Cerberus Documentation

Release 1.3.5

Nicola Iarocci

Aug 09, 2023

CONTENTS

> 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 powerful yet simple and lightweight data validation functionality out of the box and is designed to be easily extensible, allowing for custom validation. It has no dependencies and is thoroughly tested from Python 2.7 up to 3.8, PyPy and PyPy3.

CHAPTER

ONE

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

TWO

FUNDING CERBERUS

Cerberus is a *collaboratively funded project*. If you run a business and are using Cerberus in a revenue-generating product, it would make business sense to sponsor its development: it ensures the project that your product relies on stays healthy and actively maintained.

Individual users are also welcome to make either a recurring pledge or a one time donation if Cerberus has helped you in your work or personal projects. Every single sign-up makes a significant impact towards making Cerberus possible.

To join the backer ranks, check out Cerberus campaign on Patreon.

CHAPTER

THREE

TABLE OF CONTENTS

3.1 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. Please refer to one of the many established ways to work in project-specific virtual environments, i.e. the Virtual Environments and Packages section of the Pyton documentation.

3.1.1 Stable Version

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

```
$ pip install cerberus
```

3.1.2 Development Version

Obtain the source (either as source distribution from the PyPI, with `git` or other means that the Github platform provides) and use the following command in the source's root directory for an editable installation. Subsequent changes to the source code will affect its following execution without re-installation.

```
$ pip install -e .
```

3.2 Cerberus Usage

3.2.1 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.

Details about validation schemas are covered in *Validation Schemas*. See *Validation Rules* and *Normalization Rules* for an extensive documentation of all supported rules.

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* property to obtain a list of issues. See *Errors & Error Handling* for different output options.

```
>>> 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']}
```

A *DocumentError* is raised when the document is not a mapping.

The Validator class and its instances are callable, allowing for the following shorthand syntax:

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

New in version 0.4.1.

3.2.2 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 document keys 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
```

(continues on next page)

(continued from previous page)

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

`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': 'is allowed'}},
...            schema)
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.

3.2.3 Requiring all

By default any keys defined in the schema are not required. However, you can require all document keys pairs by setting `require_all` to `True` at validator initialization (`v = Validator(..., require_all=True)`) or change it latter via attribute access (`v.require_all = True`). `require_all` can also be set *as rule* to configure a validator for a subdocument that is checked against the *schema* rule:

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

>>> schema = {
...   'name': {'type': 'string'},
...   'a_dict': {
...     'type': 'dict',
...     'require_all': True,
...     'schema': {
...       'address': {'type': 'string'}
...     }
...   }
... }

>>> v.validate({'name': 'foo', 'a_dict': {}}, schema)
False
>>> v.errors
{'a_dict': [{'address': ['required field']}]}

>>> v.validate({'a_dict': {'address': 'foobar'}}, schema)
True
```

New in version 1.3.

3.2.4 Fetching Processed Documents

The normalization and coercion are performed on the copy of the original document and the result document is available via `document`-property.

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

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, unless you call the method with the keyword argument always_return_document set to True. 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 or method raises an exception then the exception will be caught and the validation with fail.

New in version 0.9.

normalized Method

Similarly, 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'])
<class 'int'>
```

New in version 1.0.

3.2.5 Warnings

Warnings, such as about deprecations or likely causes of trouble, are issued through the Python standard library's warnings module. The logging module can be configured to catch these logging.captureWarnings().

3.3 Validation Schemas

A validation schema is a mapping, usually a dict. 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 default all keys in a document are optional unless the *required*-rule is set True for individual fields or the validator's :attr:~cerberus.Validator.require_all is set to True in order to expect all schema-defined fields to be present in the document.

3.3.1 Registries

There are two default registries in the cerberus module namespace where you can store definitions for schemas and rules sets which then can be referenced in a validation schema. You can furthermore instantiate more `Registry` objects and bind them to the *rules_set_registry* or *schema_registry* of a validator. You may also set these as keyword-arguments upon intitialization.

Using registries is particularly interesting if

- schemas shall include references to themselves, vulgo: schema recursion
- schemas contain a lot of reused parts and are supposed to be *serialized*

```
>>> from cerberus import schema_registry
>>> schema_registry.add('non-system user',
...                     {'uid': {'min': 1000, 'max': 0xffff}})
>>> schema = {'sender': {'schema': 'non-system user',
...                      'allow_unknown': True},
...           'receiver': {'schema': 'non-system user',
...                        'allow_unknown': True}}
```

```
>>> from cerberus import rules_set_registry
>>> rules_set_registry.extend((('boolean', {'type': 'boolean'}),
...                            ('booleans', {'valuesrules': 'boolean'})))
>>> schema = {'foo': 'booleans'}
```

3.3.2 Validation

Validation schemas themselves are validated when passed to the validator or a new set of rules is set for a document's field. A *SchemaError* is raised when an invalid validation schema is encountered. See *Schema Validation Schema* for a reference.

However, be aware that no validation can be triggered for all changes below that level or when a used definition in a registry changes. You could therefore trigger a validation and catch the exception:

```
>>> v = Validator({'foo': {'allowed': []}})
>>> v.schema['foo'] = {'allowed': 1}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "cerberus/schema.py", line 99, in __setitem__
    self.validate({key: value})
  File "cerberus/schema.py", line 126, in validate
    self._validate(schema)
  File "cerberus/schema.py", line 141, in _validate
    raise SchemaError(self.schema_validator.errors)
SchemaError: {'foo': {'allowed': 'must be of container type'}}
>>> v.schema['foo']['allowed'] = 'strings are no valid constraint for allowed'
>>> v.schema.validate()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "cerberus/schema.py", line 126, in validate
    self._validate(schema)
  File "cerberus/schema.py", line 141, in _validate
```

(continues on next page)

(continued from previous page)

```
    raise SchemaError(self.schema_validator.errors)
SchemaError: {'foo': {'allowed': 'must be of container type'}}
```

3.3.3 Serialization

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.safe_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 favorite serializer. `json` for example. As long as there is a decoder that can produce a nested `dict`, you can use it to define a schema.

For populating and dumping one of the registries, use `extend()` and `all()`.

3.4 Validation Rules

3.4.1 allow_unknown

This can be used in conjunction with the *schema (dict)* rule when validating a mapping in order to set the *allow_unknown* property of the validator for the subdocument. This rule has precedence over `purge_unknown` (see *Purging Unknown Fields*). For a full elaboration refer to *this paragraph*.

3.4.2 allowed

This rule takes a `py3:collectionsabc.Container` of allowed values. Validates the target value if the value is in the allowed values. If the target value is an iterable, all its members must be in the allowed values.

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

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

(continues on next page)

(continued from previous page)

```
{'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.

3.4.3 allof

Validates if *all* of the provided constraints validates the field. See **of-rules* for details.

New in version 0.9.

3.4.4 anyof

Validates if *any* of the provided constraints validates the field. See **of-rules* for details.

New in version 0.9.

3.4.5 check_with

Validates the value of a field by calling either a function or method.

A function must be implemented like the following prototype:

```
def functionnname(field, value, error):
    if value is invalid:
        error(field, 'error message')
```

The `error` argument points to the calling validator's `_error` method. See *Extending Cerberus* on how to submit errors.

Here's an example that tests whether an integer is odd or not:

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

Then, you can validate a value like this:

```
>>> schema = {'amount': {'check_with': oddity}}
>>> v = Validator(schema)
>>> v.validate({'amount': 10})
False
>>> v.errors
{'amount': ['Must be an odd number']}

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

If the rule's constraint is a string, the *Validator* instance must have a method with that name prefixed by `_check_with_`. See *Methods that can be referenced by the check_with rule* for an equivalent to the function-based example above.

The constraint can also be a sequence of these that will be called consecutively.

```
schema = {'field': {'check_with': (oddity, 'prime number')}}
```

Changed in version 1.3: The rule was renamed from `validator` to `check_with`

3.4.6 contains

This rule validates that the a container object contains all of the defined items.

```
>>> document = {'states': ['peace', 'love', 'inity']}

>>> schema = {'states': {'contains': 'peace'}}
>>> v.validate(document, schema)
True

>>> schema = {'states': {'contains': 'greed'}}
>>> v.validate(document, schema)
False

>>> schema = {'states': {'contains': ['love', 'inity']}}
>>> v.validate(document, schema)
True

>>> schema = {'states': {'contains': ['love', 'respect']}}
>>> v.validate(document, schema)
False
```

3.4.7 dependencies

This rule allows one to define either a single field name, a sequence of field names or a mapping of field names and a sequence of allowed values as required in the document if the field defined upon is present in the document.

```
>>> 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 multiple field names are defined as dependencies, all of these must be present in order for the target field to be validated.

```
>>> schema = {'field1': {'required': False}, 'field2': {'required': False},
...           'field3': {'required': False, 'dependencies': ['field1', 'field2']}}
>>> document = {'field1': 7, 'field2': 11, 'field3': 13}
>>> v.validate(document, schema)
True

>>> document = {'field2': 11, 'field3': 13}
>>> v.validate(document, schema)
False

>>> v.errors
{'field3': ["field 'field1' is required"]}
```

When a mapping is provided, 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': ['one', 'two']}}}

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

>>> document = {'field1': 'three', 'field2': 7}
>>> v.validate(document, schema)
False
>>> v.errors
{'field2': ["depends on these values: {'field1': ['one', 'two']}"]}

>>> # same as using a dependencies list
>>> document = {'field2': 7}
>>> v.validate(document, schema)
False
```

(continues on next page)

(continued from previous page)

```
>>> v.errors
{'field2': ["depends on these values: {'field1': ['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': ["depends on these values: {'field1': 'one'}"]}
```

Declaring dependencies on subdocument fields with dot-notation is 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"]}
```

When a subdocument is processed the lookup for a field in question starts at the level of that document. In order to address the processed document as root level, the declaration has to start with a `^`. An occurrence of two initial carets (`^^`) is interpreted as a literal, single `^` with no special meaning.

