary

Checker A function which takes an input value and indicates (`True`/`False`) whether it contains what you expect. Will always return a Boolean value, and will not raise an exception on failure.

Validator A function which takes an input value and ensures that it is what (the type or contents) you expect it to be. Will return the value or `None` depending on the arguments you pass to it, and will raise an exception if validation fails.

The **Validator Collection** is a Python library that provides more than 60 functions that can be used to validate the type and contents of an input value.

Each function has a consistent syntax for easy use, and has been tested on Python 2.7, 3.4, 3.5, 3.6, and 3.7.

For a list of validators available, please see the lists below.

Contents

- *Validator Collection*
  - *Installation*
    - *Dependencies*
  - *Available Validators and Checkers*
  - *Hello, World and Standard Usage*
    - *Using Validators*
    - *Using Checkers*
  - *Best Practices*
    - *Defensive Approach: Check, then Convert if Necessary*
    - *Confident Approach: try . . . except*
  - *Questions and Issues*

- *Contributing*
- *Testing*
- *License*
- *Indices and tables*

CHAPTER 8

Installation

To install the **Validator Collection**, just execute:

```
$ pip install validator-collection
```

8.1 Dependencies

Python 3.x

- jsonschema for JSON Schema validation

Python 2.x

- jsonschema for JSON Schema validation
- The regex drop-in replacement for Python's (buggy) standard `re` module.

> **Note:** This conditional dependency will be automatically installed if you are installing to Python 2.x.

CHAPTER 9

Available Validators and Checkers

Validators

Core	Date/Time	Numbers	File-related	Internet-related
dict	*date*	*numeric*	*bytesIO*	*email*
json	*datetime*	*integer*	*stringIO*	*url*
string	*time*	*float*	*path*	*domain*
iterable	*timezone*	*fraction*	*path_exists*	*ip_address*
none		*decimal*	*file_exists*	*ipv4*
not_empty			*directory_exists*	*ipv6*
uuid			*readable*	*mac_address*
variable_name			*writeable*	
			executable	

Checkers

Core	Date/Time	Numbers	File-related	Internet-related
is_type	*is_date*	*is_numeric*	*is_bytesIO*	*is_email*
is_between	*is_datetime*	*is_integer*	*is_stringIO*	*is_url*
has_length	*is_time*	*is_float*	*is_pathlike*	*is_domain*
are_equivalent	*is_timezone*	*is_fraction*	*is_on_filesystem*	*is_ip_address*
are_dicts_equivalent		*is_decimal*	*is_file*	*is_ipv4*
is_dict			*is_directory*	*is_ipv6*
is_json			*is_readable*	*is_mac_address*
is_string			*is_writeable*	
is_iterable			*is_executable*	
is_not_empty				
is_none				
is_callable				
is_uuid				
is_variable_name				

CHAPTER 10

Hello, World and Standard Usage

All validator functions have a consistent syntax so that using them is pretty much identical. Here's how it works:

```
from validator_collection import validators, checkers, errors

email_address = validators.email('test@domain.dev')
# The value of email_address will now be "test@domain.dev"

email_address = validators.email('this-is-an-invalid-email')
# Will raise a ValueError

try:
    email_address = validators.email(None)
    # Will raise an EmptyValueError
except errors.EmptyValueError:
    # Handling logic goes here
except errors.InvalidEmailError:
    # More handlign logic goes here

email_address = validators.email(None, allow_empty = True)
# The value of email_address will now be None

email_address = validators.email('', allow_empty = True)
# The value of email_address will now be None

is_email_address = checkers.is_email('test@domain.dev')
# The value of is_email_address will now be True

is_email_address = checkers.is_email('this-is-an-invalid-email')
# The value of is_email_address will now be False

is_email_address = checkers.is_email(None)
# The value of is_email_address will now be False
```

Pretty simple, right? Let's break it down just in case: Each validator comes in two flavors: a *validator* and a *checker*.

10.1 Using Validators

A validator does what it says on the tin: It validates that an input value is what you think it should be, and returns its valid form.

Each validator is expressed as the name of the thing being validated, for example *email()*.

