Cerberus Documentation

Release 0.9.9

Nicola Iarocci

September 23, 2015

Contents

This is folk project of https://github.com/nicolaiarocci/cerberus, Sanhe add support for `bytes`, `date` types, and add rules for `maxsize`, `minsize`, `before`, `after`.

Cerberus is a lightweight and extensible data validation library for Python.

> `CERBERUS`, n. The watch-dog of Hades, whose duty it was to guard the entrance; everybody, sooner or later, had to go there, and nobody wanted to carry off the entrance. - *Ambrose Bierce, The Devil's Dictionary*

Cerberus provides type checking and other base functionality out of the box and is designed to be easily extensible, allowing for easy custom validation. It has no dependencies and is thoroughly tested under Python 2.6, Python 2.7, Python 3.3, Python 3.4, PyPy and PyPy3.

CHAPTER 1

At a Glance

You define a validation schema and pass it to an instance of the *Validator* class:

```
>>> schema = {'name': {'type': 'string'}}
>>> v = Validator(schema)
```

Then you simply invoke the *validate()* to validate a dictionary against the schema. If validation succeeds, `True` is returned:

```
>>> document = {'name': 'john doe'}
>>> v.validate(document)
True
```

CHAPTER 2

Table of Contents

2.1 Cerberus Installation

This part of the documentation covers the installation of Cerberus. The first step to using any software package is getting it properly installed.

2.1.1 Stable Version

Cerberus is on PyPI so all you need to do is:

```
pip install cerberus
```

2.1.2 Development Version

Cerberus is actively developed on GitHub, where the code is always available. If you want to work with the development version, there are two ways: you can either let *pip* pull in the development version, or you can tell it to operate on a git checkout. Either way, virtualenv is recommended.

Get the git checkout in a new virtualenv and run in development mode.

```
$ git clone http://github.com/nicolaiarocci/cerberus.git
Initialized empty Git repository in ~/dev/cerberus.git/
$ cd cerberus
$ virtualenv venv --distribute
New python executable in venv/bin/python
Installing distribute.............done.
$ . venv/bin/activate
$ python setup.py install
...
Finished processing dependencies for Cerberus
```

This will pull in the dependencies and activate the git head as the current version inside the virtualenv. Then all you have to do is run `git pull origin` to update to the latest version.

To just get the development version without git, do this instead:

```
$ mkdir cerberus
$ cd cerberus
$ virtualenv venv --distribute
$ . venv/bin/activate
New python executable in venv/bin/python
```

```
Installing distribute.............done.
$ pip install git+git://github.com/nicolaiarocci/cerberus.git
...
Cleaning up...
```

And you're done!

2.2 Cerberus Usage

2.2.1 Basic Usage

You define a validation schema and pass it to an instance of the *Validator* class:

```
>>> schema = {'name': {'type': 'string'}}
>>> v = Validator(schema)
```

Then you simply invoke the *validate()* to validate a dictionary against the schema. If validation succeeds, `True` is returned:

```
>>> document = {'name': 'john doe'}
>>> v.validate(document)
True
```

Alternatively, you can pass both the dictionary and the schema to the *validate()* method:

```
>>> v = Validator()
>>> v.validate(document, schema)
True
```

Which can be handy if your schema is changing through the life of the instance.

Unlike other validation tools, Cerberus will not halt and raise an exception on the first validation issue. The whole document will always be processed, and `False` will be returned if validation failed. You can then access the *errors()* method to obtain a list of issues.

```
>>> schema = {'name': {'type': 'string'}, 'age': {'type': 'integer', 'min': 10}}
>>> document = {'name': 'Little Joe', 'age': 5}
>>> v.validate(document, schema)
False
>>> v.errors
{'age': 'min value is 10'}
```

You will still get *SchemaError* and `DocumentError` exceptions.

Changed in version 0.4.1: The Validator class is callable, allowing for the following shorthand syntax:

```
>>> document = {'name': 'john doe'}
>>> v(document)
True
```

2.2.2 Validation Schema

A validation schema is a dictionary. Schema keys are the keys allowed in the target dictionary. Schema values express the rules that must be matched by the corresponding target values.

```
schema = {'name': {'type': 'string', 'maxlength': 10}}
```

In the example above we define a target dictionary with only one key, `name`, which is expected to be a string not longer than 10 characters. Something like `{'name': 'john doe'}` would validate, while something like `{'name': 'a very long string'}` or `{'name': 99}` would not.

By definition all keys are optional unless the *required* rule is set for a key.

2.2.3 Validation Rules

The following rules are currently supported:

type

Data type allowed for the key value. Can be one of the following:

- `string`
- `bytes`
- `integer`
- `float`
- `number` (integer or float)
- `boolean`
- `datetime`
- `date`
- `dict` (formally `collections.mapping`)
- `list` (formally `collections.sequence`, excluding strings)
- `set`

A list of types can be used to allow different values:

```
>>> v.schema = {'quotes': {'type': ['string', 'list']}}
>>> v.validate({'quotes': 'Hello world!'})
True
>>> v.validate({'quotes': ['Do not disturb my circles!', 'Heureka!']})
True
```

```
>>> v.schema = {'quotes': {'type': ['string', 'list'], 'schema': {'type': 'string'}}}
>>> v.validate({'quotes': 'Hello world!'})
True
>>> v.validate({'quotes': [1, 'Heureka!']})
False
>>> v.errors
{'quotes': {0: 'must be of string type'}}
```

You can extend this list and support custom types, see *Custom Data Types*.

Note: Please note that type validation is performed before any other validation rule which might exist on the same field (only exception being the `nullable` rule). In the occurrence of a type failure subsequent validation rules on the field will be skipped and validation will continue on other fields. This allows to safely assume that field type is correct when other (standard or custom) rules are invoked.

Changed in version 0.9: If a list of types is given, the key value must match *any* of them.

Changed in version 0.7.1: `dict` and `list` typechecking are now performed with the more generic `Mapping` and `Sequence` types from the builtin `collections` module. This means that instances of custom types designed to the same interface as the builtin `dict` and `list` types can be validated with Cerberus. We exclude strings when type checking for `list`/`Sequence` because it in the validation situation it is almost certain the string was not the intended data type for a sequence.

Changed in version 0.7: Added the `set` data type.