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

(continues on next page)

(continued from previous page)

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

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

Note: If you want to extend semantics of the dot-notation, you can *override* the *`_lookup_field()`* method.

Note: The evaluation of this rule does not consider any constraints defined with the *required* rule.

Changed in version 1.0.2: Support for absolute addressing with `^`.

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.

3.4.8 empty

If constrained with `False` validation of an iterable value will fail if it is empty. Per default the emptiness of a field isn't checked and is therefore allowed when the rule isn't defined. But defining it with the constraint `True` will skip the possibly defined rules `allowed`, `forbidden`, `items`, `minlength`, `maxlength`, `regex` and `validator` for that field when the value is considered empty.

```
>>> 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.

3.4.9 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)
```

(continues on next page)

(continued from previous page)

```
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
```

3.4.10 forbidden

Opposite to *allowed* this validates if a value is any but one of the defined values:

```
>>> schema = {'user': {'forbidden': ['root', 'admin']}}
>>> document = {'user': 'root'}
>>> v.validate(document, schema)
False
```

New in version 1.0.

3.4.11 items

Validates the items of any iterable against a sequence of rules that must validate each index-correspondent item. The items will only be evaluated if the given iterable's size matches the definition's. This also applies during normalization and items of a value are not normalized when the lengths mismatch.

```
>>> 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 (list)* rule for dealing with arbitrary length `list` types.

3.4.12 keysrules

This rules takes a set of rules as constraint that all keys of a mapping are validated with.

```
>>> schema = {'a_dict': {
...                'type': 'dict',
...                'keysrules': {'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.

Changed in version 1.0: Renamed from `propertyschema` to `keyschema`

Changed in version 1.3: Renamed from `keyschema` to `keysrules`

3.4.13 meta

This is actually not a validation rule but a field in a rules set that can conventionally be used for application specific data that is descriptive for the document field:

```
{'id': {'type': 'string', 'regex': r'[A-M]\d{,6}',
        'meta': {'label': 'Inventory Nr.'}}}
```

The assigned data can be of any type.

New in version 1.3.

3.4.14 min, max

Minimum and maximum value allowed for any object whose class implements comparison operations (`__gt__` & `__lt__`).

```
>>> schema = {'weight': {'min': 10.1, 'max': 10.9}}
>>> document = {'weight': 10.3}
>>> v.validate(document, schema)
True

>>> document = {'weight': 12}
>>> v.validate(document, schema)
False

>>> v.errors
{'weight': ['max value is 10.9']}
```

Changed in version 1.0: Allows any type to be compared.

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

3.4.15 minlength, maxlength

Minimum and maximum length allowed for sized types that implement `__len__`.

```
>>> schema = {'numbers': {'minlength': 1, 'maxlength': 3}}
>>> document = {'numbers': [256, 2048, 23]}
>>> v.validate(document, schema)
True

>>> document = {'numbers': [256, 2048, 23, 2]}
>>> v.validate(document, schema)
False

>>> v.errors
{'numbers': ['max length is 3']}
```

3.4.16 noneof

Validates if *none* of the provided constraints validates the field. See **of-rules* for details.

New in version 0.9.

3.4.17 nullable

If `True` the field value is allowed to be `None`. The rule will be checked on every field, regardless it's defined or not. The rule's constraint defaults `False`.

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

>>> 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.

3.4.18 *of-rules

These rules allow you to define different 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`.

`allof`	Validates if *all* of the provided constraints validates the field.
`anyof`	Validates if *any* 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.

Note: *Normalization* cannot be used in the rule sets within the constraints of these rules.

Note: Before you employ these rules, you should have investigated other possible solutions for the problem at hand with and without Cerberus. Sometimes people tend to overcomplicate schemas with these rules.

For example, to verify that a field's value 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': ['no definitions validate',
           {'anyof definition 0': ['max value is 10'],
            'anyof definition 1': ['min value is 100']}]}
```

The `anyof` rule tests each rules set in the list. Hence, 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
```

New in version 0.9.

*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_regex': ['^ham', 'spam$']}}
# is equivalent to
{'foo': {'anyof': [{'regex': '^ham'}, {'regex': 'spam$'}]}}
# but is also equivalent to
# {'foo': {'regex': r'(^ham|spam$)'}}
```

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=True)
>>> invalid_employees_phones = []
>>> for employee in employees:
...     if not employee_vldtr.validate(employee):
...         invalid_employees_phones.append(employee)
```

3.4.19 oneof

Validates if *exactly one* of the provided constraints applies. See **of-rules* for details.

New in version 0.9.

3.4.20 readonly

If `True` the value is readonly. Validation will fail if this field is present in the target dictionary. This is useful, for example, when receiving a payload which is to be validated before it is sent to the datastore. The field might be provided by the datastore, but should not writable.

A validator can be configured with the initialization argument `purge_readonly` and the property with the same name to let it delete all fields that have this rule defined positively.

Changed in version 1.0.2: Can be used in conjunction with `default` and `default_setter`, see *Default Values*.

3.4.21 regex

The validation will fail if the field's value does not *match* the provided regular expression. It is only tested on string values.

```
>>> 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-.]+$
↪'"]}
```

A trailing `$` is ensured for all patterns in order to encourage users to write complete patterns for matching (and not a searching) strings. The implementation is inconsistent with regards to a leading `^`, these are not enforced. That inconsistency will not be fixed for the `1.3.x` release series. For details on regular expression syntax, see the documentation on the standard library's `re`-module.

Hint: Mind that one can set behavioural flags as part of the expression which is equivalent to passing `flags` to the `re.compile()` function for example. So, the constraint `'(?i)holy grail'` includes the equivalent of the `re.I` flag and matches any string that includes 'holy grail' or any variant of it with upper-case glyphs. Look for `(?aiLmsux)` in the mentioned library documentation for a description there.

New in version 0.7.

3.4.22 require_all

This can be used in conjunction with the *schema (dict)* rule when validating a mapping in order to set the *require_all* property of the validator for the subdocument. For a full elaboration refer to *this paragraph*.

3.4.23 required

If `True` the field 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: To define all fields of a document as required see *this section about the available options*.

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.

Note: The evaluation of this rule does not consider any constraints defined with the *dependencies* rule.

Changed in version 0.8: Check field dependencies.

3.4.24 schema (dict)

If a field for which a `schema`-rule is defined has a *mapping* as value, that mapping will be validated against the schema that is provided as constraint.

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

Note: To validate *arbitrary keys* of a mapping, see *keysrules-rule*, resp. *valuesrules-rule* for validating *arbitrary values* of a mapping.

3.4.25 schema (list)

If `schema`-validation encounters an arbitrary sized *sequence* as value, all items of the sequence will be validated against the rules provided in `schema`'s constraint.

```
>>> 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 preferred method for defining and validating a list of dictionaries.

Note: Using this rule should be accompanied with a `type`-rule explicitly restricting the field to the `list`-type like in the example. Otherwise false results can be expected when a mapping is validated against this rule with constraints for a sequence.

```
>>> schema = {'rows': {'type': 'list',
...                    'schema': {'type': 'dict', 'schema': {'sku': {'type': 'string'},
...                                                          'price': {'type': 'integer'}
↪}}}}
>>> 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

3.4.26 type

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

Type Name	Python 2 Type	Python 3 Type
boolean	bool	bool
binary	py2:bytes[1], bytearray	bytes, bytearray
date	datetime.date	datetime.date
datetime	datetime.datetime	datetime.datetime
dict	collections.Mapping	collections.abc.Mapping
float	float	float
integer	int, long	int
list	collections.Sequence, excl. string	collections.abc.Sequence, excl. string
number	float, int, long, excl. bool	float, int, excl. bool
set	set	set
string	basestring()	str

You can extend this list and support *custom types*.

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']}]}
```

Note: While the `type` rule is not required to be set at all, it is not encouraged to leave it unset especially when using more complex rules such as `schema`. If you decide you still don't want to set an explicit type, rules such as `schema` are only applied to values where the rules can actually be used (such as `dict` and `list`). Also, in the case of `schema`, cerberus will try to decide if a `list` or a `dict` type rule is more appropriate and infer it depending on what the `schema` rule looks like.

Note: Please note that type validation is performed before most others which exist for the same field (only *nullable* and *readonly* are considered beforehand). 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 one to safely assume that field type is correct when other (standard or custom) rules are invoked.

Changed in version 1.0: Added the `binary` data type.

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

[1] This is actually an alias of `str` in Python 2.

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.

3.4.27 valuesrules

This rules takes a set of rules as constraint that all values of a mapping are validated with.

```
>>> schema = {'numbers':
...               {'type': 'dict',
...                'valuesrules': {'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`

Changed in version 1.3: renamed `valueschema` to `valuesrules`

3.5 Normalization Rules

Normalization rules are applied to fields, also in `schema` for mappings, as well when defined as a bulk operation by `schema` (for sequences), `allow_unknown`, `keysrules` and `valuesrules`. Normalization rules in definitions for testing variants like with `anyof` are not processed.

The normalizations are applied as given in this document for each level in the mapping, traversing depth-first.

3.5.1 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. If the constraint is a string, it points to a *custom method*. If the constraint is an iterable, the value is processed through that chain.