Each validator accepts a value as its first argument, and an optional `allow_empty` boolean as its second argument. For example:

```
email_address = validators.email(value, allow_empty = True)
```

If the value you're validating validates successfully, it will be returned. If the value you're validating needs to be coerced to a different type, the validator will try to do that. So for example:

```
validators.integer(1)
validators.integer('1')
```

will both return an `int` of `1`.

If the value you're validating is empty/falsey and `allow_empty` is `False`, then the validator will raise a *EmptyValueError* exception (which inherits from the built-in `ValueError`). If `allow_empty` is `True`, then an empty/falsey input value will be converted to a `None` value.

Caution: By default, `allow_empty` is always set to `False`.

Hint: Some validators (particularly numeric ones like *integer*) have additional options which are used to make sure the value meets criteria that you set for it. These options are always included as keyword arguments *after* the `allow_empty` argument, and are documented for each validator below.

10.1.1 When Validation Fails

Validators raise exceptions when validation fails. All exceptions raised inherit from built-in exceptions like `ValueError`, `TypeError`, and `IOError`.

If the value you're validating fails its validation for some reason, the validator may raise different exceptions depending on the reason. In most cases, this will be a descendent of `ValueError` though it can sometimes be a `TypeError`, or an `IOError`, etc.

For specifics on each validator's likely exceptions and what can cause them, please review the *Validator Reference*.

Hint: While validators will always raise built-in exceptions from the standard library, to give you greater programmatic control over how to respond when validation fails, we have defined a set of custom exceptions that inherit from those built-ins.

Our custom exceptions provide you with very specific, fine-grained information as to *why* validation for a given value failed. In general, most validators will raise `ValueError` or `TypeError` exceptions, and you can safely catch those and be fine. But if you want to handle specific types of situations with greater control, then you can instead catch *EmptyValueError*, *CannotCoerceError*, *MaximumValueError*, and the like.

For more detailed information, please see: *Error Reference* and *Validator Reference*.

10.1.2 Disabling Validation

Caution: If you are disabling validators using the `VALIDATORS_DISABLED` environment variable, their related *checkers* will **also** be disabled (meaning they will always return `True`).

Validation can at times be an expensive (in terms of performance) operation. As a result, there are times when you want to disable certain kinds of validation when running in production. Using the **Validator-Collection** this is simple:

> Just add the name of the validator you want disabled to the `VALIDATORS_DISABLED` environment variable, and validation will automatically be skipped.

Caution: `VALIDATORS_DISABLED` expects a comma-separated list of values. If it isn't comma-separated, it won't work properly.

Here's how it works in practice. Let's say we define the following environment variable:

```
$ export VALIDATORS_DISABLED = "variable_name, email, ipv4"
```

This disables the *variable_name()*, *email()*, and *ipv4()* validators respectively.

Now if we run:

```
from validator_collection import validators, errors

try:
    result = validators.variable_name('this is an invalid variable name')
except ValueError:
    # handle the error
```

The validator will return the `value` supplied to it un-changed. So that means `result` will be equal to `this is an invalid variable name`.

However, if we run:

```
from validator_collection import validators, errors

try:
    result = validators.integer('this is an invalid variable name')
except errors.NotAnIntegerError:
    # handle the error
```

the validator will run and raise *NotAnIntegerError*.

We can force validators to run (even if disabled using the environment variable) by passing a `force_run = True` keyword argument. For example:

```
from validator_collection import validators, errors

try:
    result = validators.variable_name('this is an invalid variable name',
                                      force_run = True)
except ValueError:
    # handle the error
```

will produce a *InvalidVariableNameError* (which is a type of `ValueError`).

10.2 Using Checkers

A *checker* is what it sounds like: It checks that an input value is what you expect it to be, and tells you `True`/`False` whether it is or not.

Important: Checkers do *not* verify or convert object types. You can think of a checker as a tool that tells you whether its corresponding *validator* would fail. See *Best Practices* for tips and tricks on using the two together.

Each checker is expressed as the name of the thing being validated, prefixed by `is_`. So the checker for an email address is *`is_email()`* and the checker for an integer is *`is_integer()`*.

Checkers take the input value you want to check as their first (and often only) positional argumet. If the input value validates, they will return `True`. Unlike *validators*, checkers will not raise an exception if validation fails. They will instead return `False`.

Hint: If you need to know *why* a given value failed to validate, use the validator instead.

