Onctuous Documentation

Release 0.5.2

Jean-Tiare Le Bigot

November 21, 2012

CONTENTS

CHAPTER

ONE

OVERVIEW

Onctuous is a fluid and pleasing to use validation tool you will love to use. Originally based on Voluptuous code by Alec Thomas <alec@swapoff.org>, we first fixed long outstanding issues like Python builtins collision and added support for default values.

The goal of Onctuous is to make it simple and smooth.

- You *can* write your own validators
- You *can* specify defaults. The best ? They are *not* required to pass validation themselves
- You *can* write readable code. This is not based on json schema specification, on purpose

You can use Onctuous to validate `list`, `scalar` (regular variables) or `dict`. For this purpose, you will need to define a so-called `Schema` and call the Schema with the input to validate. In case of success, it will return the validated input, possibly filtered or edited according to your rules

CHAPTER

TWO

DOCUMENTATION

2.1 User guide

2.1.1 Getting started with Onctuous

Onctuous is a fluid and pleasing to use validation tool you will love to use. Originally based on Voluptuous code by Alec Thomas <alec@swapoff.org>, we first fixed long outstanding issues like Python builtins collision and added support for default values.

The goal of Onctuous is to make it simple and smooth.

- You *can* write your own validators
- You *can* specify defaults. The best ? They are *not* required to pass validation themselves
- You *can* write readable code. This is not based on json schema specification, on purpose

You can use Onctuous to validate `list`, `scalar` (regular variables) or `dict`. For this purpose, you will need to define a so-called `Schema` and call the Schema with the input to validate. In case of success, it will return the validated input, possibly filtered or edited according to your rules

Installation

```
$ pip install onctuous
```

Example usage

Validate a scalar

```
from onctuous import Schema

validate_is_int = Schema(int)

# Validate 42 (this will run fine)
validated = validate_is_int(42)

# Validate "toto" (this will raise ``InvalidList`` containing a list of errors)
validated = validate_is_int("toto")
```

Validate a list

Using the same idea, you can validate a list of `int`

```
from onctuous import Schema

validate_is_int_list = Schema([int])

# This will run fine
validated = validate_is_int_list([42, 2, 7])

# This will raise ''InvalidList'' containing a list of errors
validated = validate_is_int_list([2, 7, "toto"])
```

But we can also use on of the bundled validators and check the URL looks to be valid for example and even supply a custom error message!

```
from onctuous import Schema, Url

validate_is_urls = Schema([Url(msg="Ooops, this is *not* a valid URL")])

# This will run fine
validated = validate_is_urls(["www.example.com", "ftp://user:pass@ftp.example.com:42/toto?weird/path'

# This will raise ''InvalidList'' containing a list of errors
validated = validate_is_urls([2, 7, "toto"])
```

Validate a dictionary

Again, this is the same concept with some more niceties. For example, here is a basic user schema:

```
from onctuous import Schema, Url

validate_user = Schema({
    'firstname': unicode,
    'lastname': unicode,
    'age': int,
    'website': Url(msg="Ooops, this is *not* a valid URL"),
})

# use it...
```

But wait, I don't want megative ages, do I ?

```
from onctuous import Schema, Url, InRange, All

validate_user = Schema({
    'firstname': unicode,
    'lastname': unicode,
    'age': All(int, InRange(min=0, msg="Uh, ages can not be negative...")),
    'website': Url(msg="Ooops, this is *not* a valid URL"),
})

# use it...
```

Have you noticed how this uses `All` to specify that both `int` and `range` conditions must ne met ?

What if I want to make the "Website" field optional ? Let me introduce `Markers`

```
from onctuous import Schema, Url, InRange, All, Optional

validate_user = Schema({
    'firstname': unicode,
    'lastname': unicode,
    'age': All(int, InRange(min=0, msg="Uh, ages can not be negative...")),
    Optional('website'): Url(msg="Ooops, this is *not* a valid URL"),
})

# use it...
```

You could also have used the 'Required' Marker with a default value. This is very usefull if you do not want to spend your whole time writing `if key in data...`.

```
from onctuous import Schema, Url, InRange, All, Required

validate_user = Schema({
    'firstname': unicode,
    'lastname': unicode,
    'age': All(int, InRange(min=0, msg="Uh, ages can not be negative...")),
    Required('website', "#"): Url(msg="Ooops, this is *not* a valid URL"),
})

# use it...
```

It is worth noting that that the provided default value does *not* need to pass validations. You can use it as a "Marker" further in you application.

Nested and advanced validations

You can nest shemas. You actually did it in the previous example where scalars are nested into a dict or a list. But you can arbitrarily nest lists into dict and the other way around, as you need.