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

```
>>> even_digits = lambda x: '0' + x if len(x) % 2 else x
>>> v = Validator({}, allow_unknown={'rename_handler': [str, even_digits]})
>>> v.normalized({1: 'foo'})
{'01': 'foo'}
```

New in version 1.0.

3.5.2 Purging Unknown Fields

After renaming, unknown fields will be purged if the *purge_unknown* property of a *Validator* instance is `True`; it defaults to `False`. You can set the property per keyword-argument upon initialization or as rule for subdocuments like `allow_unknown` (see *Allowing the Unknown*). The default is `False`. If a subdocument includes an `allow_unknown` rule then unknown fields will not be purged on that subdocument.

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

New in version 1.0.

3.5.3 Default Values

You can set default values for missing fields in the document by using the `default` rule.

```
>>> v.schema = {'amount': {'type': 'integer'}, 'kind': {'type': 'string', 'default':
↪'purchase'}}
>>> v.normalized({'amount': 1}) == {'amount': 1, 'kind': 'purchase'}
True

>>> v.normalized({'amount': 1, 'kind': None}) == {'amount': 1, 'kind': 'purchase'}
True

>>> v.normalized({'amount': 1, 'kind': 'other'}) == {'amount': 1, 'kind': 'other'}
True
```

You can also define a default setter callable to set the default value dynamically. The callable gets called with the current (sub)document as the only argument. Callables can even depend on one another, but normalizing will fail if there is a unresolvable/circular dependency. If the constraint is a string, it points to a *custom method*.

```
>>> v.schema = {'a': {'type': 'integer'}, 'b': {'type': 'integer', 'default_setter':␣
↪lambda doc: doc['a'] + 1}}
>>> v.normalized({'a': 1}) == {'a': 1, 'b': 2}
True

>>> v.schema = {'a': {'type': 'integer', 'default_setter': lambda doc: doc['not_there']}}
>>> v.normalized({})
>>> v.errors
{'a': ["default value for 'a' cannot be set: Circular dependencies of default setters."]}
```

You can even use both `default` and *readonly* on the same field. This will create a field that cannot be assigned a value manually but it will be automatically supplied with a default value by Cerberus. Of course the same applies for `default_setter`.

Changed in version 1.0.2: Can be used in conjunction with *readonly*.

New in version 1.0.

3.5.4 Value Coercion

Coercion allows you to apply a callable (given as object or the name of a *custom coercion method*) 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. If the constraint is an iterable of callables and names, the value is processed through that chain of coercers.

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

New in version 0.9.

3.6 Errors & Error Handling

Errors can be evaluated via Python interfaces or be processed to different output formats with error handlers.

3.6.1 Error Handlers

Error handlers will return different output via the *errors* property of a validator after the processing of a document. They base on *BaseErrorHandler* which defines the mandatory interface. The error handler to be used can be passed as keyword-argument `error_handler` to the initialization of a validator or by setting it's property with the same name at any time. On initialization either an instance or a class can be provided. To pass keyword-arguments to the initialization of a class, provide a two-value tuple with the error handler class and the dictionary containing the arguments.

The following handlers are available:

- *BasicErrorHandler*: This is the **default** that returns a dictionary. The keys refer to the document's ones and the values are lists containing error messages. Errors of nested fields are kept in a dictionary as last item of these lists.

3.6.2 Python interfaces

An error is represented as *ValidationError* that has the following properties:

- `document_path`: The path within the document. For flat dictionaries this simply is a key's name in a tuple, for nested ones it's all traversed key names. Items in sequences are represented by their index.
- `schema_path`: The path within the schema.
- `code`: The unique identifier for an error. See *Error Codes* for a list.
- `rule`: The rule that was evaluated when the error occurred.
- `constraint`: That rule's constraint.
- `value`: The value being validated.
- `info`: This tuple contains additional information that were submitted with the error. For most errors this is actually nothing. For bulk validations (e.g. with `items` or `keysrules`) this property keeps all individual errors. See the implementation of a rule in the source code to figure out its additional logging.

You can access the errors per these properties of a *Validator* instance after a processing of a document:

- `_errors`: This `ErrorsList` instance holds all submitted errors. It is not intended to manipulate errors directly via this attribute. You can test if at least one error with a specific error definition is `in` that list.
- `document_error_tree`: A `dict`-like object that allows one to query nodes corresponding to your document. The subscript notation on a node allows one to fetch either a specific error that matches the given *ErrorDefinition* or a child node with the given key. If there's no matching error respectively no errors occurred in a node or below, `None` will be returned instead. A node can also be tested with the `in` operator with either an *ErrorDefinition* or a possible child node's key. A node's errors are contained in its `errors` property which is also an `ErrorsList`. Its members are yielded when iterating over a node.
- `schema_error_tree`: Similarly for the used schema.

Changed in version 1.0: Errors are stored as *ValidationError* in a *ErrorList*.

Examples

```
>>> schema = {'cats': {'type': 'integer'}}
>>> document = {'cats': 'two'}
>>> v.validate(document, schema)
False
>>> cerberus.errors.BAD_TYPE in v._errors
True
>>> v.document_error_tree['cats'].errors == v.schema_error_tree['cats']['type'].errors
True
>>> cerberus.errors.BAD_TYPE in v.document_error_tree['cats']
True
>>> v.document_error_tree['cats'][cerberus.errors.BAD_TYPE] \
...     == v.document_error_tree['cats'].errors[0]
True
>>> error = v.document_error_tree['cats'].errors[0]
>>> error.document_path
('cats',)
>>> error.schema_path
('cats', 'type')
>>> error.rule
'type'
>>> error.constraint
'integer'
>>> error.value
'two'
```

3.7 Extending Cerberus

Though you can use functions in conjunction with the `coerce` and the `check_with` rules, you can easily extend the *`Validator`* class with custom `rules`, `types`, `check_with` handlers, `coercers` and `default_setters`. While the function-based style is more suitable for special and one-off uses, a custom class leverages these possibilities:

- custom rules can be defined with constrains in a schema
- extending the available *type* s
- use additional contextual data
- schemas are serializable

The references in schemas to these custom methods can use space characters instead of underscores, e.g. `{'foo': {'check_with': 'is odd'}}` is an alias for `{'foo': {'check_with': 'is_odd'}}`.

3.7.1 Custom Rules

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

```
schema = {'amount': {'is odd': True, 'type': 'integer'}}
```

This is how we would go to implement that:

```
from cerberus import Validator

class MyValidator(Validator):
    def _validate_is_odd(self, constraint, field, value):
        """ Test the oddity of a value.

        The rule's arguments are validated against this schema:
        {'type': 'boolean'}
        """
        if constraint is True and not bool(value & 1):
            self._error(field, "Must be an odd number")
```

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

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

As schemas themselves are validated, you can provide constraints as literal Python expression in the docstring of the rule's implementing method to validate the arguments given in a schema for that rule. Either the docstring contains solely the literal or the literal is placed at the bottom of the docstring preceded by `The rule's arguments are validated against this schema:` See the source of the contributed rules for more examples.

3.7.2 Custom Data Types

Cerberus supports and validates several standard data types (see *type*). When building a custom validator you can add and validate your own data types.

Additional types can be added on the fly by assigning a *TypeDefinition* to the designated type name in *types_mapping*:

```
from decimal import Decimal

decimal_type = cerberus.TypeDefinition('decimal', (Decimal,), ())

Validator.types_mapping['decimal'] = decimal_type
```

Caution: As the `types_mapping` property is a mutable type, any change to its items on an instance will affect its class.

They can also be defined for subclasses of *Validator*:

```
from decimal import Decimal

decimal_type = cerberus.TypeDefinition('decimal', (Decimal,), ())

class MyValidator(Validator):
    types_mapping = Validator.types_mapping.copy()
    types_mapping['decimal'] = decimal_type
```

New in version 0.0.2.

Changed in version 1.0: The type validation logic changed, see *Upgrading to Cerberus 1.0*.

Changed in version 1.2: Added the *types_mapping* property and marked methods for testing types as deprecated.

3.7.3 Methods that can be referenced by the check_with rule

If a validation test doesn't depend on a specified constraint from a schema or needs to be more complex than a rule should be, it's possible to rather define it as *value checker* than as a rule. There are two ways to use the *check_with rule*.

One is by extending *Validator* with a method prefixed with `_check_with_`. This allows to access the whole context of the validator instance including arbitrary configuration values and state. To reference such method using the `check_with` rule, simply pass the unprefixed method name as a string constraint.

For example, one can define an `oddity` validator method as follows:

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

Usage would look something like:

```
schema = {'amount': {'type': 'integer', 'check_with': 'oddity'}}
```

The second option to use the rule is to define a standalone function and pass it as the constraint. This brings with it the benefit of not having to extend `Validator`. To read more about this implementation and see examples check out the rule's documentation.

3.7.4 Custom Coercers

You can also define custom methods that return a `coerce` d value or point to a method as `rename_handler`. The method name must be prefixed with `_normalize_coerce_`.

```
class MyNormalizer(Validator):
    def __init__(self, multiplier, *args, **kwargs):
        super(MyNormalizer, self).__init__(*args, **kwargs)
        self.multiplier = multiplier
```

(continues on next page)

(continued from previous page)

```
    def _normalize_coerce_multiply(self, value):
        return value * self.multiplier
```

```
>>> schema = {'foo': {'coerce': 'multiply'}}
>>> document = {'foo': 2}
>>> MyNormalizer(multiplier=2).normalized(document, schema)
{'foo': 4}
```

3.7.5 Custom Default Setters

Similar to custom rename handlers, it is also possible to create custom default setters.

```
from datetime import datetime

class MyNormalizer(Validator):
    def _normalize_default_setter_utcnow(self, document):
        return datetime.utcnow()
```

```
>>> schema = {'creation_date': {'type': 'datetime', 'default_setter': 'utcnow'}}
>>> MyNormalizer().normalized({}, schema)
{'creation_date': datetime.datetime(...)}
```

3.7.6 Limitations

It may be a bad idea to overwrite particular contributed rules.

