Cerberus Documentation

Release 2.0.dev0

Nicola Iarocci

May 06, 2021
## Contents

1. **At a Glance** 3

2. **Funding Cerberus** 5

3. **Table of Contents** 7
   - 3.1 Cerberus Installation 7
   - 3.2 Cerberus Usage 8
   - 3.3 Validation Schemas 11
   - 3.4 Validation Rules 13
   - 3.5 Normalization Rules 28
   - 3.6 Errors & Error Handling 30
   - 3.7 Extending Cerberus 31
   - 3.8 How to Contribute 37
   - 3.9 Funding 40
   - 3.10 API Documentation 41
   - 3.11 Frequently Asked Questions 49
   - 3.12 External resources 49
   - 3.13 Cerberus Changelog 50
   - 3.14 Upgrading to Cerberus 2.0 from any 1.x version 61
   - 3.15 Upgrading to Cerberus 1.0 from prior versions 62
   - 3.16 Authors 63
   - 3.17 Contact 65
   - 3.18 License 66

4. **Copyright Notice** 67

Index 69
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.
You define a validation schema and pass it to an instance of the `Validator` class:

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

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

```python
>>> document = {'name': 'john doe'}
>>> v.validate(document)
True
```
Cerberus is a \textit{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.
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.

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

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

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

```
$ git clone git@github.com:pyeve/cerberus.git
Initialized empty Git repository in ~/dev/cerberus.git/
$ cd cerberus
$ virtualenv venv --distribute
New python executable in venv/bin/python
Installing distribute..........done.
$ . venv/bin/activate
$ python setup.py install
...
Finished processing dependencies for Cerberus
```
This will pull in the dependencies and activate the git head as the current version inside the virtualenv. Then all you have to do is run `git pull origin` to update to the latest version.

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

```
$ mkdir cerberus
$ cd cerberus
$ virtualenv venv --distribute
$ . venv/bin/activate
New python executable in venv/bin/python
Installing distribute............done.
$ pip install git+git://github.com/pyeve/cerberus.git
...
Cleaning up...
```

And you’re done!

### 3.2 Cerberus Usage

#### 3.2.1 Basic Usage

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

```python
>>> 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:

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

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

```python
>>> 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.

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> v.schema = {}
>>> v.allow_unknown = {'type': 'string'}
>>> v.validate({'an_unknown_field': 'john'})
True
>>> v.validate({'an_unknown_field': 1})
False
>>> v.errors
{'an_unknown_field': ['must be one of these types: ('string',)']}
```

`allow_unknown` can also be set at initialization:

```python
>>> 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:

```python
>>> v = Validator()
>>> v.allow_unknown
False
>>> schema = {
...     'name': {'type': 'string'},
...     'a_dict': {
...         'type': 'dict',
...     }
... }
```

(continues on next page)
... '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:

```python
>>> 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
```
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 passed as 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.
In the example above we define a schema 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 RulesSetRegistry and SchemaSetRegistry objects and bind them to the rules_set_registry respectively schema_registry of a validator. You may also set these as keyword-arguments upon initialisation.

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

```python
>>> 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}}
```

```python
>>> 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 by default when passed to the validator or a new set of rules is set for a top-level 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:

```python
>>> 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)
(continues on next page)"
Schema validation can be disabled by using the UnconcernedValidator or getting a validator class from validator_factory() with the argument validated_schema set to False. This can useful in an application’s production environment to reduce startup time.

### 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.

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

You don’t have to use YAML of course, you can use your 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` 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 `collections.abc.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.

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

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

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

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

>>> v.validate({'role': 'supplier'})
True

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

>>> v.errors
{'role': ['unallowed value intern']}  

>>> v.schema = {'a_restricted_integer': {'type': 'integer', 'allowed': [-1, 0, 1]}}

>>> v.validate({'a_restricted_integer': -1})
True

>>> v.validate({'a_restricted_integer': 2})
False

>>> v.errors
{'a_restricted_integer': ['unallowed value 2']}  
```

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

### 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 functionname(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:

```python
>>> 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.

```python
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.

```python
>>> 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.

```python
>>> 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.

```python
>>> schema = {'field1': {'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.

```python
>>> 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

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

(continues on next page)
# 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'}
...     }
...   }
... }

>>> document = {'a_dict': {'bar': 'bar'}}

>>> v.validate(document, schema)
False

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

3.4. Validation Rules
Cerberus Documentation, Release 2.0.dev0

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.