For example, let's say you are writing a blog post which obviously has an author and maybe some tags whose len are between 3 and 20 chars included.

```
from onctuous import Schema, All, Required, Length, InRange

# Same schema as user above. I just removed the Schema instanciation but
# could have kept it. It's just more natural
user = {
    'firstname': unicode,
    'lastname': unicode,
    'age': All(int, InRange(min=0, msg="Uh, ages can not be negative...")),
    Required('website', "#"): Url(msg="Ooops, this is *not* a valid URL"),
}

validate_post = Schema({
    'title': unicode,
    'body': unicode,
    'author': user,  # look how you can split a schema into re-usable chunks!
    Optional('tags'): [All(unicode, Length(min=3, max=20))],
    Required('website', "#"): Url(msg="Ooops, this is *not* a valid URL"),
})

# use it...
```

That's all for nesting.

You could also use the `Extra` special key to allow extra fields to be present while still being valid.

When instanciating the schema, there are also a global `required` and `extra` parameters that can optionally be set. They both default to `False`

Going further

There are tons of bundled validators, *see the full API documentation* for a full list.

2.1.2 Extending Onctuous

Folder structure

```
Onctuous
+-- onctuous            => the real code
|   +-- tests
|       +-- unit        => all individual validators and low level logic
|       `-- functional  => global behovior
+-- docs
    `-- pages           => what you are reading
```

Adding a custom validator

If you want to contribute to `Onctuous` (we would love it btw), you will need to add your custom validator into `ddmock.validators` module. Otherwise, put it wherever you want, there are no restrictions.

By convention, validators are:

- Callable
- Returns the validated value on success, even unmodified. This ensures chainability
- Raises `Invalid` on failure

All validators will look like this:

```
# Parent function: loads the parameters
def ValidatorName(param1, param2, msg=None):
    # this 'inner' function does the real job and is called by ``Onctuous``
    def f(v):
        if some condition:
            return v  # All changes done to the value will be reflected in the validated object
        raise Invalid(msg or 'Ooops: "Some Condition" was not met!')
    return f
```

For example, here is the `Url` validator:

```
def Url(msg=None):
    """Verify that the value is a URL."""
    def f(v):
        try:
            urlparse.urlparse(v)
            return v
        except:
            raise Invalid(msg or 'expected a URL')
    return f
```

That's all you need to do!

Adding a custom marker

Sadly, this is quite more invasive to do and will probably require you to patch the heart of `Onctuous`.

Markers lives in the same module as Validators: `ddmock.validators` and are also callable.

The most simple Marker you can do is the "Optional" marker:

```
class Optional(Marker):
    """Mark a node in the schema as optional."""
```

But you could override `__init__` or `__call__` for instance.

Then, Marker presence is detected in `Schema._validate_dict` in module `ddmock.schema`, that is to say, the heart of `Onctuous`

2.1.3 Change log - Migration guide.

Onctuous 0.5.2

This section documents all user visible changes included between Onctuous versions 0.5.1 and Onctuous versions 0.5.2

Changes

- packaging fixes
- `Coerce` is now idempotent

Onctuous 0.5.1

This section documents all user visible changes included between Onctuous versions 0.5.0 and Onctuous versions 0.5.1

Includes niceties and bugfixes according to real-world(tm) libraries ddbmock and dynamodb-mapper.

Additions

- official, full documentation

Changes

- split onctuous into sub-modules
- better error messages
- support for `None` as a default value

Onctuous 0.5.0

This section documents all user visible changes included between Voluptuous versions 0.4.2 and Onctuous versions 0.5.0

Initial Voluptuous fork by Ludia. There was no changelog before.

Additions

- `default` parameter to `Required` marker.
- 100% unit/functional tests
- lots comments

Changes

- Renamed all validators to avoid built-in collisions
- InvalidList does not accept empty `errors` array
- lots of code cleanups

Removal

- `defaults_to`. It was inneficient and failed to add default value.
- most doctests

2.2 Api reference

2.2.1 Errors

SchemaError class

class onctuous.errors.**SchemaError**
An error was encountered in the schema.

Invalid class

class onctuous.errors.**Invalid**(*message*, *path=None*)
The data was invalid.

Attr msg The error message.

Attr path The path to the error, as a list of keys in the source data.

InvalidList class

Class definition

class onctuous.errors.**InvalidList**(*errors*)
List of captures errors for reporting to the end user.

Attr errors Array of errors

Attr msg Message associated with the first reported error

Attr path Path associated with the first reported error

Public API

__init__
`InvalidList.__init__`(*errors*)
Create a new list of errors.

Parameters **errors** – list of errors to add initially

add
`InvalidList.add`(*error*)
Push an error to the internal list

2.2.2 Schema class

Class definition

class `onctuous.schema.Schema`(*schema*, *required=False*, *extra=False*)
A validation schema.

The schema is a Python tree-like structure where nodes are pattern matched against corresponding trees of values.

Nodes can be values, in which case a direct comparison is used, types, in which case an isinstance() check is performed, or callables, which will validate and optionally convert the value.