3.7.7 Attaching Configuration Data And Instantiating Custom Validators

It's possible to pass arbitrary configuration values when instantiating a *Validator* or a subclass as keyword arguments (whose names are not used by Cerberus). These can be used in all of the handlers described in this document that have access to the instance. Cerberus ensures that this data is available in all child instances that may get spawned during processing. When you implement an `__init__` method on a customized validator, you must ensure that all positional and keyword arguments are also passed to the parent class' initialization method. Here's an example pattern:

```
class MyValidator(Validator):
    def __init__(self, *args, **kwargs):
        # assign a configuration value to an instance property
        # for convenience
        self.additional_context = kwargs.get('additional_context')
        # pass all data to the base classes
        super(MyValidator, self).__init__(*args, **kwargs)

    # alternatively a dynamic property can be defined, rendering
    # the __init__ method unnecessary in this example case
    @property
    def additional_context(self):
        return self._config.get('additional_context', 'bar')
```

(continues on next page)

(continued from previous page)

```
    # an optional property setter if you deal with state
    @additional_context.setter
    def additional_context(self, value):
        self._config["additional_context"] = value

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

Warning: It is neither recommended to access the `_config` property in other situations than outlined in the sketch above nor to to change its contents during the processing of a document. Both cases are not tested and are unlikely to get officially supported.

New in version 0.9.

There's a function *validator_factory()* to get a `Validator` mutant with concatenated docstrings.

New in version 1.0.

3.7.8 Relevant *Validator*-attributes

There are some attributes of a *Validator* that you should be aware of when writing custom Validators.

Validator.document

A validator accesses the *document* property when fetching fields for validation. It also allows validation of a field to happen in context of the rest of the document.

New in version 0.7.1.

Validator.schema

Alike, the *schema* property holds the used schema.

Note: This attribute is not the same object that was passed as `schema` to the validator at some point. Also, its content may differ, though it still represents the initial constraints. It offers the same interface like a `dict`.

Validator._error

There are three signatures that are accepted to submit errors to the `Validator`'s error stash. If necessary the given information will be parsed into a new instance of *ValidationError*.

Full disclosure

In order to be able to gain complete insight into the context of an error at a later point, you need to call *_error()* with two mandatory arguments:

- the field where the error occurred
- an instance of a *ErrorDefinition*

For custom rules you need to define an error as `ErrorDefinition` with a unique id and the causing rule that is violated. See `errors` for a list of the contributed error definitions. Keep in mind that bit 7 marks a group error, bit 5 marks an error raised by a validation against different sets of rules.

Optionally you can submit further arguments as information. Error handlers that are targeted for humans will use these as positional arguments when formatting a message with `str.format()`. Serializing handlers will keep these values in a list.

New in version 1.0.

Simple custom errors

A simpler form is to call `_error()` with the field and a string as message. However the resulting error will contain no information about the violated constraint. This is supposed to maintain backward compatibility, but can also be used when an in-depth error handling isn't needed.

Multiple errors

When using child-validators, it is a convenience to submit all their errors ; which is a list of *ValidationError* instances.

New in version 1.0.

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. As the properties `document_path` and `schema_path` (see below) are inherited by the child validator, you can extend these by passing a single value or values-tuple with the keywords `document_crumb` and `schema_crumb`. Study the source code for example usages.

New in version 0.9.

Changed in version 1.0: Added `document_crumb` and `schema_crumb` as optional keyword- arguments.

Validator.root_document, .root_schema, .root_allow_unknown & .root_require_all

A child-validator - as used when validating a `schema` - can access the first generation validator's document and schema that are being processed as well as the constraints for unknown fields via its `root_document`, `root_schema`, `root_allow_unknown` and `root_require_all` properties.

New in version 1.0.

Changed in version 1.3: Added `root_require_all`

Validator.document_path & Validator.schema_path

These properties maintain the path of keys within the document respectively the schema that was traversed by possible parent-validators. Both will be used as base path when an error is submitted.

New in version 1.0.

Validator.recent_error

The last single error that was submitted is accessible through the `recent_error`-attribute.

New in version 1.0.

Validator.mandatory_validations, Validator.priority_validations & Validator._remaining_rules

You can use these class properties and instance instance property if you want to adjust the validation logic for each field validation. `mandatory_validations` is a tuple that contains rules that will be validated for each field, regardless if the rule is defined for a field in a schema or not. `priority_validations` is a tuple of ordered rules that will be validated before any other. `_remaining_rules` is a list that is populated under consideration of these and keeps track of the rules that are next in line to be evaluated. Thus it can be manipulated by rule handlers to change the remaining validation for the current field. Preferably you would call *`_drop_remaining_rules()`* to remove particular rules or all at once.

New in version 1.0.

Changed in version 1.2: Added `_remaining_rules` for extended leverage.

3.8 How to Contribute

There are no plans to develop Cerberus further than the current feature set. Bug fixes and documentation improvements are welcome and will be published with yearly service releases.

3.8.1 Making Changes

- Fork the repository on GitHub.
- Create a new topic branch from the `1.3.x` branch.
- Make commits of logical units (if needed rebase your feature branch before submitting it).
- Make sure your commit messages are in the proper format.
- If your commit fixes an open issue, reference it in the commit message.
- Make sure you have added the necessary tests for your changes.
- Run all the tests to assure nothing else was accidentally broken.
- Install and enable pre-commit (`pip install pre-commit`, then `pre-commit install`) to ensure styleguides and codechecks are followed.
- Don't forget to add yourself to the `AUTHORS.rst` document.

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

3.8.2 Submitting Changes

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

3.8.3 Running the Tests

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

```
$ pytest cerberus/tests
```

Testing with other Python versions

Before you submit a pull request, make sure your tests and changes run in all supported python versions. Instead of creating all those environments by hand, you can use tox that automatically manages virtual environments. Mind that the interpreters themselves need to be available on the system.

```
$ 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 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 3.11:

```
$ tox -e py311
```

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

Running the benchmarks

There's a benchmark suite that you can use to measure how changes imapact Cerberus' performance:

```
$ pytest cerberus/benchmarks
```

Building the HTML-documentation

To preview the rendered HTML-documentation you must initially install the documentation framework and a theme:

```
$ pip install -r docs/requirements.txt
```

The HTML build is triggered with:

```
$ make -C docs html
```

The result can be accessed by opening `docs/_build/html/index.html`.

3.9 Funding

We believe that collaboratively funded software can offer outstanding returns on investment, by encouraging users to collectively share the cost of development.

Cerberus continues to be open-source and permissively licensed, but we firmly believe it is in the commercial best-interest for users of the project to invest in its ongoing development.

Signing up as a Backer:

- Directly contribute to faster releases, more features, and higher quality software.
- Allow more time to be invested in documentation, issue triage, and community support.
- Safeguard the future development of Cerberus.

If you run a business and is using Cerberus in a revenue-generating product, it would make business sense to sponsor its development: it ensures the project that your product relies on stays healthy and actively maintained. It can also help your exposure in the Cerberus community and makes it easier to attract Cerberus developers.

Of course, individual users are also welcome to make a recurring pledge if Cerberus has helped you in your work or personal projects. Alternatively, consider donating as a sign of appreciation - like buying me coffee once in a while :)

3.9.1 Support Cerberus development

You can support Cerberus development by pledging on Patreon or donating on PayPal.

- Become a Backer (recurring pledge)
- Donate via PayPal (one time)

3.10 API Documentation

3.10.1 Validator Class

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

Validator class. Normalizes and/or validates any mapping against a validation-schema which is provided as an argument at class instantiation or upon calling the *validate()*, *validated()* or *normalized()* method. An instance itself is callable and executes a validation.

All instantiation parameters are optional.

There are the introspective properties `types`, `validators`, `coercers`, `default_setters`, `rules`, `normalization_rules` and `validation_rules`.

The attributes reflecting the available rules are assembled considering constraints that are defined in the docstrings of rules' methods and is effectively used as validation schema for *schema*.

Parameters

- **schema** (any mapping) – See *schema*. Defaults to `None`.
- **ignore_none_values** (`bool`) – See *ignore_none_values*. Defaults to `False`.
- **allow_unknown** (`bool` or any mapping) – See *allow_unknown*. Defaults to `False`.
- **require_all** (`bool`) – See *require_all*. Defaults to `False`.
- **purge_unknown** (`bool`) – See *purge_unknown*. Defaults to to `False`.

- **purge_readonly** (bool) – Removes all fields that are defined as `readonly` in the normalization phase.
- **error_handler** (class or instance based on *BaseErrorHandler* or tuple) – The error handler that formats the result of *errors*. When given as two-value tuple with an error-handler class and a dictionary, the latter is passed to the initialization of the error handler. Default: *BasicErrorHandler*.

_drop_remaining_rules(**rules*)

Drops rules from the queue of the rules that still need to be evaluated for the currently processed field. If no arguments are given, the whole queue is emptied.

_error(**args*)

Creates and adds one or multiple errors.

Parameters

args – Accepts different argument's signatures.

1. Bulk addition of errors:

- iterable of *ValidationError*-instances

The errors will be added to *_errors*.