```python
>>> 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:

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

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

```python
>>> v = Validator()
>>> schema = {'this_field': {'type': 'dict',
... 'excludes': 'that_field'}}
... ```
You can also pass multiple fields to exclude in a list:

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

### 3.4.10 forbidden

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

```python
>>> 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.

```python
>>> 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 *itemsrules* rule for dealing with arbitrary length *list* types.
3.4.12 itemsrules

All items of the term:sequence will be validated against the rules provided in the constraint.

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

3.4.13 keysrules

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

```python
>>> 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

3.4.14 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:

```python
{'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.

3.4.15 min, max

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

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

(continues on next page)
True

```python
>>> 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.

### 3.4.16 minlength, maxlength

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

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

### 3.4.17 noneof

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

New in version 0.9.

### 3.4.18 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`.

```python
>>> 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.

### 3.4.19 *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.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>all_of</code></td>
<td>Validates if all of the provided constraints validates the field.</td>
</tr>
<tr>
<td><code>any_of</code></td>
<td>Validates if any of the provided constraints validates the field.</td>
</tr>
<tr>
<td><code>none_of</code></td>
<td>Validates if none of the provided constraints validates the field.</td>
</tr>
<tr>
<td><code>one_of</code></td>
<td>Validates if exactly one of the provided constraints applies.</td>
</tr>
</tbody>
</table>

**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:

```python
>>> 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 `any_of` rule tests each rules set in the list. Hence, the above schema is equivalent to creating two separate schemas:

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

```python
{'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:

```python
>>> schemas = [{'department': {'required': True, 'regex': '^IT$'}, 'phone': {'nullable': True}}, ...
...                           {'department': {'required': True}, 'phone': {'required': True}}]
>>> employee_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)
```

3.4.20 oneof

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

New in version 0.9.

3.4.21 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.
3.4.22 regex

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

```python
>>> schema = {
    ...
    'email': {
        ...
        'type': 'string',
        ...
        'regex': r'^[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 (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.

3.4.23 require_all

This can be used in conjunction with the `schema` 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.

3.4.24 required

If True the field is mandatory. Validation will fail when it is missing, unless `validate()` is called with `update=True`:

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

>>> v.errors
{'name': ['required field']}
```
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.

3.4.25 schema

A given mapping as value will be validated against the schema that is provided as constraint.

```python
>>> 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.

3.4.26 type

Tests whether the field value’s type matches (one of) the specified type(s). There are several ways a type can be specified, each with dis-/advantages regarding different usage aspects:

- any object like classes or abstract types that can be used as second argument to the builtin’s `isinstance()` function
- strings that reference either one of the named,
  - strict types whose mapping to actual types are documented in the table below
  - abstract types that map to the types from the `collections.abc` module - their extend depends on the Python version - these are are camel-cased (e.g. `Set` or `MutableMapping`)  
  - custom types that can be defined per `Validator` class
- generic aliases from the `typing` module, including compound types
-- type parameters that are given as string have Cerberus’ semantics of named types and are not resolved like static type checkers do, e.g. `Set["string"]` is a valid type specification

Named types allow the serialization of schemas and the exclusion of particular subtypes.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Python Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool</td>
</tr>
<tr>
<td>bytesarray</td>
<td>bytearray</td>
</tr>
<tr>
<td>bytes</td>
<td>bytes</td>
</tr>
<tr>
<td>complex</td>
<td>complex</td>
</tr>
<tr>
<td>date</td>
<td><code>datetime.date</code>, but not its subclass <code>datetime.datetime</code></td>
</tr>
<tr>
<td>datetime</td>
<td><code>datetime.datetime</code></td>
</tr>
<tr>
<td>dict</td>
<td>dict</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>frozenset</td>
<td>frozenset</td>
</tr>
<tr>
<td>integer</td>
<td><code>int</code>, but not its subclass <code>bool</code></td>
</tr>
<tr>
<td>list</td>
<td>list</td>
</tr>
<tr>
<td>number</td>
<td><code>float, int</code>, but not <code>bool</code></td>
</tr>
<tr>
<td>set</td>
<td>set</td>
</tr>
<tr>
<td>string</td>
<td>str</td>
</tr>
<tr>
<td>tuple</td>
<td>tuple</td>
</tr>
<tr>
<td>type</td>
<td><code>type (classes)</code></td>
</tr>
</tbody>
</table>