Changed in version 0.6: Added the `number` data type.

Changed in version 0.4.0: Type validation is always executed first, and blocks other field validation rules on failure.

Changed in version 0.3.0: Added the `float` data type.

required

If `True` the key/value pair is mandatory. Validation will fail when it is missing, unless *`validate()`* is called with `update=True`:

```
>>> v.schema = {'name': {'required': True, 'type': 'string'}, 'age': {'type': 'integer'}}
>>> document = {'age': 10}
>>> v.validate(document)
False
>>> v.errors
{'name': 'required field'}

>>> v.validate(document, update=True)
True
```

Note: String fields with empty values will still be validated, even when `required` is set to `True`. If you don't want to accept empty values, see the *empty* rule. Also, if *dependencies* are declared for the field, its `required` rule will only be validated if all dependencies are included with the document.

Changed in version 0.8: Check field dependencies.

readonly

If `True` the value is readonly. Validation will fail if this field is present in the target dictionary.

nullable

If `True` the field value can be set to `None`. It is essentially the functionality of the `ignore_none_values` parameter of the *Validator Class*, but allowing for more fine grained control down to the field level.

```
>>> v.schema = {'a_nullable_integer': {'nullable': True, 'type': 'integer'}, 'an_integer': {'type':

>>> v.validate({'a_nullable_integer': 3})
True
>>> v.validate({'a_nullable_integer': None})
True

>>> v.validate({'an_integer': 3})
True
>>> v.validate({'an_integer': None})
```

```
False
>>> v.errors
{'an_integer': 'null value not allowed'}
```

Changed in version 0.7: `nullable` is valid on fields lacking type definition.

New in version 0.3.0.

minlength, maxlength

Minimum and maximum length allowed for `string` and `list` types.

min, max

Minimum and maximum value allowed for `integer`, `float` and `number` types.

minsize, maxsize

Minimum and maximum size in bytes allowed for `bytes` types.

before, after

Minimum and maximum bound allowed for `datetime` and `date` types.

Changed in version 0.7: Added support for `float` and `number` types.

allowed

Allowed values for `string`, `list` and `int` types. Validation will fail if target values are not included in the allowed list.

```
>>> v.schema = {'role': {'type': 'list', 'allowed': ['agent', 'client', 'supplier']}}
>>> v.validate({'role': ['agent', 'supplier']})
True

>>> v.validate({'role': ['intern']})
False
>>> v.errors
{'role': "unallowed values ['intern']"}

>>> v.schema = {'role': {'type': 'string', 'allowed': ['agent', 'client', 'supplier']}}
>>> v.validate({'role': 'supplier'})
True

>>> v.validate({'role': 'intern'})
False
>>> v.errors
{'role': 'unallowed value intern'}

>>> v.schema = {'a_restricted_integer': {'type': 'integer', 'allowed': [-1, 0, 1]}}
>>> v.validate({'a_restricted_integer': -1})
True
```

```
>>> v.validate({'a_restricted_integer': 2})
False
>>> v.errors
{'a_restricted_integer': 'unallowed value 2'}
```

Changed in version 0.5.1: Added support for the `int` type.

empty

Only applies to string fields. If `False` validation will fail if the value is empty. Defaults to `True`.

```
>>> schema = {'name': {'type': 'string', 'empty': False}}
>>> document = {'name': ''}
>>> v.validate(document, schema)
False

>>> v.errors
{'name': 'empty values not allowed'}
```

New in version 0.0.3.

items (dict)

Deprecated since version 0.0.3: Use *schema (dict)* instead.

When a dictionary, `items` defines the validation schema for items in a `list` type:

```
>>> schema = {'rows': {'type': 'list', 'items': {'sku': {'type': 'string'}, 'price': {'type': 'intege
>>> document = {'rows': [{'sku': 'KT123', 'price': 100}]}
>>> v.validate(document, schema)
True
```

Note: The *items (dict)* rule is deprecated, and will be removed in a future release.

items (list)

When a list, `items` defines a list of values allowed in a `list` type of fixed length in the given order:

```
>>> schema = {'list_of_values': {'type': 'list', 'items': [{'type': 'string'}, {'type': 'integer'}]}}
>>> document = {'list_of_values': ['hello', 100]}
>>> v.validate(document, schema)
True
>>> document = {'list_of_values': [100, 'hello']}
>>> v.validate(document, schema)
False
```

See *schema (dict)* rule below for dealing with arbitrary length `list` types.

schema (dict)

Validation rules for *Mappings*-fields.

```
>>> schema = {'a_dict': {'type': 'dict', 'schema': {'address': {'type': 'string'}, 'city': {'type':
>>> document = {'a_dict': {'address': 'my address', 'city': 'my town'}}
>>> v.validate(document, schema)
True
```

Note: If all keys should share the same validation rules you probably want to use *valueschema* instead.

schema (list)

You can also use this rule to validate arbitrary length *Sequence*-items.

```
>>> schema = {'a_list': {'type': 'list', 'schema': {'type': 'integer'}}}
>>> document = {'a_list': [3, 4, 5]}
>>> v.validate(document, schema)
True
```

The *schema* rule on `list` types is also the prefered method for defining and validating a list of dictionaries.

```
>>> schema = {'rows': {'type': 'list', 'schema': {'type': 'dict', 'schema': {'sku': {'type': 'string
>>> document = {'rows': [{'sku': 'KT123', 'price': 100}]}
>>> v.validate(document, schema)
True
```

Changed in version 0.0.3: Schema rule for `list` types of arbitrary length

valueschema

Validation schema for all values of a `dict`. The `dict` can have arbitrary keys, the values for all of which must validate with given schema:

```
>>> schema = {'numbers': {'type': 'dict', 'valueschema': {'type': 'integer', 'min': 10}}}
>>> document = {'numbers': {'an integer': 10, 'another integer': 100}}
>>> v.validate(document, schema)
True

>>> document = {'numbers': {'an integer': 9}}
>>> v.validate(document, schema)
False

>>> v.errors
{'numbers': {'an integer': 'min value is 10'}}
```

New in version 0.7.