Hint: Some checkers (particularly numeric ones like *`is_integer`*) have additional options which are used to make sure the value meets criteria that you set for it. These options are always *optional* and are included as keyword arguments *after* the input value argument. For details, please see the *Checker Reference*.

10.2.1 Disabling Checking

Caution: If you are disabling validators using the `VALIDATORS_DISABLED` environment variable, their related checkers will **also** be disabled. This means they will always return `True` unless you call them using `force_run = True`.

Checking can at times be an expensive (in terms of performance) operation. As a result, there are times when you want to disable certain kinds of checking when running in production. Using the **Validator-Collection** this is simple:

> Just add the name of the checker you want disabled to the `CHECKERS_DISABLED` environment variable, and validation will automatically be skipped.

Caution: `CHECKERS_DISABLED` expects a comma-separated list of values. If it isn't comma-separated, it won't work properly.

Here's how it works in practice. Let's say we define the following environment variable:

```
$ export CHECKERS_DISABLED = "is_variable_name, is_email, is_ipv4"
```

This disables the *`is_variable_name()`*, *`is_email()`*, and *`is_ipv4()`* validators respectively.

Now if we run:

```
from validator_collection import checkers, errors
```

(continues on next page)

(continued from previous page)

```
result = checkers.is_variable_name('this is an invalid variable name')
# result will be True
```

The checker will return `True`.

However, if we run:

```
from validator_collection import checkers

result = validators.is_integer('this is an invalid variable name')
# result will be False
```

the checker will return `False`

We can force checkers to run (even if disabled using the environment variable) by passing a `force_run = True` keyword argument. For example:

```
from validator_collection import checkers, errors

result = checkers.is_variable_name('this is an invalid variable name',
                                   force_run = True)
# result will be False
```

will return `False`.

CHAPTER 11

Best Practices

Checkers and *Validators* are designed to be used together. You can think of them as a way to quickly and easily verify that a value contains the information you expect, and then make sure that value is in the form your code needs it in.

There are two fundamental patterns that we find work well in practice.

11.1 Defensive Approach: Check, then Convert if Necessary

We find this pattern is best used when we don't have any certainty over a given value might contain. It's fundamentally defensive in nature, and applies the following logic:

1. Check whether `value` contains the information we need it to or can be converted to the form we need it in.
2. If `value` does not contain what we need but *can* be converted to what we need, do the conversion.
3. If `value` does not contain what we need but *cannot* be converted to what we need, raise an error (or handle it however it needs to be handled).

We tend to use this where we're first receiving data from outside of our control, so when we get data from a user, from the internet, from a third-party API, etc.

Here's a quick example of how that might look in code:

```
from validator_collection import checkers, validators

def some_function(value):
    # Check whether value contains a whole number.
    is_valid = checkers.is_integer(value,
                                   coerce_value = False)

    # If the value does not contain a whole number, maybe it contains a
    # numeric value that can be rounded up to a whole number.
    if not is_valid and checkers.is_integer(value, coerce_value = True):
        # If the value can be rounded up to a whole number, then do so:
        value = validators.integer(value, coerce_value = True)
```

(continues on next page)

(continued from previous page)

```
    elif not is_valid:
        # Since the value does not contain a whole number and cannot be converted to
        # one, this is where your code to handle that error goes.
        raise ValueError('something went wrong!')

    return value

value = some_function(3.14)
# value will now be 4

new_value = some_function('not-a-number')
# will raise ValueError
```

Let's break down what this code does. First, we define `some_function()` which takes a value. This function uses the *`is_integer()`* checker to see if `value` contains a whole number, regardless of its type.

If it doesn't contain a whole number, maybe it contains a numeric value that can be rounded up to a whole number? It again uses the *`is_integer()`* to check if that's possible. If it is, then it calls the *`integer()`* validator to coerce `value` to a whole number.

If it can't coerce `value` to a whole number? It raises a `ValueError`.

11.2 Confident Approach: try ... except

Sometimes, we'll have more confidence in the values that we can expect to work with. This means that we might expect `value` to *generally* have the kind of data we need to work with. This means that situations where `value` doesn't contain what we need will truly be exceptional situations, and can be handled accordingly.