2. Custom error:

- the invalid field's name
- the error message

A custom error containing the message will be created and added to *_errors*. There will however be fewer information contained in the error (no reference to the violated rule and its constraint).

3. Defined error:

- the invalid field's name
- the error-reference, see `cerberus.errors`
- arbitrary, supplemental information about the error

A *ValidationError* instance will be created and added to *_errors*.

_errors

The list of errors that were encountered since the last document processing was invoked. Type: *ErrorList*

_get_child_validator(*document_crumb=None*, *schema_crumb=None*, ***kwargs*)

Creates a new instance of Validator-(sub-)class. All initial parameters of the parent are passed to the initialization, unless a parameter is given as an explicit *keyword*-parameter.

Parameters

- **document_crumb** (tuple or hashable) – Extends the *document_path* of the child-validator.
- **schema_crumb** (tuple or hashable) – Extends the *schema_path* of the child-validator.
- **kwargs** (dict) – Overriding keyword-arguments for initialization.

Returns

an instance of `self.__class__`

_lookup_field(*path*)

Searches for a field as defined by path. This method is used by the `dependency` evaluation logic.

Parameters
path (str) – Path elements are separated by a `.`. A leading `^` indicates that the path relates to the document root, otherwise it relates to the currently evaluated document, which is possibly a subdocument. The sequence `^^` at the start will be interpreted as a literal `^`.

Returns
Either the found field name and its value or `None` for both.

Return type
A two-value `tuple`.

_remaining_rules

Keeps track of the rules that are next in line to be evaluated during the validation of a field. Type: `list`

_valid_schemas = {}

A `set` of hashes derived from validation schemas that are legit for a particular `Validator` class.

property allow_unknown

If `True` unknown fields that are not defined in the schema will be ignored. If a mapping with a validation schema is given, any undefined field will be validated against its rules. Also see *Allowing the Unknown*. Type: `bool` or any mapping

classmethod clear_caches()

Purge the cache of known valid schemas.

document

The document that is or was recently processed. Type: any mapping

document_error_tree

A tree representiation of encountered errors following the structure of the document. Type: *DocumentErrorTree*

document_path

The path within the document to the current sub-document. Type: `tuple`

error_handler

The error handler used to format *errors* and process submitted errors with *_error()*. Type: *BaseErrorHandler*

property errors

The errors of the last processing formatted by the handler that is bound to *error_handler*.

property ignore_none_values

Whether to not process `None`-values in a document or not. Type: `bool`

property is_child

`True` for child-validators obtained with *_get_child_validator()*. Type: `bool`

mandatory_validations = ('nullable',)

Rules that are evaluated on any field, regardless whether defined in the schema or not. Type: `tuple`

normalized(*document*, *schema=None*, *always_return_document=False*)

Returns the document normalized according to the specified rules of a schema.

Parameters

- **document** (any mapping) – The document to normalize.

- **schema** (any mapping) – The validation schema. Defaults to `None`. If not provided here, the schema must have been provided at class instantiation.
- **always_return_document** (`bool`) – Return the document, even if an error occurred. Defaults to: `False`.

Returns

A normalized copy of the provided mapping or `None` if an error occurred during normalization.

priority_validations = ('nullable', 'readonly', 'type', 'empty')

Rules that will be processed in that order before any other. Type: `tuple`

property purge_unknown

If `True`, unknown fields will be deleted from the document unless a validation is called with disabled normalization. Also see *Purging Unknown Fields*. Type: `bool`

recent_error

The last individual error that was submitted. Type: *ValidationError*

property require_all

If `True` known fields that are defined in the schema will be required. Type: `bool`

property root_allow_unknown

The *allow_unknown* attribute of the first level ancestor of a child validator.

property root_document

The *document* attribute of the first level ancestor of a child validator.

property root_require_all

The *require_all* attribute of the first level ancestor of a child validator.

property root_schema

The *schema* attribute of the first level ancestor of a child validator.

property rules_set_registry

The registry that holds referenced rules sets. Type: `Registry`

property schema

The validation schema of a validator. When a schema is passed to a method, it replaces this attribute. Type: any mapping or `None`

schema_error_tree

A tree representation of encountered errors following the structure of the schema. Type: *SchemaErrorTree*

schema_path

The path within the schema to the current sub-schema. Type: `tuple`

property schema_registry

The registry that holds referenced schemas. Type: `Registry`

```
types_mapping = {'binary': TypeDefinition(name='binary', included_types=(<class
'bytes'>, <class 'bytearray'>), excluded_types=()), 'boolean':
TypeDefinition(name='boolean', included_types=(<class 'bool'>,), excluded_types=()),
'container': TypeDefinition(name='container', included_types=(<class
'collections.abc.Container'>,), excluded_types=(<class 'str'>,)), 'date':
TypeDefinition(name='date', included_types=(<class 'datetime.date'>,),
excluded_types=()), 'datetime': TypeDefinition(name='datetime',
included_types=(<class 'datetime.datetime'>,), excluded_types=()), 'dict':
TypeDefinition(name='dict', included_types=(<class 'collections.abc.Mapping'>,),
excluded_types=()), 'float': TypeDefinition(name='float', included_types=(<class
'float'>, (<class 'int'>,)), excluded_types=()), 'integer':
TypeDefinition(name='integer', included_types=((<class 'int'>,),),
excluded_types=()), 'list': TypeDefinition(name='list', included_types=(<class
'collections.abc.Sequence'>,), excluded_types=(<class 'str'>,)), 'number':
TypeDefinition(name='number', included_types=((<class 'int'>,), <class 'float'>),
excluded_types=(<class 'bool'>,)), 'set': TypeDefinition(name='set',
included_types=(<class 'set'>,), excluded_types=()), 'string':
TypeDefinition(name='string', included_types=(<class 'str'>,), excluded_types=())}
```

This mapping holds all available constraints for the type rule and their assigned *TypeDefinition*.

validate(*document*, *schema=None*, *update=False*, *normalize=True*)

Normalizes and validates a mapping against a validation-schema of defined rules.

Parameters

- **document** (any mapping) – The document to normalize.
- **schema** (any mapping) – The validation schema. Defaults to `None`. If not provided here, the schema must have been provided at class instantiation.
- **update** (`bool`) – If `True`, required fields won't be checked.
- **normalize** (`bool`) – If `True`, normalize the document before validation.

Returns

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

Return type

`bool`

validated(**args*, ***kwargs*)

Wrapper around *validate()* that returns the normalized and validated document or `None` if validation failed.

3.10.2 Rules Set & Schema Registry

class cerberus.schema.**Registry**(*definitions={}*)

A registry to store and retrieve schemas and parts of it by a name that can be used in validation schemas.

Parameters

definitions (any mapping) – Optional, initial definitions.

add(*name*, *definition*)

Register a definition to the registry. Existing definitions are replaced silently.

Parameters

- **name** (str) – The name which can be used as reference in a validation schema.
- **definition** (any mapping) – The definition.

all()

Returns a dict with all registered definitions mapped to their name.

clear()

Purge all definitions in the registry.

extend(*definitions*)

Add several definitions at once. Existing definitions are replaced silently.

Parameters
definitions (a mapping or an iterable with two-value tuple s) – The names and definitions.

get(*name*, *default=None*)

Retrieve a definition from the registry.

Parameters

- **name** (str) – The reference that points to the definition.
- **default** – Return value if the reference isn't registered.

remove(**names*)

Unregister definitions from the registry.

Parameters
names – The names of the definitions that are to be unregistered.

3.10.3 Type Definitions

class cerberus.**TypeDefinition**(*name*, *included_types*, *excluded_types*)

This class is used to define types that can be used as value in the *types_mapping* property. The name should be descriptive and match the key it is going to be assigned to. A value that is validated against such definition must be an instance of any of the types contained in included_types and must not match any of the types contained in excluded_types.

3.10.4 Error Handlers

class cerberus.errors.**BaseErrorHandler**(**args*, ***kwargs*)

Base class for all error handlers. Subclasses are identified as error-handlers with an instance-test.

__call__(*errors*)

Returns errors in a handler-specific format.

Parameters
errors (iterable of *ValidationError* instances or a *Validator* instance) – An object containing the errors.

__init__(**args*, ***kwargs*)

Optionally initialize a new instance.

__iter__()

Be a superhero and implement an iterator over errors.

__weakref__

list of weak references to the object (if defined)

add(*error*)

Add an error to the errors' container object of a handler.

Parameters
error (*ValidationError*) – The error to add.

emit(*error*)

Optionally emits an error in the handler's format to a stream. Or light a LED, or even shut down a power plant.

Parameters
error (*ValidationError*) – The error to emit.

end(*validator*)

Gets called when a validation ends.

Parameters
validator (*Validator*) – The calling validator.

extend(*errors*)

Adds all errors to the handler's container object.

Parameters
errors (iterable of *ValidationError* instances) – The errors to add.

start(*validator*)

Gets called when a validation starts.

Parameters
validator (*Validator*) – The calling validator.

class cerberus.errors.**BasicErrorHandler**(*tree=None*)

Models cerberus' legacy. Returns a `dict`. When mangled through `str` a pretty-formatted representation of that tree is returned.