Changed in version 0.9: renamed `keyschema` to `valueschema`

propertyschema

This is the counterpart to `valueschema` that validates the *keys* of a `dict`. For historical reasons it is *not* named `keyschema`.

```
>>> schema = {'a_dict': {'type': 'dict', 'propertyschema': {'type': 'string', 'regex': '[a-z]+'}}}
>>> document = {'a_dict': {'key': 'value'}}
>>> v.validate(document, schema)
True
```

```
>>> document = {'a_dict': {'KEY': 'value'}}
>>> v.validate(document, schema)
False
```

New in version 0.9.

regex

Validation will fail if field value does not match the provided regex rule. Only applies to string fiels.

```
>>> schema = {'email': {'type': 'string', 'regex': '^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$
>>> document = {'email': 'john@example.com'}
>>> v.validate(document, schema)
True

>>> document = {'email': 'john_at_example_dot_com'}
>>> v.validate(document, schema)
False

>>> v.errors
{'email': "value does not match regex '^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$'"}
```

For details on regex rules, see Regular Expressions Syntax on Python official site.

New in version 0.7.

dependencies

This rule allows for either a list or dict of dependencies. When a list is provided, all listed fields must be present in order for the target field to be validated.

```
>>> schema = {'field1': {'required': False}, 'field2': {'required': False, 'dependencies': ['field1']
>>> document = {'field1': 7}
>>> v.validate(document, schema)
True

>>> document = {'field2': 7}
>>> v.validate(document, schema)
False

>>> v.errors
{'field2': "field 'field1' is required"}
```

When a dictionary is provided, then not only all dependencies must be present, but also any of their allowed values must be matched.

```
>>> schema = {'field1': {'required': False}, 'field2': {'required': True, 'dependencies': {'field1':

>>> document = {'field1': 'one', 'field2': 7}
>>> v.validate(document, schema)
True

>>> document = {'field1': 'three', 'field2': 7}
>>> v.validate(document, schema)
False
>>> v.errors
{'field2': "field 'field1' is required with one of these values: ['one', 'two']"}
```

```
>>> # same as using a dependencies list
>>> document = {'field2': 7}
>>> v.validate(document, schema)
False
>>> v.errors
{'field2': "field 'field1' is required with one of these values: ['one', 'two']"}

>>> # one can also pass a single dependency value
>>> schema = {'field1': {'required': False}, 'field2': {'dependencies': {'field1': 'one'}}}
>>> document = {'field1': 'one', 'field2': 7}
>>> v.validate(document, schema)
True

>>> document = {'field1': 'two', 'field2': 7}
>>> v.validate(document, schema)
False

>>> v.errors
{'field2': "field 'field1' is required with one of these values: ['one']"}
```

Dependencies on sub-document fields are also supported:

```
>>> schema = {
...   'test_field': {'dependencies': ['a_dict.foo', 'a_dict.bar']},
...   'a_dict': {
...     'type': 'dict',
...     'schema': {
...       'foo': {'type': 'string'},
...       'bar': {'type': 'string'}
...     }
...   }
... }

>>> document = {'test_field': 'foobar', 'a_dict': {'foo': 'foo'}}
>>> v.validate(document, schema)
False

>>> v.errors
{'test_field': "field 'a_dict.bar' is required"}
```

Changed in version 0.8.1: Support for sub-document fields as dependencies.

Changed in version 0.8: Support for dependencies as a dictionary.

New in version 0.7.

*of-rules

These rules allow you to list multiple sets of rules to validate against. The field will be considered valid if it validates against the set in the list according to the prefixes logics `all`, `any`, `one` or `none`.

New in version 0.9.

anyof

Validates if *any* of the provided constraints validates the field.

allof

Validates if *all* of the provided constraints validates the field.

noneof

Validates if *none* of the provided constraints validates the field.

oneof

Validates if *exactly one* of the provided constraints applies.

For example, to verify that a property is a number between 0 and 10 or 100 and 110, you could do the following:

```
>>> schema = {'prop1':
...            {'type': 'number',
...             'anyof':
...             [{'min': 0, 'max': 10}, {'min': 100, 'max': 110}]}}

>>> document = {'prop1': 5}
>>> v.validate(document, schema)
True

>>> document = {'prop1': 105}
>>> v.validate(document, schema)
True

>>> document = {'prop1': 55}
>>> v.validate(document, schema)
False
>>> v.errors
{'prop1': {'anyof': 'no definitions validated', 'definition 1': 'min value is 100', 'definition 0':
```

The `anyof` rule works by creating a new instance of a schema for each item in the list. The above schema is equivalent to creating two separate schemas:

```
>>> schema1 = {'prop1': {'type': 'number', 'min':   0, 'max':  10}}
>>> schema2 = {'prop1': {'type': 'number', 'min': 100, 'max': 110}}

>>> document = {'prop1': 5}
>>> v.validate(document, schema1) or v.validate(document, schema2)
True

>>> document = {'prop1': 105}
>>> v.validate(document, schema1) or v.validate(document, schema2)
True

>>> document = {'prop1': 55}
>>> v.validate(document, schema1) or v.validate(document, schema2)
False
```

*of-rules typesaver

You can concatenate any of-rule with an underscore and another rule with a list of rule-values to save typing:

```
{'foo': {'anyof_type': ['string', 'integer']}}
# is equivalent to
{'foo': {'anyof': [{'type': 'string'}, {'type': 'integer'}]}}
```

Thus you can use this to validate a document against several schemas without implementing your own logic:

```
>>> schemas = [{'department': {'required': True, 'regex': '^IT$'}, 'phone': {'nullable': True}},
...            {'department': {'required': True}, 'phone': {'required': True}}]
>>> emloyee_vldtr = Validator({'employee': {'oneof_schema': schemas, 'type': 'dict'}}, allow_unknown=
>>> invalid_employees_phones = []
>>> for employee in employees:
...     if not employee_vldtr.validate(employee):
...         invalid_employees_phones.append(employee)
```

excludes

You can declare fields to excludes others:

```
>>> v = Validator()
>>> schema = {'this_field': {'type': 'dict',
...                          'excludes': 'that_field'},
...           'that_field': {'type': 'dict',
...                          'excludes': 'this_field'}}
>>> v.validate({'this_field': {}, 'that_field': {}}, schema)
False
>>> v.validate({'this_field': {}}, schema)
True
>>> v.validate({'that_field': {}}, schema)
True
>>> v.validate({}, schema)
True
```

