validator.py Documentation

Release 0.4.0

Samuel Lucidi

July 07, 2016

Contents

CHAPTER 1

About

`validator.py` is a tool for ensuring that data conforms to certain sets of rules, called validations. A validation is essentially a schema for a dictionary, containing a list of rules for each key/value pair in the dictionary you want to validate. This is intended to fill a similar use case to form validations in WTForms or Rails, but for general sources of data, not just web forms. To get right on with it, here's a quick example of what this is for and how it works:

```
from validator import Required, Not, Truthy, Blank, Range, Equals, In, validate

# let's say that my dictionary needs to meet the following rules...
rules = {
    "foo": [Required, Equals(123)], # foo must be exactly equal to 123
    "bar": [Required, Truthy()],    # bar must be equivalent to True
    "baz": [In(["spam", "eggs", "bacon"])], # baz must be one of these options
    "qux": [Not(Range(1, 100))] # qux must not be a number between 1 and 100 inclusive
}

# then this following dict would pass:
passes = {
    "foo": 123,
    "bar": True, # or a non-empty string, or a non-zero int, etc...
    "baz": "spam",
    "qux": 101
}
>>> validate(rules, passes)
(True, {})

# but this one would fail
fails = {
    "foo": 321,
    "bar": False, # or 0, or [], or an empty string, etc...
    "baz": "barf",
    "qux": 99
}
>>> validate(rules, fails)
(False, {
 'foo': ["must be equal to 123"],
 'bar': ['must be True-equivalent value'],
 'baz': ["must be one of ['spam', 'eggs', 'bacon']"],
 'qux': ['must not fall between 1 and 100']
})
```

Notice that the validation that passed just returned True and an empty `dict`, but the one that failed returned a tuple with False and a `dict` with a list of related error messages for each key that failed. This lets you easily see exactly what failed in a human readable way.

CHAPTER 2

Installation

Stable releases can be installed via `pip install validator.py`. Alternatively, you can get the latest sources or a release tarball from http://github.com/mansam/validator.py.

`validator.py` is written with Python 2.7, but is tested with 2.6 and PyPy. It should also work with 2.5 and 3.x, though the tests currently won't run on 3.x.

CHAPTER 3

Getting Started with Validations

A validation (the set of rules used to test a dict) can be flat –consisting of just a single level of tests– or it can contain additional conditionally nested validations.

To create a validation, you insert a list of callables into a validation dictionary for each key/value pair in the dictionary you want to validate. When you call `validate` with the validation and your dictionary, each of those callables will be called with the respective value in your dictionary as their argument. If the callable returns `True`, then you're good to go. For example:

```
dictionary = {
    "foo": "bar"
}
validation = {
    "foo": [lambda x: x == "bar"]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

When `validate` got called in the example, the value of `dictionary["foo"]` got passed to lambda in the list, and since `dictionary["foo"] == "bar"`, everything is good and the dictionary is considered valid!

Writing your own callables is helpful in some cases, but `validator.py` helpfully provides a wide range of validations that should cover most of the common use cases.

CHAPTER 4

Available Validators

CHAPTER 5

The `Equals` validator

The `Equals` validator just checks that the dictionary value matches the parameter to `Equals`. We use it to rewrite our previous example more succinctly:

```
dictionary = {
    "foo": "bar"
}
validation = {
    "foo": [Equals("bar")]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

In the event that it fails, it explains so clearly:

```
>>> validate(validation, failure)
(False, {"foo": ["must be equal to 'baz'"]})
```

CHAPTER 6

The `Required` validator

By default, a key is considered optional. A key that's in the validation but isn't in the dictionary under test just gets silently skipped. To make sure that a key is present, use the `Required` validator. Adding the `Required` validator to the list of rules for a key ensures that the key must be present in the dictionary. Unlike most of the other validators that `validator.py` provides, `Required` shouldn't be written with parentheses.

```
dictionary = {
    "foo": "bar"
}
validation = {
    "foo": [Required, Equals("bar")]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

In the event that a key is missing:

```
failure = {}
>>> validate(validation, failure)
(False, {"foo": ["is missing"]})
```

CHAPTER 7

The `Truthy` validator

The `Truthy` validator checks that the dictionary value is something that Python treats as true. True, non-0 integers, non-empty lists, and strings all fall into this category.

```
dictionary = {
    "foo": 1
}
validation = {
    "foo": [Required, Truthy()]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

In the event that a key is not True-equivalent:

```
failure = {"foo": 0}
>>> validate(validation, failure)
(False, {"foo": ["must be True-equivalent value"]})
```

CHAPTER 8

The `Range` validator

The `Range` validator checks that the dictionary value falls inclusively between the start and end values passed to it.

```
dictionary = {
    "foo": 10
}
validation = {
    "foo": [Required, Range(1, 11)]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

If the value falls outside the specified range:

```
failure = {"foo": 12}
>>> validate(validation, failure)
(False, {"foo": ["must fall between 1 and 11"]})
```

You can also have Range exclude its endpoints by changing the *inclusive* keyword argument to false.

```
Range(1, 11, inclusive=False)
```

CHAPTER 9

The `Pattern` validator

The `Pattern` validator checks that the dictionary value matches the regex pattern that was passed to it.