Here are examples of the different ways to specify a type:

```python
>>> document = {"items": frozenset(('a', 'b', 'c'))}
>>> v.schema = {"items": {"type": frozenset}}
>>> v.validate(document)
True
>>> v.schema = {"items": {"type": 'frozenset'}}
>>> v.validate(document)
True
>>> v.schema = {"items": {"type": 'set'}}
>>> v.validate(document)
False
>>> v.schema = {"items": {"type": 'Set'}}
>>> v.validate(document)
True
>>> import typing
>>> v.schema = {"items": {"type": typing.Set[int]}}
>>> v.validate(document)
False
>>> v.schema = {"items": {"type": typing.Set["integer"]}}
>>> v.validate(document)
False
```

A list of types can be used to allow different values of different types:
>>> 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'],...
                   'itemsrules': {'type': 'string'}}
>>> v.validate({'quotes': 'Hello world!'}
True
>>> v.validate({'quotes': [1, 'Heureka!']})
False
>>> v.errors
{'quotes': [{0: ['must be one of these types: ('string',)']}]}]

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.

Changed in version 0.7.1: dict and list typechecking are now performed with the more generic Mapping and Sequence types from the built-in collections module. This means that instances of custom types designed to the same interface as the built-in 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.

```python
>>> 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
```

(continues on next page)
New in version 0.7.

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.

```python
>>> 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.

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

```python
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.

```python
>>> 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}
```

(continues on next page)
>>> 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**

```python
>>> schema = {'cats': {'type': 'integer'}}
>>> document = {'cats': 'two'}
>>> v.validate(document, schema)
False
>>> cerberus.errors.TYPE in v._errors
True
>>> v.document_error_tree['cats'].errors
... == v.schema_error_tree['cats']['type'].errors
True
>>> cerberus.errors.TYPE in v.document_error_tree['cats']
True
>>> v.document_error_tree['cats'][cerberus.errors.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:

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

This is how we would go to implement that:

```python
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:

```python
>>> 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 `TypeDefininition` to the designated type name in `types_mapping`:

```python
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`:

```python
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 2.0 from any 1.x version*.

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 a 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:

```python
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:

```python
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 coerced value or point to a method as `rename_handler`. The method name must be prefixed with `_normalize_coerce_`:

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

    def _normalize_coerce_multiply(self, value):
        return value * self.multiplier
```
```python
>>> schema = {'foo': {'coerce': 'multiply'}}
>>> document = {'foo': 2}
>>> MyNormalizer(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.

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

```python
>>> 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:

```python
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')

        # 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

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

Note: There’s currently a feature freeze until the basic code modernization for the 2.0 release is finished. Have a look at the ROADMAP.md for a status on its progress.

3.8.1 Getting Started

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

3.8.2 Making Changes

• Fork the repository on GitHub.
• Create a topic branch from where you want to base your work.
• This is usually the master branch.
• Please avoid working directly on master branch.
• Make commits of logical units (if needed rebase your feature branch before submitting it).
• Make sure your commit messages are in the proper format.
• If your commit fixes an open issue, reference it in the commit message (#15).
• Make sure 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. CI will reject a change that does not conform to the guidelines.
• Don’t forget to add yourself to AUTHORS.

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

### 3.8.3 Submitting Changes

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

### 3.8.4 First time contributor?

It’s alright. We’ve all been there.

### 3.8.5 Type annotations

There are two purposes for type annotations in the codebase: The first one is to provide information about callables’ signatures for applications such as IDEs, so they can give useful hints within the client codebase when unexpected types of values are passed to Cerberus’ interfaces. The other is to document the intended use of variables, such as `bar = {} # type: Dict[str, int]`. Though testing type annotations with checkers like mypy can reveal possible problems, this is not what they’re intended for in this project. The annotations however must be checked for correctness. As Python’s typing system and the most advanced type checker aren’t mature yet and it isn’t possible to annotate every part correctly with a reasonable effort due to the nature of Cerberus architecture, there has to be a compromise to comply with the afore mentioned goals and requirement:

1. All parts of the public API (functions, methods, class properties) must be type annotated. It is okay to cast variables and instruct type checkers to ignore errors (# type: ignore). It should occasionally be checked whether these can be removed due to progress both within Cerberus’ code and the used checker.

2. The annotation of code that is not supposed to be used directly is helpful for maintenance and should be added if the checker doesn’t need to be pleased with the previously mentioned methods. If such are required, type annotations must not be added at all.

### 3.8.6 Don’t know where to start?

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

### 3.8.7 Running the Tests

Cerberus runs under Python 3.5 to 3.8 and PyPy3. Therefore test will be run in those platforms in our continuous integration server.

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

```bash
$ python setup.py test
```
Testing with other Python versions

Before you submit a pull request, make sure your tests and changes run in all supported python versions. 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.

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

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