3.10.5 Python Error Representations

class cerberus.errors.**ErrorDefinition**(*code*, *rule*)

This class is used to define possible errors. Each distinguishable error is defined by a *unique* error `code` as integer and the `rule` that can cause it as string. The instances' names do not contain a common prefix as they are supposed to be referenced within the module namespace, e.g. `errors.CUSTOM`.

class cerberus.errors.**ValidationError**(*document_path*, *schema_path*, *code*, *rule*, *constraint*, *value*, *info*)

A simple class to store and query basic error information.

property child_errors

A list that contains the individual errors of a bulk validation error.

code

The error's identifier code. Type: `int`

constraint

The constraint that failed.

property definitions_errors

Dictionary with errors of an *of-rule mapped to the index of the definition it occurred in. Returns `None` if not applicable.

document_path

The path to the field within the document that caused the error. Type: `tuple`

property field

Field of the contextual mapping, possibly `None`.

info

May hold additional information about the error. Type: `tuple`

property is_group_error

`True` for errors of bulk validations.

property is_logic_error

`True` for validation errors against different schemas with *of-rules.

property is_normalization_error

`True` for normalization errors.

rule

The rule that failed. Type: *string*

schema_path

The path to the rule within the schema that caused the error. Type: `tuple`

value

The value that failed.

Error Codes

Its `code` attribute uniquely identifies an *`ErrorDefinition`* that is used a concrete error's *`code`*. Some codes are actually reserved to mark a shared property of different errors. These are useful as bitmasks while processing errors. This is the list of the reserved codes:

`0110 0000`	`0x60`	96	An error that occurred during normalization.
`1000 0000`	`0x80`	128	An error that contains child errors.
`1001 0000`	`0x90`	144	An error that was emitted by one of the *of-rules.

None of these bits in the upper nibble must be used to enumerate error definitions, but only to mark one with the associated property.

Important: Users are advised to set bit 8 for self-defined errors. So the code `0001 0000 0001` / `0x101` would the first in a domain-specific set of error definitions.

This is a list of all error defintions that are shipped with the `errors` module:

Code (dec.)	Code (hex.)	Name	Rule
0	0x0	CUSTOM	None

continues on next page

Table 1 – continued from previous page

Code (dec.)	Code (hex.)	Name	Rule
2	0x2	REQUIRED_FIELD	required
3	0x3	UNKNOWN_FIELD	None
4	0x4	DEPENDENCIES_FIELD	dependencies
5	0x5	DEPENDENCIES_FIELD_VALUE	dependencies
6	0x6	EXCLUDES_FIELD	excludes
34	0x22	EMPTY_NOT_ALLOWED	empty
35	0x23	NOT_NULLABLE	nullable
36	0x24	BAD_TYPE	type
37	0x25	BAD_TYPE_FOR_SCHEMA	schema
38	0x26	ITEMS_LENGTH	items
39	0x27	MIN_LENGTH	minlength
40	0x28	MAX_LENGTH	maxlength
65	0x41	REGEX_MISMATCH	regex
66	0x42	MIN_VALUE	min
67	0x43	MAX_VALUE	max
68	0x44	UNALLOWED_VALUE	allowed
69	0x45	UNALLOWED_VALUES	allowed
70	0x46	FORBIDDEN_VALUE	forbidden
71	0x47	FORBIDDEN_VALUES	forbidden
72	0x48	MISSING_MEMBERS	contains
96	0x60	NORMALIZATION	None
97	0x61	COERCION_FAILED	coerce
98	0x62	RENAMING_FAILED	rename_handler
99	0x63	READONLY_FIELD	readonly
100	0x64	SETTING_DEFAULT_FAILED	default_setter
128	0x80	ERROR_GROUP	None
129	0x81	MAPPING_SCHEMA	schema
130	0x82	SEQUENCE_SCHEMA	schema
131	0x83	KEYSRULES	keysrules
131	0x83	KEYSCHEMA	keysrules
132	0x84	VALUESRULES	valuesrules
132	0x84	VALUESCHEMA	valuesrules
143	0x8f	BAD_ITEMS	items
144	0x90	LOGICAL	None
145	0x91	NONEOF	noneof
146	0x92	ONEOF	oneof
147	0x93	ANYOF	anyof
148	0x94	ALLOF	allof

Error Containers

class cerberus.errors.**ErrorList**(*iterable=()*, */*)

A list for *ValidationError* instances that can be queried with the `in` keyword for a particular *ErrorDefinition*.

class cerberus.errors.**ErrorTree**(*errors=()*)

Base class for *DocumentErrorTree* and *SchemaErrorTree*.

add(*error*)

Add an error to the tree.

Parameters
error – *ValidationError*

fetch_errors_from(*path*)

Returns all errors for a particular path.

Parameters
path – tuple of hashable s.

Return type
ErrorList

fetch_node_from(*path*)

Returns a node for a path.

Parameters
path – Tuple of hashable s.

Return type
ErrorTreeNode or None

class cerberus.errors.**DocumentErrorTree**(*errors=()*)

Implements a dict-like class to query errors by indexes following the structure of a validated document.

class cerberus.errors.**SchemaErrorTree**(*errors=()*)

Implements a dict-like class to query errors by indexes following the structure of the used schema.

3.10.6 Exceptions

exception cerberus.**SchemaError**

Raised when the validation schema is missing, has the wrong format or contains errors.

exception cerberus.**DocumentError**

Raised when the target document is missing or has the wrong format

3.10.7 Utilities

class cerberus.utils.**TypeDefinition**(*name*, *included_types*, *excluded_types*)

This class is used to define types that can be used as value in the *types_mapping* property. The name should be descriptive and match the key it is going to be assigned to. A value that is validated against such definition must be an instance of any of the types contained in included_types and must not match any of the types contained in excluded_types.

excluded_types

Alias for field number 2

included_types

Alias for field number 1

name

Alias for field number 0

cerberus.utils.**mapping_to_frozenset**(*mapping*)

Be aware that this treats any sequence type with the equal members as equal. As it is used to identify equality of schemas, this can be considered okay as definitions are semantically equal regardless the container type.

class cerberus.utils.**readonly_classproperty**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

cerberus.utils.**validator_factory**(*name*, *bases=None*, *namespace={}*)

Dynamically create a *Validator* subclass. Docstrings of mixin-classes will be added to the resulting class' one if __doc__ is not in namespace.

Parameters

- **name** (str) – The name of the new class.
- **bases** (tuple of or a single class) – Class(es) with additional and overriding attributes.
- **namespace** (dict) – Attributes for the new class.

Returns

The created class.

3.10.8 Schema Validation Schema

Against this schema validation schemas given to a vanilla *Validator* will be validated:

```
{'allof': {'logical': 'allof', 'type': 'list'},
'allow_unknown': {'oneof': [{'type': 'boolean'},
                            {'check_with': 'bulk_schema',
                             'type': ['dict', 'string']}]},
'allowed': {'type': 'container'},
'anyof': {'logical': 'anyof', 'type': 'list'},
'check_with': {'oneof': [{'type': 'callable'},
                         {'schema': {'oneof': [{'type': 'callable'},
                                               {'allowed': (),
                                                'type': 'string'}]},
                          'type': 'list'},
                         {'allowed': (), 'type': 'string'}]},
'coerce': {'oneof': [{'type': 'callable'},
                     {'schema': {'oneof': [{'type': 'callable'},
                                           {'allowed': (),
                                            'type': 'string'}]},
                      'type': 'list'},
                     {'allowed': (), 'type': 'string'}]},
'contains': {'empty': False},
'default': {'nullable': True},
'default_setter': {'oneof': [{'type': 'callable'},
                             {'allowed': (), 'type': 'string'}]},
'dependencies': {'check_with': 'dependencies',
                 'type': ('dict', 'hashable', 'list')},
'empty': {'type': 'boolean'},
'excludes': {'schema': {'type': 'hashable'},
             'type': ('hashable', 'list')},
'forbidden': {'type': 'list'},
'items': {'check_with': 'items', 'type': 'list'},
'keysrules': {'check_with': 'bulk_schema',
              'forbidden': ['rename', 'rename_handler'],
              'type': ['dict', 'string']},
'max': {'nullable': False},
'maxlength': {'type': 'integer'},
```

(continues on next page)

(continued from previous page)

```
'meta': {},
'min': {'nullable': False},
'minlength': {'type': 'integer'},
'noneof': {'logical': 'noneof', 'type': 'list'},
'nullable': {'type': 'boolean'},
'oneof': {'logical': 'oneof', 'type': 'list'},
'purge_unknown': {'type': 'boolean'},
'readonly': {'type': 'boolean'},
'regex': {'type': 'string'},
'rename': {'type': 'hashable'},
'rename_handler': {'oneof': [{'type': 'callable'},
                             {'schema': {'oneof': [{'type': 'callable'},
                                                   {'allowed': (),
                                                    'type': 'string'}]},
                              'type': 'list'},
                             {'allowed': (), 'type': 'string'}]},
'require_all': {'type': 'boolean'},
'required': {'type': 'boolean'},
'schema': {'anyof': [{'check_with': 'schema'},
                     {'check_with': 'bulk_schema'}],
           'type': ['dict', 'string']},
'type': {'check_with': 'type', 'type': ['string', 'list']},
'valuesrules': {'check_with': 'bulk_schema',
                'forbidden': ['rename', 'rename_handler'],
                'type': ['dict', 'string']}}
```

3.11 Frequently Asked Questions

3.11.1 Can I use Cerberus to validate objects?

Yes. See Validating user objects with Cerberus.

3.11.2 Are Cerberus validators thread-safe, can they be used in different threads?

The normalization and validation methods of validators make a copy of the provided document and store it as *document* property. Because of this it is advised to create a new *Validator* instance for each processed document when used in a multi-threaded context. Alternatively you can use a `threading.Lock` to confirm that only one document processing is running at any given time.

3.12 External resources

Here are some recommended resources that deal with Cerberus. If you find something interesting on the web, please amend it to this document and open a pull request (see *How to Contribute*).

3.12.1 Community forums

There's a cerberus tag on the Question & Answers platform *Stackoverflow*. The Google Group regarding the mother project *Eve* may also a spot to seek these.

