Validator Collection Documentation

Release 1.0.0.1

Insight Industry Inc.

Apr 16, 2018

Contents:

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. 1.0.0	
develop	

CHAPTER 1

Validator Reference

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

1.1 Core

1.1.1 dict

dict(*value*, *allow_empty=False*, *json_serializer=None*)

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 `ValueError` 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

- **ValueError** – if `value` is empty and `allow_empty` is `False`
- **ValueError** – if `value` is not a `dict`

1.1.2 string

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

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 `ValueError` 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

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

1.1.3 iterable

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

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 ValueError 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

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

1.1.4 none

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

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 ValueError if value is empty but **not** None. Defaults to False.

Returns None

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

1.1.5 not_empty

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

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 ValueError if value is empty. Defaults to False.

Returns value / None

Raises **ValueError** – if value is empty and allow_empty is False

1.1.6 uuid

uuid(*value*, *allow_empty=False*)
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 `ValueError` if `value` is empty. Defaults to `False`.

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

Return type `UUID` / `None`

Raises

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

1.1.7 variable_name

variable_name(*value*, *allow_empty=False*)
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 `ValueError` if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Return type `str` or `None`

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

1.2 Date / Time

1.2.1 date

date(*value*, *allow_empty=False*, *minimum=None*, *maximum=None*)
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 `ValueError` 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.

Returns value / None

Return type date / None

Raises

- **ValueError** – if value is empty and allow_empty is False
- **ValueError** – if value is not a valid value type or None
- **ValueError** – if minimum is supplied but value occurs before minimum
- **ValueError** – if maximum is supplied but value occurs after minimum

1.2.2 datetime

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

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 ValueError 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.

Returns value / None

Return type datetime / None

Raises

- **ValueError** – if value is empty and allow_empty is False
- **ValueError** – if minimum is supplied but value occurs before minimum
- **ValueError** – if maximum is supplied but value occurs after minimum

1.2.3 time

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

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 `ValueError` 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.

Returns `value` in UTC time / `None`

Return type `time` / `None`

Raises

- **ValueError** – if `value` is empty and `allow_empty` is `False`
- **ValueError** – if `value` is not a valid value type or `None`
- **ValueError** – if `minimum` is supplied but `value` occurs before `minimum`
- **ValueError** – if `maximum` is supplied but `value` occurs after `minimum`

1.2.4 timezone

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

Validate that `value` is a valid `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** (`str` / `tzinfo` / numeric / `None`) – The value to validate.
- **allow_empty** (`bool`) – If `True`, returns `None` if `value` is empty. If `False`, raises a `ValueError` 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

- **ValueError** – if `value` is empty and `allow_empty` is `False`
- **ValueError** – if `value` is not a valid value type or `None`

1.3 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`.

1.3.1 numeric

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

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 a `ValueError` 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

- **ValueError** – if `value` is `None` and `allow_empty` is `False`
- **ValueError** – if `minimum` is supplied and `value` is less than the `minimum`
- **ValueError** – if `maximum` is supplied and `value` is more than the `maximum`

1.3.2 integer

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

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 `ValueError` 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

- **ValueError** – if `value` is `None` and `allow_empty` is `False`
- **ValueError** – if `minimum` is supplied and `value` is less than the `minimum`
- **ValueError** – if `maximum` is supplied and `value` is more than the `maximum`

1.3.3 float

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

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 `ValueError` if `value` is `None`. Defaults to `False`.

Returns `value` / `None`

Return type `float` / `None`

Raises

- **ValueError** – if `value` is `None` and `allow_empty` is `False`
- **ValueError** – if `minimum` is supplied and `value` is less than the `minimum`
- **ValueError** – if `maximum` is supplied and `value` is more than the `maximum`

1.3.4 fraction

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

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 `ValueError` if `value` is `None`. Defaults to `False`.