```
dictionary = {
    "foo": "30%"
}
validation = {
    "foo": [Required, Pattern("\d\d\%")]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

If the value doesn't match the regex:

```
failure = {"foo": "99.0"}
>>> validate(validation, failure)
(False, {"foo": ["must match regex pattern \d\d\%"]})
```

CHAPTER 10

The `In` validator

The `In` validator checks that the dictionary value is a member of a collection passed to it.

```
dictionary = {
    "foo": "spam"
}
validation = {
    "foo": [Required, In(["spam", "eggs", "bacon"])]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

If the value doesn't belong to the collection:

```
failure = {"foo": "beans"}
>>> validate(validation, failure)
(False, {"foo": ["must be one of ['spam', 'eggs', 'bacon']"]})
```

CHAPTER 11

The `Not` validator

The `Not` validator negates a validator that is passed to it and checks the dictionary value against that negated validator.

```
dictionary = {
    "foo": "beans"
}
validation = {
    "foo": [Required, Not(In(["spam", "eggs", "bacon"]))]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

If the value doesn't pass the Not'd validator (meaning it would have passed the validator without the Not), then Not provides a helpfully negated version of the validator's error message:

```
failure = {"foo": "spam"}
>>> validate(validation, failure)
(False, {"foo": ["must not be one of ['spam', 'eggs', 'bacon']"]})
```

CHAPTER 12

The `InstanceOf` validator

The `InstanceOf` validator checks that the dictionary value is an instance of the base class passed to InstanceOf, or an instance of one of its subclasses.

```
dictionary = {
    "foo": u"i'm_a_unicode_string"
}
validation = {
    "foo": [Required, InstanceOf(basestring)]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

If the value isn't an instance of the base class or one of its subclasses:

```
failure = {"foo": object}
>>> validate(validation, failure)
(False, {"foo": ["must be an instance of basestring or its subclasses"]})
```

CHAPTER 13

The `SubclassOf` validator

The `SubclassOf` validator checks that the dictionary value inherits from the base class passed to it. To be clear, this means that the dictionary value is expected to be a class, not an instance of a class.

```
dictionary = {
    "foo": unicode
}
validation = {
    "foo": [Required, InstanceOf(basestring)]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

If the value isn't a subclass of base class or one of its subclasses (e.g. if it's an instance of that class or a subclass of something else):

```
failure = {"foo": "bar"}
>>> validate(validation, failure)
(False, {"foo": ["must be a subclass of basestring"]})
```

CHAPTER 14

The `Length` validator

The `Length` validator checks that value the must have at least *minimum* elements and optionally at most *maximum* elements.

```
dictionary = {
    "foo": [1, 2, 3, 4, 5]
}
validation = {
    "foo": [Length(0, maximum=5)]
}

>>> validate(validation, dictionary)
(True, {})
# Success!
```

CHAPTER 15

Conditional Validations

In some cases you might want to apply some rules only if other validations pass. You can do that with the `If(validator, Then(validation))` construct that `validator.py` provides. For example, you might want to ensure that `pet['name']` is a cat's name, but only if `pet['type'] == 'cat'`. To do this, you'd use the `If` validator on the key that serves as the condition for the other set of the rules.

```
pet = {
    "name": "whiskers",
    "type": "cat"
}
cat_name_rules = {
    "name": [In(["whiskers", "fuzzy", "tiger"])]
}
dog_name_rules = {
    "name": [In(["spot", "ace", "bandit"])]
}
validation = {
    "type": [
        If(Equals("cat"), Then(cat_name_rules)),
        If(Equals("dog"), Then(dog_name_rules))
    ]
}

>>> validate(validation, pet)
(True, {})
# Success!
```

A failed conditional validation will give you appropriately nested error messages so you know exactly where things went wrong.

```
pet = {"type":"cat", "name": "lily"}
>>> validate(validation, pet)
(False, {'type': [{'name': ["must be one of ['whiskers', 'fuzzy', 'tiger']"]}]})
```

CHAPTER 16

Nested Validations

You can nest validation dictionaries within each other in order to accommodate more complex data structures. Here's an example:

```
validator = {
    "foo": [Required, Equals(1)],
    "bar": [Required, {
             "baz": [Required, Equals(2)],
             "qux": [Required, {
                 "quux": [Required, Equals(3)]
             }]
         }
    ]
}
test_case = {
    "foo": 1,
    "bar": {
        "baz": 2,
        "qux": {
            "quux": 3
        }
    }
}
```

The above example says that the `bar` key represents a dictionary that also has its own set of validations. For good measure, this example has yet another dictionary under the `qux` key. As long as everything checks out, `validate` will return the normal `(True, {})` response indicating success.

In the event of failure, you get an appropriately nested error message like those produced by the conditional validator. Here's an example of what such an error might look like:

```
>>> validate(fails, test_case)
(False, {'bar': [{'baz': ['must be equal to 3'],
           'qux': [{'quux': ['must be equal to 4']}]}],
  'foo': ['must be equal to 2']})
```

This is very powerful, but you'll need to take care that you don't create conflicting validations or cyclic validations–`validator.py` won't be able to help you catch cycles.

CHAPTER 17

More Information

For more information, please visit http://github.com/mansam/validator.py or contact me at mansam@csh.rit.edu. You can also send me a message on freenode if you have any questions.

CHAPTER 18

Indices

- genindex
- search