```plaintext
_________ summary _________
py35: commands succeeded
py36: commands succeeded
py37: commands succeeded
py38: commands succeeded
pypy3: commands succeeded
doclinks: commands succeeded
doctest: commands succeeded
linting: commands succeeded
congratulations :) 
```

If something goes wrong and one test fails, you might need to run that test in the specific python version. You can use the created environments to run some specific tests. For example, if a test suite fails in Python 3.5:

```bash
$ tox -e py35
```

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

Using Pytest

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

```bash
$ pytest cerberus/tests
```

Using Docker

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

```bash
$ ./run-docker-tests -e pypy3 -e doctest
```

You can run the script without any arguments to test the project exactly as Continuous Integration does without having to setup anything. The tox environments are preserved in a volume named cerberus-tox, just remove it with docker volume rm to clean them.

Running the benchmarks

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

```bash
$ 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`.

Continuous Integration

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

3.8.8 Source Code

Source code is available at [GitHub](https://github.com).

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)
3.9.2 Generous Backers

Generous backers who actively support Eve and Cerberus development:

![blokt](image)

3.10 API Documentation

3.10.1 Validator Class

```python
```

**Bases:** cerberus.base.UnconcernedValidator

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 doc-strings of rules’ methods and is effectively used as validation schema for `schema`.

**Parameters**

- `schema` – See `schema`. Defaults to `None`.
- `ignore_none_values` – See `ignore_none_values`. Defaults to `False`.
- `allow_unknown` – See `allow_unknown`. Defaults to `False`.
- `require_all` – See `require_all`. Defaults to `False`.
- `purge_unknown` – See `purge_unknown`. Defaults to `False`.
- `purge_readonly` – Removes all fields that are defined as `readonly` in the normalization phase.
- `error_handler` – 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`.

**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*. 

3.10. API Documentation
Type: \texttt{bool} or any mapping

\texttt{classmethod clear\_caches()}  
Purge the cache of known valid schemas.

\textbf{errors}  
The errors of the last processing formatted by the handler that is bound to \texttt{error\_handler}.

\textbf{ignore\_none\_values}  
Whether to not process \texttt{None}-values in a document or not.  
Type: \texttt{bool}

\textbf{is\_child}  
True for child-validators obtained with \texttt{_get\_child\_validator()}.
Type: \texttt{bool}

\texttt{normalized(document: Mapping[Hashable, Any], schema: Union[ValidatedSchema, Mapping[Hashable, Mapping[str, Any]], None] = None, always\_return\_document: bool = False) \rightarrow Optional[Mapping[Hashable, Any]]}  
Returns the document normalized according to the specified rules of a schema.

\textbf{Parameters}  
\begin{itemize}
  \item \texttt{document} – The document to normalize.
  \item \texttt{schema} – The validation schema. Defaults to \texttt{None}. If not provided here, the schema must have been provided at class instantiation.
  \item \texttt{always\_return\_document} – Return the document, even if an error occurred. Defaults to: \texttt{False}.
\end{itemize}

\textbf{Returns} A normalized copy of the provided mapping or \texttt{None} if an error occurred during normalization.

\textbf{purge\_readonly}  
If \texttt{True}, fields declared as readonly will be deleted from the document unless a validation is called with disabled normalization.  
Type: \texttt{bool}

\textbf{purge\_unknown}  
If \texttt{True}, unknown fields will be deleted from the document unless a validation is called with disabled normalization. Also see \textit{Purging Unknown Fields}.
Type: \texttt{bool}

\textbf{require\_all}  
If \texttt{True} known fields that are defined in the schema will be required.  
Type: \texttt{bool}

\textbf{root\_allow\_unknown}  
The \texttt{allow\_unknown} attribute of the first level ancestor of a child validator.

\textbf{root\_document}  
The \texttt{document} attribute of the first level ancestor of a child validator.

\textbf{root\_require\_all}  
The \texttt{require\_all} attribute of the first level ancestor of a child validator.

\textbf{root\_schema}  
The \texttt{schema} attribute of the first level ancestor of a child validator.
rules_set_registry
The registry that holds referenced rules sets.
Type: Registry

schema
The validation schema of a validator. When a schema is passed to a validator method (e.g. validate), it replaces this attribute.
Type: any mapping or None

schema_registry
The registry that holds referenced schemas.
Type: Registry

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

Parameters
• document – The document to normalize.
• schema – The validation schema. Defaults to None. If not provided here, the schema must have been provided at class instantiation.
• update – If True, required fields won’t be checked.
• normalize – If True, normalize the document before validation.