You can require both field to build an exclusive *or*:

```
>>> v = Validator()
>>> schema = {'this_field': {'type': 'dict',
...                          'excludes': 'that_field',
...                          'required': True},
...           'that_field': {'type': 'dict',
...                          'excludes': 'this_field',
...                          'required': True}}
>>> v.validate({'this_field': {}, 'that_field': {}}, schema)
False
>>> v.validate({'this_field': {}}, schema)
True
>>> v.validate({'that_field': {}}, schema)
True
>>> v.validate({}, schema)
False
```

You can also pass multiples fields to exclude in a list :

```
>>> schema = {'this_field': {'type': 'dict',
...                          'excludes': ['that_field', 'bazo_field']},
...           'that_field': {'type': 'dict',
...                          'excludes': 'this_field'},
...           'bazo_field': {'type': 'dict'}}
```

```
>>> v.validate({'this_field': {}, 'bazo_field': {}}, schema)
False
```

2.2.4 Allowing the Unknown

By default only keys defined in the schema are allowed:

```
>>> schema = {'name': {'type': 'string', 'maxlength': 10}}
>>> v.validate({'name': 'john', 'sex': 'M'}, schema)
False
>>> v.errors
{'sex': 'unknown field'}
```

However, you can allow unknown key/value pairs by either setting `allow_unknown` to `True`:

```
>>> v.schema = {}
>>> v.allow_unknown = True
>>> v.validate({'name': 'john', 'sex': 'M'})
True
```

Or you can set `allow_unknown` to a validation schema, in which case unknown fields will be validated against it:

```
>>> v.schema = {}
>>> v.allow_unknown = {'type': 'string'}
>>> v.validate({'an_unknown_field': 'john'})
True
>>> v.validate({'an_unknown_field': 1})
False
>>> v.errors
{'an_unknown_field': 'must be of string type'}
```

`allow_unknown` can also be set at initialization:

```
>>> v.schema = {}
>>> v.allow_unknown = True
>>> v.validate({'name': 'john', 'sex': 'M'})
True
```

`allow_unknown` can also be set as rule to configure a validator for a nested mapping that is checked against the `schema`-rule:

```
>>> v = Validator()
>>> v.allow_unknown
False

>>> schema = {
...     'name': {'type': 'string'},
...     'a_dict': {
...       'type': 'dict',
...       'allow_unknown': True,  # this overrides the behaviour for
...       'schema': {             # the validation of this definition
...         'address': {'type': 'string'}
...       }
...     }
... }

>>> v.validate({'name': 'john', 'a_dict':{'an_unknown_field': 'is allowed'}}, schema)
True
```

```
>>> # this fails as allow_unknown is still False for the parent document.
>>> v.validate({'name': 'john', 'an_unknown_field': 'is not allowed', 'a_dict':{'an_unknown_field':
False

>>> v.errors
{'an_unknown_field': 'unknown field'}
```

Changed in version 0.9: `allow_unknown` can also be set for nested dict fields.

Changed in version 0.8: `allow_unknown` can also be set to a validation schema.

2.2.5 Normalization Rules

Renaming Of Fields

You can define a field to be renamed before any further processing.

```
>>> v = Validator({'foo': {'rename': 'bar'}})
>>> v.normalized({'foo': 0})
{'bar': 0}
```

To let a callable rename a field or arbitrary fields, you can define a handler for renaming:

```
>>> v = Validator({}, allow_unknown={'rename_handler': int})
>>> v.normalized({'0': 'foo'})
{0: 'foo'}
```

Purging Unknown Fields

After renaming, unknown fields will be purged if the `purge_unknown`-property of a `Validator`-instance is `True`. You can set the property per keyword-argument upon initialization or as rule for subdocuments like `allow_unknown`. The default is `False`.

```
>>> v = Validator({'foo': {'type': 'string'}}, purge_unknown=True)
>>> v.normalized({'bar': 'foo'})
{}
```

Value Coercion

Coercion allows you to apply a callable to a value before the document is validated. The return value of the callable replaces the new value in the document. This can be used to convert values or sanitize data before it is validated.

```
>>> v.schema = {'amount': {'type': 'integer'}}
>>> v.validate({'amount': '1'})
False

>>> v.schema = {'amount': {'type': 'integer', 'coerce': int}}
>>> v.validate({'amount': '1'})
True
>>> v.document
{'amount': 1}

>>> to_bool = lambda v: v.lower() in ['true', '1']
>>> v.schema = {'flag': {'type': 'boolean', 'coerce': to_bool}}
```

```
>>> v.validate({'flag': 'true'})
True
>>> v.document
{'flag': True}
```

New in version 0.9.

2.2.6 Fetching Processed Documents

Beside the `document`-property a `Validator`-instance has shorthand methods to process a document and fetch its processed result.

validated Method

There's a wrapper-method `validated` that returns the validated document. If the document didn't validate `None` is returned. It can be useful for flows like this:

```
v = Validator(schema)
valid_documents = [x for x in [v.validated(y) for y in documents] if x is not None]
```

If a coercion callable raises a `TypeError` or `ValueError` then the exception will be caught and the validation with fail. All other exception pass through.

New in version 0.9.

normalized Method

Similary, the `normalized`-method returns a normalized copy of a document without validating it:

```
>>> schema = {'amount': {'coerce': int}}
>>> document = {'model': 'consumerism', 'amount': '1'}
>>> normalized_document = v.normalized(document, schema)
>>> type(normalized_document['amount'])
<type 'int'>
```

New in version 0.10.

2.2.7 Schema Definition Formats

Cerberus schemas are built with vanilla Python types: `dict`, `list`, `string`, etc. Even user-defined validation rules are invoked in the schema by name, as a string. A useful side effect of this design is that schemas can be defined in a number of ways, for example with PyYAML.

```
>>> import yaml
>>> schema_text = '''
... name:
...   type: string
... age:
...   type: integer
...   min: 10
... '''
>>> schema = yaml.load(schema_text)
>>> document = {'name': 'Little Joe', 'age': 5}
>>> v.validate(document, schema)
```

```
False
>>> v.errors
{'age': 'min value is 10'}
```

You don't have to use YAML of course, you can use your favorate serializer. JSON for example. As long as there is a decoder thant can produce a nested `dict`, you can use it to define a schema.