In this situation, a good approach is to apply the following logic:

1. Skip a *checker* entirely, and just wrap the validator in a `try...except` block.

We tend to use this in situations where we're working with data that our own code has produced (meaning we know - generally - what we can expect, unless something went seriously wrong).

Here's an example:

```
from validator_collection import validators, errors

def some_function(value):
    try:
        email_address = validators.email(value, allow_empty = False)
    except errors.InvalidEmailError as error:
        # handle the error here
    except ValueError as error:
        # handle other ValueErrors here

    # do something with your new email address value

    return email_address

email = some_function('email@domain.com')
# This will return the email address.

email = some_function('not-a-valid-email')
# This will raise a ValueError that some_function() will handle.
```

(continues on next page)

(continued from previous page)

```
email = some_function(None)
# This will raise a ValueError that some_function() will handle.
```

So what's this code do? It's pretty straightforward. `some_function()` expects to receive a `value` that contains an email address. We expect that `value` will *typically* be an email address, and not something weird (like a number or something). So we just try the validator - and if validation fails, we handle the error appropriately.

CHAPTER 12

Questions and Issues

You can ask questions and report issues on the project's Github Issues Page

CHAPTER 13

Contributing

We welcome contributions and pull requests! For more information, please see the *Contributor Guide*

CHAPTER 14

Testing

We use TravisCI for our build automation and ReadTheDocs for our documentation.

Detailed information about our test suite and how to run tests locally can be found in our *Testing Reference*.

CHAPTER 15

License

The **Validator Collection** is made available on a **MIT License**.

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

tests, 57

v

validator_collection.checkers, 27
validator_collection.errors, 41
validator_collection.validators, 5

Index

A

are_dicts_equivalent() (*in module validator_collection.checkers*), 28
are_equivalent() (*in module validator_collection.checkers*), 27
AtInDomainError (*class in validator_collection.errors*), 49

B

bytesIO() (*in module validator_collection.validators*), 14

C

CannotCoerceError (*class in validator_collection.errors*), 45
Checker, **65**
CoercionFunctionEmptyError (*class in validator_collection.errors*), 45
CoercionFunctionError (*class in validator_collection.errors*), 45
ColonInDomainError (*class in validator_collection.errors*), 50

D

date() (*in module validator_collection.validators*), 9
datetime() (*in module validator_collection.validators*), 10
decimal() (*in module validator_collection.validators*), 14
dict() (*in module validator_collection.validators*), 5
directory_exists() (*in module validator_collection.validators*), 16
domain() (*in module validator_collection.validators*), 20

E

email() (*in module validator_collection.validators*), 19
EmptyValueError (*class in validator_collection.errors*), 44
executable() (*in module validator_collection.validators*), 18

F

file_exists() (*in module validator_collection.validators*), 16
float() (*in module validator_collection.validators*), 13
fraction() (*in module validator_collection.validators*), 13

H

has_length() (*in module validator_collection.checkers*), 29

I

integer() (*in module validator_collection.validators*), 12
InvalidDomainError (*class in validator_collection.errors*), 49
InvalidEmailError (*class in validator_collection.errors*), 49
InvalidIPAddressError (*class in validator_collection.errors*), 50
InvalidMACAddressError (*class in validator_collection.errors*), 50
InvalidURLError (*class in validator_collection.errors*), 49
InvalidVariableNameError (*class in validator_collection.errors*), 47
ip_address() (*in module validator_collection.validators*), 21
ipv4() (*in module validator_collection.validators*), 22
ipv6() (*in module validator_collection.validators*), 22
is_between() (*in module validator_collection.checkers*), 28
is_bytesIO() (*in module validator_collection.checkers*), 35