Public API

__init__
`Schema.__init__`(*schema*, *required=False*, *extra=False*)
Create a new Schema.

Parameters

- **schema** – Validation schema.
- **required** – Keys defined in the schema must be in the data.
- **extra** – Keys in the data need not have keys in the schema.

__call__
`Schema.__call__`(*data*)
Validate data against `self.schema`. This simply is a shortcut for `validate` method.

Parameters **data** – input data to validate

Returns validated input

Raise `InvalidList`

validate
`Schema.validate`(*path*, *schema*, *data*)
Validate data against this `schema`.

Parameters

- **path** – (list) current path in the object, Starts as `[]`
- **schema** – schema to validate agains
- **data** – input data to validate

Returns validated input

Raise `InvalidList`

2.2.3 Schema validators and markers

Markers

Marker base class

class `onctuous.validators.`**`Marker`**(*schema*, *msg=None*)
Mark nodes for special treatment.

Optional Class

class `onctuous.validators.`**`Optional`**(*schema*, *msg=None*)
Mark a node in the schema as optional.

Required Class

class `onctuous.validators.`**`Required`**(*schema*, *default=...*, *msg=None*)
Mark a node in the schema as being required.

Extra Method

`onctuous.validators.`**`Extra`**(*_*)
Allow keys in the data that are not present in the schema.

Validators

Msg

`onctuous.validators.`**`Msg`**(*schema*, *msg*)
Report a user-friendly message if a schema fails to validate. Messages are only applied to invalid direct descendants of the schema.

Coerce

`onctuous.validators.`**`Coerce`**(*target_type*, *msg=None*)
Coerce a value to a type. If the input value of the validator is already of this type, tha value is retured immediately to prevent stupid crash like datetime(datime()). This way, `Coerce` can safely be used to make sure the type is OK.

If the type constructor throws a ValueError, the value will be marked as Invalid.

Parameters **target_type** – target type for the coercion operation. May be any type or callable.

IsTrue

onctuous.validators.**IsTrue**(*msg=None*)
Assert that a value is true, in the Python sense. "In the Python sense" means that implicitly false values, such as empty lists, dictionaries, etc. are treated as "false":

IsFalse

onctuous.validators.**IsFalse**(*msg=None*)
Assert that a value is false, in the Python sense.

Boolean

onctuous.validators.**Boolean**(*msg=None*)
Convert human-readable boolean values to a bool.

Accepted values are 1, true, yes, on, enable, and their negatives. Non-string values are cast to bool.

Any

onctuous.validators.**Any**(**validators*, ***kwargs*)
Use the first validated value.

Parameters **msg** – Message to deliver to user if validation fails.

Returns Return value of the first validator that passes.

All

onctuous.validators.**All**(**validators*, ***kwargs*)
Value must pass all validators.

The output of each validator is passed as input to the next.

Parameters **msg** – Message to deliver to user if validation fails.

Match

onctuous.validators.**Match**(*pattern*, *msg=None*)
Value must match the regular expression.

Pattern may also be a compiled regular expression:

Sub

onctuous.validators.**Sub**(*pattern*, *substitution*, *msg=None*)
Regex substitution.

Url

onctuous.validators.**Url**(*msg=None*)
Verify that the value is a URL.

IsFile

onctuous.validators.**IsFile**(*msg=None*)
Verify the file exists.

IsDir

onctuous.validators.**IsDir**(*msg=None*)
Verify the directory exists.

PathExists

onctuous.validators.**PathExists**(*msg=None*)
Verify the path exists, regardless of its type.

InRange

onctuous.validators.**InRange**(*min=None*, *max=None*, *msg=None*)
Limit a value to a range.

Either min or max may be omitted.

Raises Invalid If the value is outside the range and clamp=False.

Clamp

onctuous.validators.**Clamp**(*min=None*, *max=None*, *msg=None*)
Clamp a value to a range.

Either min or max may be omitted.

Length

onctuous.validators.**Length**(*min=None*, *max=None*, *msg=None*)
The length of a value must be in a certain range.

ToLower

onctuous.validators.**ToLower**(*v*)
Transform a string to lower case.

ToUpper

onctuous.validators.**ToUpper**(*v*)
Transform a string to upper case.

Capitalize

onctuous.validators.**Capitalize**(*v*)
Capitalise a string.

Title

onctuous.validators.**Title**(*v*)
Title case a string.

2.3 Indices and tables

- *genindex*
- *modindex*
- *search*

CHAPTER

THREE

CONTRIBUTE

Want to contribute, report a but of request a feature ? The development goes on BitBucket:

- **Download**: http://pypi.python.org/pypi/onctuous
- **Report bugs**: https://bitbucket.org/Ludia/onctuous/issues
- **Fork the code**: https://bitbucket.org/Ludia/onctuous/overview