2.3 Extending Cerberus

2.3.1 Custom Validators

Cerberus supports custom validation in two styles:

- *Class-based Custom Validators*
- *Function-based Custom Validation*

As a general rule, when you are customizing validators in your application, `Class-based` style is more suitable for common validators, which are also more human-readable (since the rule name is defined by yourself), while `Function-based` style is more suitable for special and one-off ones.

Class-based Custom Validators

Suppose that in our use case some values can only be expressed as odd integers, therefore we decide to add support for a new `isodd` rule to our validation schema:

```
schema = {'oddity': {'isodd': True, 'type': 'integer'}}
```

This is how we would go to implement that:

```
from cerberus import Validator

class MyValidator(Validator):
    def _validate_isodd(self, isodd, field, value):
        if isodd and not bool(value & 1):
            self._error(field, "Must be an odd number")
```

By subclassing Cerberus *`Validator`* class and adding the custom `_validate_<rulename>` function, we just enhanced Cerberus to suit our needs. The custom rule `isodd` is now available in our schema and, what really matters, we can use it to validate all odd values:

```
>>> v = MyValidator(schema)
>>> v.validate({'oddity': 10})
False
>>> v.errors
{'oddity': 'Must be an odd number'}
>>> v.validate({'oddity': 9})
True
```

In a schema schema you can use space characters instead of underscores, e.g. `{'oddity': {'is odd': 42'}}` is an alias for `{'oddity': {'is_odd': 42'}}`.

New in version 0.7.1: Custom validators also have access to a special `self.document` variable that allows validation of a field to happen in context of the rest of the document.

To make use of additional contextual information in a sub-class of *`Validator`*, use a pattern like this:

```
class MyValidator(Validator):
    def __init__(self, *args, **kwargs):
        if 'additional_context' in kwargs:
            self.additional_context = kwargs['additional_context']
        super(MyValidator, self).__init__(*args, **kwargs)

    def _validate_type_foo(self, field, value):
        make_use_of(self.additional_context)
```

New in version 0.9.

Custom Data Types

Cerberus supports and validates several standard data types (see *type*). When building *Class-based Custom Validators* you can add and validate your own data types. For example Eve (a tool for quickly building and deploying RESTful Web Services) supports a custom `objectid` type, which is used to validate that field values conform to the BSON/MongoDB `ObjectId` format.

You extend the supported set of data types by adding a `_validate_type_<typename>` method to your own *`Validator`* subclass. This snippet, directly from Eve source, shows how the `objectid` has been implemented:

```
def _validate_type_objectid(self, field, value):
    """ Enables validation for `objectid` schema attribute.

    :param field: field name.
    :param value: field value.
    """
    if not re.match('[a-f0-9]{24}', value):
        self._error(field, ERROR_BAD_TYPE.format('ObjectId'))
```

New in version 0.0.2.

Function-based Custom Validation

With a special rule `validator`, you can customize validators by defining your own functions with the following prototype:

```
def validate_<fieldname>(field, value, error):
    pass
```

As a contrast, if the odd value is a special case, you may want to make the above rule `isodd` into `Function-based` style, which is a more lightweight alternative:

```
def validate_oddity(field, value, error):
    if not bool(value & 1):
        error(field, "Must be an odd number")
```

Then, you can validate an odd value like this:

```
>>> schema = {'oddity': {'validator': validate_oddity}}
>>> v = Validator(schema)
>>> v.validate({'oddity': 10})
False
>>> v.errors
{'oddity': 'Must be an odd number'}
```

```
>>> v.validate({'oddity': 9})
True
```

New in version 0.8.

Limitations

You must not call your custom rule `validator` and it may be a bad idea to overwrite particular contributed rules.

Relevant *Validator*-properties

Validator.__get_child_validator

If you need another instance of your `Validator`-subclass, the `__get_child_validator`-method returns another instance that is initiated with the same arguments as `self` was. You can specify overriding keyword- arguments.

New in version 0.9.

Validator.root_document

A child-validator - as used when validating a `schema` - can access the first generation validator's document that is being processed via its `root_document`-property. It's untested what happens when you change that. It may make `boom`.

New in version 0.10.

2.4 How to Contribute

Contributions are welcome! Not familiar with the codebase yet? No problem! There are many ways to contribute to open source projects: reporting bugs, helping with the documentation, spreading the word and of course, adding new features and patches.

2.4.1 Getting Started

1. Make sure you have a GitHub account.
2. Open a new issue, assuming one does not already exist.
3. Clearly describe the issue including steps to reproduce when it is a bug.

2.4.2 Making Changes

- Fork the repository on GitHub.
- Create a topic branch from where you want to base your work.
- This is usually the `master` branch.
- Please avoid working directly on `master` branch.
- Make commits of logical units (if needed rebase your feature branch before submitting it).

- Check for unnecessary whitespace with `git diff --check` before committing.
- Make sure your commit messages are in the proper format.
- If your commit fixes an open issue, reference it in the commit message (#15).
- Make sure your code comforms to PEP8.
- Make sure you have added the necessary tests for your changes.
- Run all the tests to assure nothing else was accidentally broken.
- Don't forget to add yourself to AUTHORS.

These guidelines also apply when helping with documentation (actually, for typos and minor additions you might choose to fork and edit).

2.4.3 Submitting Changes

- Push your changes to a topic branch in your fork of the repository.
- Submit a Pull Request.
- Wait for maintainer feedback.

2.4.4 First time contributor?

It's alright. We've all been there.

2.4.5 Dont' know where to start?

There are usually several TODO comments scattered around the codebase, maybe check them out and see if you have ideas, or can help with them. Also, check the open issues in case there's something that sparks your interest. What about documentation? I suck at english so if you're fluent with it (or notice any error), why not help with that? In any case, other than GitHub help pages, you might want to check this excellent Effective Guide to Pull Requests

2.4.6 Running the Tests

Cerberus runs under Python 2.6, 2.7, Python 3.3, Python 3.4 and PyPy. Therefore tests will be run in those four platforms in our continuous integration server.