is_callable() (*in module validator_collection.checkers*), 31
is_date() (*in module validator_collection.checkers*), 32
is_datetime() (*in module validator_collection.checkers*), 32
is_decimal() (*in module validator_collection.checkers*), 35
is_dict() (*in module validator_collection.checkers*), 29
is_directory() (*in module validator_collection.checkers*), 36
is_domain() (*in module validator_collection.checkers*), 39
is_email() (*in module validator_collection.checkers*), 39
is_executable() (*in module validator_collection.checkers*), 38
is_file() (*in module validator_collection.checkers*), 36
is_float() (*in module validator_collection.checkers*), 34
is_fraction() (*in module validator_collection.checkers*), 35
is_integer() (*in module validator_collection.checkers*), 34
is_ip_address() (*in module validator_collection.checkers*), 39
is_ipv4() (*in module validator_collection.checkers*), 40
is_ipv6() (*in module validator_collection.checkers*), 40
is_iterable() (*in module validator_collection.checkers*), 30
is_json() (*in module validator_collection.checkers*), 29
is_mac_address() (*in module validator_collection.checkers*), 40
is_none() (*in module validator_collection.checkers*), 31
is_not_empty() (*in module validator_collection.checkers*), 31
is_numeric() (*in module validator_collection.checkers*), 33
is_on_filesystem() (*in module validator_collection.checkers*), 36
is_pathlike() (*in module validator_collection.checkers*), 36
is_readable() (*in module validator_collection.checkers*), 37
is_string() (*in module validator_collection.checkers*), 30
is_stringIO() (*in module validator_collection.checkers*), 36
is_time() (*in module validator_collection.checkers*), 33
is_timezone() (*in module validator_collection.checkers*), 33
is_type() (*in module validator_collection.checkers*), 27
is_url() (*in module validator_collection.checkers*), 39
is_uuid() (*in module validator_collection.checkers*), 32
is_variable_name() (*in module validator_collection.checkers*), 31
is_writeable() (*in module validator_collection.checkers*), 37
iterable() (*in module validator_collection.validators*), 7

J

json() (*in module validator_collection.validators*), 6
JSONValidationError (*class in validator_collection.errors*), 46

M

mac_address() (*in module validator_collection.validators*), 22
MaximumLengthError (*class in validator_collection.errors*), 46
MaximumValueError (*class in validator_collection.errors*), 45
MinimumLengthError (*class in validator_collection.errors*), 45
MinimumValueError (*class in validator_collection.errors*), 45

N

NegativeOffsetMismatchError (*class in validator_collection.errors*), 47
none() (*in module validator_collection.validators*), 8
not_empty() (*in module validator_collection.validators*), 8
NotADictError (*class in validator_collection.errors*), 46
NotADirectoryError (*class in validator_collection.errors*), 48
NotAFileError (*class in validator_collection.errors*), 48
NotAnIntegerError (*class in validator_collection.errors*), 47
NotAnIterableError (*class in validator_collection.errors*), 46
NotBytesIOError (*class in validator_collection.errors*), 49
NotCallableError (*class in validator_collection.errors*), 47

NotExecutableError (*class in validator_collection.errors*), 48
NotJSONError (*class in validator_collection.errors*), 46
NotJSONSchemaError (*class in validator_collection.errors*), 46
NotNoneError (*class in validator_collection.errors*), 46
NotPathlikeError (*class in validator_collection.errors*), 48
NotReadableError (*class in validator_collection.errors*), 48
NotStringIOError (*class in validator_collection.errors*), 49
NotWriteableError (*class in validator_collection.errors*), 48
numeric() (*in module validator_collection.validators*), 12

P

path() (*in module validator_collection.validators*), 15
path_exists() (*in module validator_collection.validators*), 15
PathExistsError (*class in validator_collection.errors*), 48
PositiveOffsetMismatchError (*class in validator_collection.errors*), 47
Python Enhancement Proposals
    PEP 257, 55
    PEP 8, 51

R

readable() (*in module validator_collection.validators*), 17

S

SlashInDomainError (*class in validator_collection.errors*), 49
string() (*in module validator_collection.validators*), 7
stringIO() (*in module validator_collection.validators*), 15

T

tests (*module*), 57
time() (*in module validator_collection.validators*), 10
timezone() (*in module validator_collection.validators*), 11

U

url() (*in module validator_collection.validators*), 20
UTCOffsetError (*class in validator_collection.errors*), 47
uuid() (*in module validator_collection.validators*), 8

V

Validator, **65**
validator_collection.checkers (*module*), 27
validator_collection.errors (*module*), 41
validator_collection.validators (*module*), 5
ValidatorUsageError (*class in validator_collection.errors*), 45
variable_name() (*in module validator_collection.validators*), 9

W

WhitespaceInDomainError (*class in validator_collection.errors*), 50
writeable() (*in module validator_collection.validators*), 17