Returns `value` / `None`

Return type `Fraction` / `None`

Raises

- **ValueError** – if `value` is `None` and `allow_empty` is `False`
- **ValueError** – if `minimum` is supplied and `value` is less than the `minimum`
- **ValueError** – if `maximum` is supplied and `value` is more than the `maximum`

1.3.5 decimal

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

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 `ValueError` 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

- **ValueError** – if `value` is `None` and `allow_empty` is `False`
- **ValueError** – if `minimum` is supplied and `value` is less than the `minimum`
- **ValueError** – if `maximum` is supplied and `value` is more than the `maximum`

1.4 File-related

1.4.1 bytesIO

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

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 `ValueError` if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Return type `StringIO` / `None`

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

1.4.2 stringIO

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

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 `ValueError` if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Return type `StringIO` / `None`

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

1.4.3 path

`path`(*value*, *allow_empty=False*)

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 `ValueError` if `value` is empty. Defaults to `False`.

Returns The path represented by `value`.

Return type Path-like object / `None`

Raises

- **`ValueError`** – if `allow_empty` is `False` and `value` is empty
- **`ValueError`** – if `value` is not a valid path

1.4.4 path_exists

`path_exists`(*value*, *allow_empty=False*)

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 `ValueError` if `value` is empty. Defaults to `False`.

Returns The file name represented by `value`.

Return type Path-like object / `None`

Raises

- **`ValueError`** – if `allow_empty` is `False` and `value` is empty
- **`IOError`** – if `value` does not exist

1.4.5 file_exists

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

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 `ValueError` if `value` is empty. Defaults to `False`.

Returns The file name represented by `value`.

Return type Path-like object / `None`

Raises

- **`ValueError`** – if `allow_empty` is `False` and `value` is empty
- **`IOError`** – if `value` does not exist on the local filesystem
- **`ValueError`** – if `value` is not a valid file

1.4.6 directory_exists

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

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 `ValueError` if `value` is empty. Defaults to `False`.

Returns The file name represented by `value`.

Return type Path-like object / `None`

Raises

- **`ValueError`** – if `allow_empty` is `False` and `value` is empty
- **`IOError`** – if `value` does not exist on the local filesystem
- **`ValueError`** – if `value` is not a valid directory

1.5 Internet-related

1.5.1 email

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

Validate that `value` is a valid email address.

Parameters

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

Returns `value` / `None`

Return type `str` / `None`

Raises

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

1.5.2 url

url(*value*, *allow_empty=False*)
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 `ValueError` if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Return type `str` / `None`

Raises

- **ValueError** – if `value` is empty and `allow_empty` is `False`
- **ValueError** – if `value` is not a valid URL or empty with `allow_empty` set to `True`

1.5.3 ip_address

ip_address(*value*, *allow_empty=False*)
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 `ValueError` if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Raises

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

1.5.4 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 `ValueError` if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Raises

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

1.5.5 ipv6

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

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 `ValueError` if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Raises

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

1.5.6 mac_address

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

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 `ValueError` if `value` is empty. Defaults to `False`.

Returns `value` / `None`

Return type `str` / `None`

Raises

- **ValueError** – if `value` is empty and `allow_empty` is `False`
- **ValueError** – 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_ip_address*
are_equivalent	*is_timezone*	*is_fraction*	*is_on_filesystem*	*is_ipv4*
are_dicts_equivalent		*is_decimal*	*is_file*	*is_ipv6*
is_dict			*is_directory*	*is_mac_address*
is_string				
is_iterable				
is_not_empty				
is_none				
is_callable				
is_uuid				
is_variable_name				

2.1 Core

2.1.1 is_type

is_type(*obj*, *type_*)

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

2.1.2 are_equivalent

are_equivalent(**args*)

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

2.1.3 are_dicts_equivalent

are_dicts_equivalent(**args*)

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

2.1.4 is_between

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

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`

2.1.5 has_length

has_length(*value*, *minimum=None*, *maximum=None*)

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`

2.1.6 is_dict

`is_dict`(*value*)

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`

2.1.7 is_string

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

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`

2.1.8 is_iterable

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

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`

2.1.9 is_not_empty

is_not_empty(*value*)

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`

2.1.10 is_none

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

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`

2.1.11 is_variable_name

is_variable_name(*value*)

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`

2.1.12 is_callable

is_callable(*value*)
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`

2.1.13 is_uuid

is_uuid(*value*)
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`

2.2 Date / Time

2.2.1 is_date

is_date(*value*, *minimum=None*, *maximum=None*)
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.