The easiest way to get started is to run the tests in your local environment with:

```
$ python setup.py test
```

Testing with other Python versions

Before you submit a pull request, make sure your tests and changes run in all supported python versions: 2.6, 2.7, 3.3, 3.4 and PyPy. Instead of creating all those environments by hand, Cerberus uses tox.

Make sure you have all required python versions installed and run:

```
$ pip install tox  # First time only
$ tox
```

This might take some time the first run as the different virtual environments are created and dependencies are installed. If everything is ok, you will see the following:

```
_________ summary _________
py26: commands succeeded
py27: commands succeeded
py33: commands succeeded
py34: commands succeeded
pypy: commands succeeded
flake8: commands succeeded
congratulations :)
```

If something goes **wrong** and one test fails, you might need to run that test in the specific python version. You can use the created environments to run some specific tests. For example, if a test suite fails in Python 3.4:

```
# From the project folder
$ tox -e py34
```

Have a look at `/tox.ini` for the available test environments and their workings.

Using Pytest

You also choose to run the whole test suite using pytest:

```
# Run the whole test suite
$ py.test
```

Using Docker

If you have a running Docker-daemon running you can run tests from a container that has the necessary interpreters and packages installed and pass arguments to `tox`:

```
# from the project's root directory
$ ./run-docker-tests -e pypy3 -e doctest
```

You can run the script without any arguments to test the project exactly as *Continuous Integration* does without having to setup anything. If there's a directoy `.tox` in the project-folder it will be used to store and access cached virtual environments and test-logs.

Continuous Integration

Each time code is pushed to the `master` branch the whole test-suite is executed on Travis-CI. This is also the case for pull-requests. A box at the bottom of its conversation-view will inform about the tests' status. The contributor can then fix the code, add commits, squash the commits and push again. The CI will also run flake8 so make sure that your code complies to PEP8 and test links and sample-code in the documentation.

2.4.7 Source Code

Source code is available at GitHub.

2.5 API Documentation

2.5.1 Validator Class

class cerberus.**Validator**(**args*, ***kwargs*)

Validator class. Validates any Python dict against a validation schema,

which is provided as an argument at class instantiation, or upon calling the `validate()` method.

Parameters

- **schema** – optional validation schema.
- **transparent_schema_rules** – if `True` unknown schema rules will be ignored (no SchemaError will be raised). Defaults to `False`. Useful you need to extend the schema grammar beyond Cerberus' domain.
- **ignore_none_values** – If `True` it will ignore None values for type checking. (no UnknownType error will be added). Defaults to `False`. Useful if your document is composed from function kwargs with defaults.
- **allow_unknown** – if `True` unknown key/value pairs (not present in the schema) will be ignored, and validation will pass. Defaults to `False`, returning an 'unknown field error' un validation.

Changed in version 0.9.1: 'required' will always be validated, regardless of any dependencies.

New in version 0.9: 'anyof', 'noneof', 'allof', 'anyof' validation rules. PyPy support. 'coerce' rule. 'propertyschema' validation rule. 'validator.validated' takes a document as argument and returns a Validated document or 'None' if validation failed.

Changed in version 0.9: Use 'str.format' in error messages so if someone wants to override them does not get an excpetion if arguments are not passed. 'keyschema' is renamed to 'valueschema'. Closes #92. 'type' can be a list of valid types. Usages of 'document' to 'self.document' in '_validate'. When 'items' is applied to a list, field name is used as key for 'validator.errors', and offending field indexes are used as keys for Field errors ({'a_list_of_strings': {1: 'not a string'}}) Additional kwargs that are passed to the __init__-method of an Instance of Validator-(sub-)class are passed to child-validators. Ensure that additional **kwargs of a subclass persist through validation Improve failure message when testing against multiple types. Ignore 'keyschema' when not a mapping. Ignore 'schema' when not a sequence. 'allow_unknown' can also be set for nested dicts. Closes #75. Raise SchemaError when an unallowed 'type' is used in conjunction with 'schema' rule.

Changed in version 0.8.1: 'dependencies' for sub-document fields. Closes #64. 'readonly' should be validated before any other validation. Closes #63. 'allow_unknown' does not apply to sub-dictionaries in a list. Closes #67. update mode does not ignore required fields in subdocuments. Closes #72. 'allow_unknown' does not respect custom rules. Closes #66.

New in version 0.8: 'dependencies' also support a dict of dependencies. 'allow_unknown' can be a schema used to validate unknown fields. Support for function-based validation mode.

Changed in version 0.7.2: Successfully validate int as a float type.

Changed in version 0.7.1: Validator options like 'allow_unknown' and 'ignore_none_values' are now taken into consideration when validating sub-dictionaries. Make self.document always the root level document. Up-front validation for schemas.

New in version 0.7: 'keyschema' validation rule. 'regex' validation rule. 'dependencies' validation rule. 'mix', 'max' now apply on floats and numbers too. Closes #30. 'set' data type.

New in version 0.6: 'number' (integer or float) validator.

Changed in version 0.5.0: `validator.errors` returns a dict where keys are document fields and values are validation errors.

Changed in version 0.4.0: `validate_update()` is deprecated. Use `validate()` with `update=True` instead. Type validation is always performed first (only exception being `nullable`). On failure, it blocks other rules on the same field. Closes #18.

New in version 0.2.0: *self.errors* returns an empty list when validate() has not been called. Option so allow nullable field values. Option to allow unknown key/value pairs.

New in version 0.1.0: Option to ignore None values for type checking.

New in version 0.0.3: Support for transparent schema rules. Added new 'empty' rule for string fields.

New in version 0.0.2: Support for addition and validation of custom data types.

current
Get the current document being validated.

When validating, the current (sub)document will be available via this property.

errors

Return type a list of validation errors. Will be empty if no errors were found during. Resets after each call to `validate()`.

validate(*document*, *schema=None*, *update=False*, *context=None*)
Validates a Python dictionary against a validation schema.

Parameters

- **document** – the dict to validate.
- **schema** – the validation schema. Defaults to `None`. If not provided here, the schema must have been provided at class instantiation.
- **update** – If `True` validation of required fields won't be performed.
- **context** – the document in which context validation should be performed. Defaults to `None`.

Returns True if validation succeeds, False otherwise. Check the `errors()` property for a list of validation errors.

Changed in version 0.4.0: Support for update mode.

validate_schema(*schema*)
Validates a schema against supported rules.