3.12.2 7 Best Python Libraries For Validating Data (February 2018)

Clickbait that mentions Cerberus. It's a starting point to compare libraries with a similar scope though.

3.12.3 Nicola Iarocci: Cerberus, or Data Validation for Humans (November 2017)

Get fastened for the full tour on Cerberus that Nicola gave in a talk at PiterPy 2017. No bit is missed, so don't miss it! The talk also includes a sample of the actual pronunciation of Iarocci as extra takeaway.

3.12.4 Henry Ölsner: Validate JSON data using cerberus (March 2016)

In this blog post the author describes how to validate network configurations with a schema noted in YAML. The article that doesn't spare on code snippets develops the resulting schema by gradually increasing its complexity. A custom type check is also implemented, but be aware that version *0.9.2* is used. With 1.0 and later the implementation should look like this:

```
def _validate_type_ipv4address(self, value):
    try:
        ipaddress.IPv4Address(value)
    except:
        return False
    else:
        return True
```

3.13 Cerberus Changelog

Cerberus is a collaboratively funded project, see the funding page.

3.13.1 Version 1.3.5

Released on August 9, 2023.

New

- Support for Python 3.10 & 3.11
- The HTML documentation uses the *furo* theme

Fixed

- `*of` rules are skipped for `None` values (#582)
- Validations of mappings would raise an exception when the field's rules were provided as reference to a registry item (#599)

Improved

- Various minor improvements of the documentation

3.13.2 Version 1.3.4

Released on May 5, 2021.

Fixed

- Reverts the unsatisfying fix for #557,
- instead a `RuntimeError` is thrown when Python is running with optimization level 2 (#567)

3.13.3 Version 1.3.3

Released on April 11, 2021.

New

- Adds a benchmark to observe overall performance between code changes (#531)
- Adds support for Python 3.9
- The Continuous Integration now runs on GitHub Actions

Fixed

- Fixed unresolved registry references when getting a constraint for an error (#562)
- Fixed crash when submitting non-hashable values to `allowed` (#524)
- Fixed schema validation for rules specifications with space (#527)
- Replaced deprecated rule name `validator` with `check_with` in the docs (#527)
- Use the UnconcernedValidator when the Python interpreter is executed with an optimization flag (#557)
- Several fixes and refinements of the docs

3.13.4 Version 1.3.2

Released on October 29, 2019.

New

- Support for Python 3.8

Fixed

- Fixed the message of the `BasicErrorHandler` for an invalid amount of items (#505)
- Added `setuptools` as dependency to the package metadata (#499)
- The `CHANGES.rst` document is properly included in the package (#493)

Improved

- Docs: Examples were added for the `min`- and `maxlength` rules. (#509)

3.13.5 Version 1.3.1

Releases on May 10, 2019.

Fixed

- Fixed the expansion of the deprecated rule names `keyschema` and `valueschema` (#482)
- `*of_`-typesavers properly expand rule names containing `_` (#484)

Improved

- Add `maintainer` and `maintainer_email` to setup.py (#481)
- Add `project_urls` to setup.py (#480)
- Don't ignore all exceptions during coercions for nullable fields. If a
- Coercion raises an exception for a nullable field where the field is not `None` the validation now fails. (#490)

3.13.6 Version 1.3

Releases on April 30, 2019.

New

- Add `require_all` rule and validator argument (#417)
- The `contains` rule (#358)
- All fields that are defined as `readonly` are removed from a document when a validator has the `purge_readonly` flag set to `True` (#240)
- The `validator` rule is renamed to `check_with`. The old name is deprecated and will not be available in the next major release of Cerberus (#405)
- The rules `keyschema` and `valueschema` are renamed to `keysrules` and `valuesrules`; the old names are deprecated and will not be available in the next major release of Cerbers (#385)
- The `meta` pseudo-rule can be used to store arbitrary application data related to a field in a schema
- Python 3.7 officially supported (#451)
- **Python 2.6 and 3.3 are no longer supported**

Fixed

- Fix test test_{default,default_setter}_none_nonnullable (#435)
- Normalization rules defined within the `items` rule are applied (#361)
- Defaults are applied to undefined fields from an `allow_unknown` definition (#310)
- The `forbidden` value now handles any input type (#449)
- The *allowed* rule will not be evaluated on fields that have a legit `None` value (#454)
- If the cerberus distribution cannot not be found, the version is set to the value `unknown` (#472)

Improved

- Suppress DeprecationWarning about collections.abc (#451)
- Omit warning when no schema for `meta` rule constraint is available (#425)
- Add `.eggs` to .gitignore file (#420)
- Reformat code to match Black code-style (#402)
- Perform lint checks and fixes on staged files, as a pre-commit hook (#402)
- Change `allowed` rule to use containers instead of lists (#384)
- Remove `Registry` from top level namespace (#354)
- Remove `utils.is_class`
- Check the `empty` rule against values of type `Sized`
- Various micro optimizations and 'safety belts' that were inspired by adding type annotations to a branch of the code base

Docs

- Fix semantical versioning naming. There are only two hard things in Computer Science: cache invalidation and naming things – *Phil Karlton* (#429)
- Improve documentation of the regex rule (#389)
- Expand upon *validator* rules (#320)
- Include all errors definitions in API docs (#404)
- Improve changelog format (#406)
- Update homepage URL in package metadata (#382)
- Add feature freeze note to CONTRIBUTING and note on Python support in README
- Add the intent of a `dataclasses` module to ROADMAP.md
- Update README link; make it point to the new PyPI website
- Update README with elaborations on versioning and testing
- Fix misspellings and missing pronouns
- Remove redundant hint from `*of-rules`.
- Add usage recommendation regarding the `*of-rules`
- Add a few clarifications to the GitHub issue template
- Update README link; make it point to the new PyPI website

3.13.7 Version 1.2

Released on April 12, 2018.

- New: docs: Add note that normalization cannot be applied within an `*of-rule`. (Frank Sachsenheim)
- New: Add the ability to query for a type of error in an error tree. (Frank Sachsenheim)
- New: Add errors.MAPPING_SCHEMA on errors within subdocuments. (Frank Sachsenheim)
- New: Support for Types Definitions, which allow quick types check on the fly. (Frank Sachsenheim)
- Fix: Simplify the tests with Docker by using a volume for tox environments. (Frank Sachsenheim)
- Fix: Schema registries do not work on dict fields. Closes #318. (Frank Sachsenheim)
- Fix: Need to drop some rules when `empty` is allowed. Closes #326. (Frank Sachsenheim)
- Fix: typo in README (Christian Hogan)
- Fix: Make `purge_unknown` and `allow_unknown` play nice together. Closes #324. (Audric Schiltknecht)
- Fix: API reference lacks generated content. Closes #281. (Frank Sachsenheim)
- Fix: `readonly` works properly just in the first validation. Closes #311. (Frank Sachsenheim)
- Fix: `coerce` ignores `nullable: True`. Closes #269. (Frank Sachsenheim)
- Fix: A dependency is not considered satisfied if it has a null value. Closes #305. (Frank Sachsenheim)
- Override `UnvalidatedSchema.copy`. (Peter Demin)
- Fix: README link. (Gabriel Wainer)
- Fix: Regression: allow_unknown causes dictionary validation to fail with a KeyError. Closes #302. (Frank Sachsenheim)
- Fix: Error when setting fields as tuples instead of lists. Closes #271. (Sebastian Rajo)
- Fix: Correctly handle nested logic and group errors. Closes #278 and #299. (Kornelijus Survila)
- CI: Reactivate testing on PyPy3. (Frank Sachsenheim)

3.13.8 Version 1.1

Released on January 25, 2017.

- New: Python 3.6 support. (Frank Sachsenheim)
- New: Users can implement their own semantics in Validator._lookup_field. (Frank Sachsenheim)
- New: Allow applying of `empty` rule to sequences and mappings. Closes #270. (Frank Sachsenheim)
- Fix: Better handling of unicode in `allowed` rule. Closes #280. (Michael Klich).
- Fix: Rules sets with normalization rules fail. Closes #283. (Frank Sachsenheim)
- Fix: Spelling error in RULE_SCHEMA_SEPARATOR constant (Antoine Lubineau)
- Fix: Expand schemas and rules sets when added to a registry. Closes #284 (Frank Sachsenheim)
- Fix: `readonly` conflicts with `default` rule. Closes #268 (Dominik Kellner).
- Fix: Creating custom Validator instance with `_validator_*` method raises `SchemaError`. Closes #265 (Frank Sachsenheim).
- Fix: Consistently use new style classes (Dominik Kellner).

- Fix: `NotImplemented` does not derive from `BaseException`. (Bryan W. Weber).
- Completely switch to py.test. Closes #213 (Frank Sachsenheim).
- Convert `self.assert` method calls to plain `assert` calls supported by pytest. Addresses #213 (Bruno Oliveira).
- Docs: Clarifications concerning dependencies and unique rules. (Frank Sachsenheim)
- Docs: Fix custom coerces documentation. Closes #285. (gilbsgilbs)
- Docs: Add note concerning regex flags. Closes #173. (Frank Sachsenheim)
- Docs: Explain that normalization and coercion are performed on a copy of the original document (Sergey Leshchenko)

3.13.9 Version 1.0.1

Released on September 1, 2016.

- Fix: bump trove classifier to Production/Stable (5).

3.13.10 Version 1.0

Released on September 1, 2016.

Warning: This is a major release which breaks backward compatibility in several ways. Don't worry, these changes are for the better. However, if you are upgrading, then you should really take the time to read the list of *Breaking Changes* and consider their impact on your codebase. For your convenience, some *upgrade notes* have been included.