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

Return type `bool`

2.2.2 is_datetime

is_datetime(*value*, *minimum=None*, *maximum=None*)
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.

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

Return type bool

2.2.3 is_time

is_time(*value*, *minimum=None*, *maximum=None*)
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.

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

Return type bool

2.2.4 is_timezone

is_timezone(*value*, *positive=True*)
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

2.3 Numbers

2.3.1 is_numeric

is_numeric(*value*, *minimum=None*, *maximum=None*)
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`

2.3.2 is_integer

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

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`

2.3.3 is_float

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

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`

2.3.4 is_fraction

is_fraction(*value*, *minimum=None*, *maximum=None*)

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`

2.3.5 is_decimal

is_decimal(*value*, *minimum=None*, *maximum=None*)

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`

2.4 File-related

2.4.1 is_bytesIO

is_bytesIO(*value*)

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`

2.4.2 is_stringIO

is_stringIO(*value*)

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`

2.4.3 is_pathlike

is_pathlike(*value*)
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`

2.4.4 is_on_filesystem

is_on_filesystem(*value*)
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`

2.4.5 is_file

is_file(*value*)
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`

2.4.6 is_directory

is_directory(*value*)
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 Internet-related

2.5.1 is_email

is_email(*value*)
Indicate whether `value` is an email address.

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

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

Return type `bool`

2.5.2 is_url

is_url(*value*)

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`

2.5.3 is_ip_address

is_ip_address(*value*)

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`

2.5.4 is_ipv4

is_ipv4(*value*)

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`

2.5.5 is_ipv6

is_ipv6(*value*)

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`

2.5.6 is_mac_address

is_mac_address(*value*)

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

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. 1.0.0	
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:

3.1 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.

3.2 Style Guide

3.2.1 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.

3.2.2 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`.

3.2.3 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).

3.2.4 Documentation Conventions

We are very big believers in documentation (maybe you can tell). To document the **PDF Layer Extractor** 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**.

3.3 Dependencies

Python 3.x

Python 2.x

None. Uses the standard library.

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.

3.4 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!

3.5 Ideas and Feature Requests

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

3.6 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*

3.7 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.

3.8 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.

3.9 References

CHAPTER 4

Testing the Validator Collection

Contents

4.1 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.

4.2 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`.

4.3 Configuring & Running Tests

4.3.1 Installing with the Test Suite

Installing via pip

From Local Development Environment

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

See also:

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

4.3.2 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
```

4.3.3 Configuration File

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

4.3.4 Running Tests

Entire Test Suite

Test Module

Test Function

```
tests/ $ pytest
```

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

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

4.4 Skipping Tests

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

4.5 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 5

Glossary

Checker A function which takes an input value and indicates (`True`/`False`) whether it is what you expect it to be. Will always return a boolean.

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.

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, and 3.6.

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

Contents

CHAPTER 6

Installation

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

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

6.1 Dependencies

Python 3.x

Python 2.x

None. Uses the standard library.

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 7

Available Validators and Checkers

Validators

Checkers

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

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_ip_address*
are_equivalent	*is_timezone*	*is_fraction*	*is_on_filesystem*	*is_ipv4*
are_dicts_equivalent		*is_decimal*	*is_file*	*is_ipv6*
is_dict			*is_directory*	*is_mac_address*
is_string				
is_iterable				
is_not_empty				
is_none				
is_callable				
is_uuid				
is_variable_name				

CHAPTER 8

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

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

email_address = validators.email(None)
# Will raise a ValueError

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*.

8.1 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 `ValueError` exception. 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`.

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 `ValueError` though it can sometimes be a `TypeError`, or an `AttributeError`, etc. For specifics on each validator's likely exceptions and what can cause them, please review the *Validator Reference*.

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.

8.2 Using Checkers

Likewise, 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*.

CHAPTER 9

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.

9.1 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`.

9.2 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

def some_function(value):
    try:
      email_address = validators.email(value, allow_empty = False)
    except ValueError:
      # handle the error 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 10

Questions and Issues

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

CHAPTER 11

Contributing

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

CHAPTER 12

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 13

License

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

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

v

Index

M

N

P

S

T

U

V