Parameters **schema** – the schema to be validated as a legal cerberus schema according to the rules of this Validator object.

New in version 0.7.1.

validate_update(*document*, *schema=None*, *context=None*)
Validates a Python dictionary against a validation schema. The difference with `validate()` is that the `required` rule will be ignored here.

Parameters

- **schema** – optional validation schema. Defaults to `None`. If not provided here, the schema must have been provided at class instantiation.
- **context** – the context in which the document should be validated. Defaults to `None`.

Returns True if validation succeeds, False otherwise. Check the `errors()` property for a list of validation errors.

Deprecated since version 0.4.0: Use `validate()` with `update=True` instead.

validated(**args*, ***kwargs*)
Wrapper around `Validator.validate` that returns the validated document or `None` if validation failed.

2.5.2 Exceptions

class cerberus.**SchemaError**
Raised when the validation schema is missing, has the wrong format or contains errors.

2.6 Frequently Asked Questions

2.6.1 Can I use Cerberus to validate objectis?

Yes. See Validating user objects with Cerberus.

2.7 Changelog

Here you can see the full list of changes between each Cerberus release.

2.7.1 Version 0.10

Not released yet.

- New: 'excludes' rule (calve). Addresses #132.
- New: Add testing with Docker (Frank Sachsenheim).
- New: '*of'-rules can be extended by concentating another rule (Frank Sachsenheim).
- New: Known, validated definition schemas are cached, thus validation run-time of schemas is reduced (Frank Sachsenheim).
- New: Implemented rules of a Validator-instance are accessible as 'validation_rules'-property (Frank Sachsenheim).
- New: 'rename'-rule renames a field to a given string during normalization (Frank Sachsenheim).
- New: 'rename_handler'-rule that takes an callable that renames unknown fields (Frank Sachsenheim).
- New: 'Validator.purge_unknown'-property and conditional purging of unknown fields (Frank Sachsenheim).
- New: 'Validator.trail'-property can be used to determine the relation of the currently validating document to the 'root_document' (Frank Sachsenheim).
- Change: The processed root-document of is now available as 'root_document'- property of the (child-)Validator (Frank Sachsenheim).
- Change: Removed 'context'-argument from 'validate'-method as this is set upon the creation of a child-validator (Frank Sachsenheim).
- Change: 'ValidationError'-exception renamed to 'DocumentError' (Frank Sachsenheim).

- Change: Consolidated all schema-related error-messages' names (Frank Sachsenheim).
- Change: Use a logger for deprecation-warnings if available (Frank Sachsenheim).
- Fix: 'coerce'-constraints are validated (Frank Sachsenheim).
- Fix: Unknown fields are normalized (Frank Sachsenheim).
- Fix: Dependency on boolean field now works as expected. Addresses #138 (Roman Redkovich)
- Fix: Add missing deprecation-warnings (Frank Sachsenheim).

2.7.2 Version 0.9.1

Released on July 7 2015

- Fix: 'required' is always evaluated, independent of eventual missing dependencies. This changes the previous behaviour whereas a required field with dependencies would only be reported as missing if all dependencies were met. A missing required field will always be reported. Also see the discussion in https://github.com/nicolaiarocci/eve/pull/665.

2.7.3 Version 0.9

Released on June 24 2015. Codename: 'Mastrolindo'.

- New: 'oneof' rule which provides a list of definitions in which only one should validate (C.D. Clark III).
- New: 'noneof' rule which provides a list of definitions that should all not validate (C.D. Clark III).
- New: 'anyof' rule accepts a list of definitions and checks that one definition validates (C.D. Clark III).
- New: 'allof' rule validates if if all definitions validate (C.D. Clark III).
- New: 'validator.validated' takes a document as argument and returns a validated document or 'None' if validation failed (Frank Sachsenheim).
- New: PyPy support (Frank Sachsenheim).
- New: Type coercion (Brett).
- New: Added 'propertyschema' validation rule (Frank Sachsenheim).
- Change: Use 'str.format' in error messages so if someone wants to override them does not get an exception if arguments are not passed. Closes #105 (Brett).
- Change: 'keyschema' renamed to 'valueschema', print a deprecation warning (Frank Sachsenheim).
- Change: 'type' can also be a list of types (Frank Sachsenheim).
- Fix: useages of 'document' to 'self.document' in '_validate' (Frank Sachsenheim).
- Fix: when 'items' is applied to a list, field name is used as key for 'validator.errors', and offending field indexes are used as keys for field errors ({'a_list_of_strings': {1: 'not a string'}}) 'type' can be a list of valid types.
- Fix: Ensure that additional **kwargs* of a subclass persist through validation (Frank Sachsenheim).
- Fix: improve failure message when testing against multiple types (Frank Sachsenheim).
- Fix: ignore 'keyschema' when not a mapping (Frank Sachsenheim).
- Fix: ignore 'schema' when not a sequence (Frank Sachsenheim).
- Fix: allow_unknown can also be set for nested dicts. Closes #75 (Tobias Betz).
- Fix: raise SchemaError when an unallowed 'type' is used in conjunction with 'schema' rule (Tobias Betz).

- Docs: added section that points out that YAML, JSON, etc. can be used to define schemas (C.D. Clark III).
- Docs: Improve 'allow_unknown' documentation (Frank Sachsenheim).

2.7.4 Version 0.8.1

Released on Mar 16 2015.

- Fix: dependency on a sub-document field does not work. Closes #64.
- Fix: readonly validation should happen before any other validation. Closes #63.
- Fix: allow_unknown does not apply to sub-dictionaries in a list. Closes #67.
- Fix: two tests being ignored because of name typo.
- Fix: update mode does not ignore required fields in subdocuments. Closes #72.
- Fix: allow_unknown does not respect custom rules. Closes #66.
- Fix: typo in docstrings (Riccardo).

2.7.5 Version 0.8

Released on Jan 7 2015.

- 'dependencies' also supports dependency values.
- 'allow_unknown' can also be set to a validation schema, in which case unknown fields will be validated against it. Closes nicolaiarocci/eve#405.
- New function-based custom validation mode (Luo Peng).
- Fields with empty definitions in schema were reported as non-existent. Now they are considered as valid, whatever their value is (Jaroslav Semančík).
- If dependencies are precised for a 'required' field, then the presence of the field is only validated if all dependencies are present (Trong Hieu HA).
- Documentation typo (Nikita Vlaznev #55).
- [CI] Add travis_retry to pip install in case of network issues (Helgi Þormar Þorbjörnsson #49)

2.7.6 Version 0.7.2

Released on Jun 19 2014.