- New: Add capability to use references in schemas. (Frank Sachsenheim)
- New: Support for binary type. (Matthew Ellison)
- New: Allow callables for 'default' schema rule. (Dominik Kellner)
- New: Support arbitrary types with 'max' and 'min' (Frank Sachsenheim).
- New: Support any iterable with 'minlength' and 'maxlength'. Closes #158. (Frank Sachsenheim)
- New: 'default' normalization rule. Closes #131. (Damián Nohales)
- New: 'excludes' rule (calve). Addresses #132.
- New: 'forbidden' rule. (Frank Sachsenheim)
- New: 'rename'-rule renames a field to a given value 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: 'coerce', 'rename_handler' and 'validator' can use class-methods (Frank Sachsenheim).
- New: '*of'-rules can be extended by concatenating another rule. (Frank Sachsenheim)
- New: Allows various error output with error handlers (Frank Sachsenheim).
- New: Available rules etc. of a Validator-instance are accessible as 'validation_rules', 'normalization_rules', 'types', 'validators' and 'coercer' -property. (Frank Sachsenheim)

- New: Custom rule's method docstrings can contain an expression to validate constraints for that rule when a schema is validated. (Frank Sachsenheim).
- New: 'Validator.root_schema' complements 'Validator.root_document'. (Frank Sachsenheim)
- New: 'Validator.document_path' and 'Validator.schema_path' properties can be used to determine the relation of the currently validating document to the 'root_document' / 'root_schema'. (Frank Sachsenheim)
- New: Known, validated definition schemas are cached, thus validation run-time of schemas is reduced. (Frank Sachsenheim)
- New: Add testing with Docker. (Frank Sachsenheim)
- New: Support CPython 3.5. (Frank Sachsenheim)
- Fix: 'allow_unknown' inside *of rule is ignored. Closes #251. (Davis Kirkendall)
- Fix: unexpected TypeError when using allow_unknown in a schema defining a list of dicts. Closes #250. (Davis Kirkendall)
- Fix: validate with 'update=True' does not work when required fields are in a list of subdicts. (Jonathan Huot)
- Fix: 'number' type fails if value is boolean. Closes #144. (Frank Sachsenheim)
- Fix: allow None in 'default' normalization rule. (Damián Nohales)
- Fix: in 0.9.2, coerce does not maintain proper nesting on dict fields. Closes #185.
- Fix: normalization not working for valueschema and propertyschema. Closes #155. (Frank Sachsenheim)
- Fix: 'coerce' on List elements produces unexpected results. Closes #161. (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)
- Docs: clarify read-only rule. Closes #127.
- Docs: split Usage page into Usage; Validation Rules: Normalization Rules. (Frank Sachsenheim)

Breaking Changes

Several relevant breaking changes have been introduced with this release. For the inside scoop, please see the *upgrade notes*.

- Change: 'errors' values are lists containing error messages. Previously, they were simple strings if single errors, lists otherwise. Closes #210. (Frank Sachsenheim)
- Change: Custom validator methods: remove the second argument. (Frank Sachsenheim)
- Change: Custom validator methods: invert the logic of the conditional clauses where is tested what a value is not / has not. (Frank Sachsenheim)
- Change: Custom validator methods: replace calls to 'self._error' with 'return True', or False, or None. (Frank Sachsenheim)
- Change: Remove 'transparent_schema_rule' in favor of docstring schema validation. (Frank Sachsenheim)
- Change: Rename 'property_schema' rule to 'keyschema'. (Frank Sachsenheim)
- Change: Replace 'validate_update' method with 'update' keywork argument. (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 warnings.warn for deprecation-warnings if available. (Frank Sachsenheim)

3.13.11 Version 0.9.2

Released on September 23, 2015

- Fix: don't rely on deepcopy since it can't properly handle complex objects in Python 2.6.

3.13.12 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/pyeve/eve/pull/665.

3.13.13 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).

3.13.14 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).

3.13.15 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 pyeve/eve:issue:*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)

3.13.16 Version 0.7.2

Released on Jun 19 2014.

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

3.13.17 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.

3.13.18 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.

3.13.19 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).

3.13.20 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).

3.13.21 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.

3.13.22 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).

3.13.23 Version 0.2.0

Released on April 18 2013.

- 'allow_unknown' option added.

3.13.24 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).

3.13.25 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.

3.13.26 Version 0.0.2

Released on November 22 2012.

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

3.13.27 Version 0.0.1

Released on October 16 2012.

First public preview release.

3.14 Upgrading to Cerberus 1.0

3.14.1 Major Additions

Error Handling

The inspection on and representation of errors is thoroughly overhauled and allows a more detailed and flexible handling. Make sure you have look on *Errors & Error Handling*.

Also, `errors` (as provided by the default `BasicErrorHandler`) values are lists containing error messages, and possibly a `dict` as last item containing nested errors. Previously, they were strings if single errors per field occurred; lists otherwise.

3.14.2 Deprecations

`Validator` class

transparent_schema_rules

In the past you could override the schema validation by setting `transparent_schema_rules` to `True`. Now all rules whose implementing method's docstring contain a schema to validate the arguments for that rule in the validation schema, are validated. To omit the schema validation for a particular rule, just omit that definition, but consider it a bad practice. The *`Validator`*-attribute and -initialization-argument `transparent_schema_rules` are removed without replacement.

validate_update

The method `validate_update` has been removed from *`Validator`*. Instead use *`validate()`* with the keyword-argument `update` set to `True`.

Rules

items (for mappings)

The usage of the `items`-rule is restricted to sequences. If you still had schemas that used that rule to validate mappings, just rename these instances to `schema` (*docs*).

keyschema & valueschema

To reflect the common terms in the Pythoniverse[1], the rule for validating all *values* of a mapping was renamed from `keyschema` to `valueschema`. Furthermore a rule was implemented to validate all *keys*, introduced as `propertyschema`, now renamed to `keyschema`. This means code using prior versions of cerberus would not break, but bring up wrong results!

To update your code you may adapt cerberus' iteration:

1. Rename `keyschema` to `valueschema` in your schemas. (`0.9`)
2. Rename `propertyschema` to `keyschema` in your schemas. (`1.0`)

Note that `propertyschema` will *not* be handled as an alias like
`keyschema` was in the `0.9`-branch.

Custom validators

Data types

Since the `type`-rule allowed multiple arguments cerberus' type validation code was somewhat cumbersome as it had to deal with the circumstance that each type checking method would file an error though another one may not - and thus positively validate the constraint as a whole. The refactoring of the error handling allows cerberus' type validation to be much more lightweight and to formulate the corresponding methods in a simpler way.

[1] compare dictionary

Previously such a method would test what a value *is not* and submit an error. Now a method tests what a value *is* to be expected and returns `True` in that case.

This is the most critical part of updating your code, but still easy when your head is clear. Of course your code is well tested. It's essentially these three steps. Search, Replace and Regex may come at your service.

1. Remove the second method's argument (probably named `field`).
2. Invert the logic of the conditional clauses where is tested what a value is not / has not.
3. Replace calls to `self._error` below such clauses with `return True`.

A method doesn't need to return `False` or any value when expected criteria are not met.

Here's the change from the *documentation* example.

pre-1.0:

```
def _validate_type_objectid(self, field, value):
    if not re.match('[a-f0-9]{24}', value):
        self._error(field, errors.BAD_TYPE)
```

1.0:

```
def _validate_type_objectid(self, value):
    if re.match('[a-f0-9]{24}', value):
        return True
```

3.15 Authors

Cerberus is developed and maintained by the Cerberus community. It was created by Nicola Iarocci.

3.15.1 Core maintainers

- Nicola Iarocci (nicolaiarocci)
- Frank Sachsenheim (funkyfuture)

3.15.2 Contributors

- Antoine Lubineau
- Arsh Singh
- Audric Schiltknecht
- Brandon Aubie
- Brett
- Bruno Oliveira
- Bryan W. Weber
- C.D. Clark III
- Christian Hogan
- Connor Zapfel

- Damián Nohales
- Danielle Pizzolli
- Davis Kirkendall
- Denis Carriere
- Dominik Kellner
- Eelke Hermens
- Evgeny Odegov
- Florian Rathgeber
- Gabriel Wainer
- Harro van der Klauw
- Jaroslav Semančík
- Jonathan Huot
- Kaleb Pomeroy
- Kirill Pavlov
- Kornelijus Survila
- Lujeni
- Luke Bechtel
- Luo Peng
- Martijn Vermaat
- Martin Ortbauer
- Matthew Ellison
- Michael Klich
- Nik Haldimann
- Nikita Melentev
- Nikita Vlaznev
- Paul Weaver
- Peter Demin
- Riccardo
- Roman Redkovich
- Scott Crunkleton
- Sebastian Heid
- Sebastian Rajo
- Sergey Leshchenko
- Tobias Betz
- Trong Hieu HA
- Vipul Gupta

- Waldir Pimenta
- Yauhen Shulitski
- calve
- gilbsgilbs

A full, up-to-date list of contributors is available from git with:

    git shortlog -sne

3.16 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:

3.16.1 Blog

New releases are usually announced on my Website.

3.16.2 Twitter

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

3.16.3 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.

3.16.4 Issues tracker

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

3.16.5 GitHub repository

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

3.17 License

Cerberus is an open source project by Nicola Iarocci.

ISC License

Copyright (c) 2012-2016 Nicola Iarocci.

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

CHAPTER

FOUR

COPYRIGHT NOTICE

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

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

I

M

N

P

R

S

T

V