- Successfully validate int as float type (Florian Rathgeber).

2.7.7 Version 0.7.1

Released on Jun 17 2014.

- Validation schemas are now validated up-front. When you pass a Schema to the Validator it will be validated against the supported ruleset (Paul Weaver). Closes #39.
- Custom validators also have access to a special 'self.document' variable that allows validation of a field to happen in context of the rest of the document (Josh Villbrandt).

- Validator options like 'allow_unknown' and 'ignore_none_values' are now taken into consideration when validating sub-dictionaries. Closes #40.

2.7.8 Version 0.7

Released on May 16 2014.

- Python 3.4 is now supported.
- tox support.
- Added 'dependencies' validation rule (Lujeni).
- Added 'keyschema' validation rule (Florian Rathgeber).
- Added 'regex' validation rule. Closes #29.
- Added 'set' as a core data type. Closes #31.
- Not-nullable fields are validated independetly of their type definition (Jaroslav Semančík).
- Python trove classifiers added to setup.py. Closes #32.
- 'min' and 'max' now apply to floats and numbers too. Closes #30.

2.7.9 Version 0.6

Released on February 10 2014

- Added 'number' data type, which validates against both float and integer values (Brandon Aubie).
- Added support for running tests with py.test
- Fix non-blocking problem introduced with 0.5 (Martin Ortbauer).
- Fix bug when _error() is calld twice for a field (Jaroslav Semančík).
- More precise error message in rule 'schema' validation (Jaroslav Semančík).
- Use 'allowed' field for integer just like for string (Peter Demin).

2.7.10 Version 0.5

Released on December 4 2013

- 'validator.errors' now returns a dictionary where keys are document fields and values are lists of validation errors for the field.
- Validator instances are now callable. Instead of *validated = validator.validate(document)* you can now choose to do 'validated = validator(document)' (Eelke Hermens).

2.7.11 Version 0.4.0

Released on September 24 2013.

- 'validate_update' is deprecated and will be removed with next release. Use 'validate' with 'update=True' instead. Closes #21.
- Fixed a minor encoding issue which made installing on Windows/Python3 impossible. Closes #19 (Arsh Singh).

- Fix documentation typo (Daniele Pizzolli).
- 'type' validation is always performed first (only exception being 'nullable'). On failure, subsequent rules on the same field are skipped. Closes #18.

2.7.12 Version 0.3.0

Released on July 9 2013.

- docstrings now conform to PEP8.
- *self.errors* returns an empty list if validate() has not been called.
- added validation for the 'float' data type.
- 'nullable' rule added to allow for null field values to be accepted in validations. This is different than required in that you can actively change a value to None instead of omitting or ignoring it. It is essentially the ignore_none_values, allowing for more fine grained control down to the field level (Kaleb Pomeroy).

2.7.13 Version 0.2.0

Released on April 18 2013.

- 'allow_unknown' option added.

2.7.14 Version 0.1.0

Released on March 15 2013. Codename: 'Claw'.

- entering beta phase.
- support for Python 3.
- pep8 and pyflakes fixes (Harro van der Klauw).
- removed superflous typecheck for empty validator (Harro van der Klauw).
- 'ignore_none_values' option to ignore None values when type checking (Harro van der Klauw).
- 'minlenght' and 'maxlength' now apply to lists as well (Harro van der Klauw).

2.7.15 Version 0.0.3

Released on January 29 2013

- when a list item fails, its offset is now returned along with the list name.
- 'transparent_schema_rules' option added.
- 'empty' rule for string fields.
- 'schema' rule on lists of arbitrary lenght (Martjin Vermaat).
- 'allowed' rule on strings (Martjin Vermaat).
- 'items' (dict) is now deprecated. Use the upgraded 'schema' rule instead.
- AUTHORS file added to sources.
- CHANGES file added to sources.

2.7.16 Version 0.0.2

Released on November 22 2012.

- Added support for addition and validation of custom data types.
- Several documentation improvements.

2.7.17 Version 0.0.1

Released on October 16 2012.

First public preview release.

2.8 Authors

Cerberus is written and maintained by Nicola Iarocci and various contributors:

2.8.1 Development Lead

- Nicola Iarocci <nicola@nicolaiarocci.com>

2.8.2 Patches and Suggestions

- Arsh Singh
- Brandon Aubie
- Brett
- C.D. Clark III
- Danielle Pizzolli
- Denis Carriere
- Eelke Hermens
- Florian Rathgeber
- Frank Sachsenheim
- Harro van der Klauw
- Jaroslav Semančík
- Kaleb Pomeroy
- Kirill Pavlov
- Lujeni
- Luo Peng
- Martijn Vermaat
- Martin Ortbauer
- Nikita Vlaznev

- Paul Weaver
- Peter Demin
- Riccardo
- Roman Redkovich
- Tobias Betz
- Trong Hieu HA
- calve

2.8.3 Add feature

- Sanhe Hu

2.9 Contact

If you've scoured the *prose* and *API documentation* and still can't find an answer to your question, below are various support resources that should help. We do request that you do at least skim the documentation before posting tickets or mailing list questions, however!

If you'd like to stay up to date on the community and development of Cerberus, there are several options:

2.9.1 Blog

New releases are usually announced on my Website.

2.9.2 Twitter

I often tweet about new features and releases of Cerberus. Follow @nicolaiarocci.

2.9.3 Mailing List

The mailing list is intended to be a low traffic resource for users, developers and contributors of both the Cerberus and Eve projects.

2.9.4 Bugs/ticket tracker

To file new bugs or search existing ones, you may visit Issues page. This does require a (free, easy to set up) Github account.

2.9.5 GitHub

Of course the best way to track the development of Cerberus is through the GitHub repo.

2.10 License

Cerberus is an open source project by Nicola Iarocci.

CHAPTER 3

Copyright Notice

Cerberus is an open source project by Nicola Iarocci. See the original LICENSE for more information.

C

E

S

V