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

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

3.10.2 Rules Set & Schema Registry

class cerberus.base.Registry (definitions: Union[Mapping[str, RegistryItem], Iterable[Tuple[str, RegistryItem]]] = ()
A registry to store and retrieve schemas and parts of it by a name that can be used in validation schemas.

Parameters definitions – Optional, initial definitions.

add (name: str, definition: RegistryItem) → None
Register a definition to the registry. Existing definitions are replaced silently.

Parameters
• name – The name which can be used as reference in a validation schema.
• definition – The definition.

all () → Mapping[str, RegistryItem]
Returns a dict with all registered definitions mapped to their name.

clear()
Purge all definitions in the registry.
**extend** *(definitions: Union[Mapping[str, RegistryItem], Iterable[Tuple[str, RegistryItem]]]) → None*
Add several definitions at once. Existing definitions are replaced silently.

**Parameters**
- **definitions** – The names and definitions.

**get** *(name: str, default: Optional[RegistryItem] = None) → Optional[RegistryItem]*
Retrieve a definition from the registry.

**Parameters**
- **name** – The reference that points to the definition.
- **default** – Return value if the reference isn’t registered.

**remove** *(names)* → None
Unregister definitions from the registry.

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

### 3.10.3 Type Definitions

**class** cerberus.schema.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.

**__call__** *(errors: Iterable[cerberus.errors.ValidationError]) → Any*
Returns errors in a handler-specific format.

**__init__** *(*args, **kwargs)*
Optionally initialize a new instance.

**__iter__** () → Iterator[Any]
Be a superhero and implement an iterator over errors.

**__weakref__**
list of weak references to the object (if defined)

**add** *(error: cerberus.errors.ValidationError) → None*
Add an error to the errors’ container object of a handler.

**Parameters**
- **error** – The error to add.

**emit** *(error: cerberus.errors.ValidationError) → None*
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** – The error to emit.

**end** *(validator: UnconcernedValidator) → None*

Gets called when a validation ends.

Parameters **validator** – The calling validator.

**extend** *(errors: Iterable[cerberus.errors.ValidationError]) → None*

Adds all errors to the handler’s container object.

**start** *(validator: UnconcernedValidator) → None*

Gets called when a validation starts.

Parameters **validator** – The calling validator.

**class** cerberus.errors.BasicErrorHandler *(tree: Dict[KT, VT] = None)*

Bases: cerberus.errors.BaseErrorHandler

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: int, rule: Optional[str]*)

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.


A simple class to store and query basic error information.

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

- **code** = None
  The error’s identifier code. Type: int

- **constraint** = None
  The constraint that failed.

- **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** = None
  The path to the field within the document that caused the error. Type: tuple

- **field**
  Field of the contextual mapping, possibly None.

- **info** = None
  May hold additional information about the error. Type: tuple

- **is_group_error**
  True for errors of bulk validations.

- **is_logic_error**
  True for validation errors against different schemas with *of-rules.
**is_normalization_error**
True for normalization errors.

**rule = None**
The rule that failed. Type: *string*

**schema_path = None**
The path to the rule within the schema that caused the error. Type: *tuple*

**value = None**
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:

<table>
<thead>
<tr>
<th>Code (dec.)</th>
<th>Code (hex.)</th>
<th>Name</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0</td>
<td>CUSTOM</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>0x2</td>
<td>REQUIRED_FIELD</td>
<td>required</td>
</tr>
<tr>
<td>3</td>
<td>0x3</td>
<td>UNKNOWN_FIELD</td>
<td>None</td>
</tr>
<tr>
<td>4</td>
<td>0x4</td>
<td>DEPENDENCIES_FIELD</td>
<td>dependencies</td>
</tr>
<tr>
<td>5</td>
<td>0x5</td>
<td>DEPENDENCIES_FIELD_VALUE</td>
<td>dependencies</td>
</tr>
<tr>
<td>6</td>
<td>0x6</td>
<td>EXCLUDES_FIELD</td>
<td>excludes</td>
</tr>
<tr>
<td>34</td>
<td>0x22</td>
<td>EMPTY</td>
<td>empty</td>
</tr>
<tr>
<td>35</td>
<td>0x23</td>
<td>NULLABLE</td>
<td>nullable</td>
</tr>
<tr>
<td>36</td>
<td>0x24</td>
<td>TYPE</td>
<td>type</td>
</tr>
<tr>
<td>38</td>
<td>0x26</td>
<td>ITEMS_LENGTH</td>
<td>items</td>
</tr>
<tr>
<td>39</td>
<td>0x27</td>
<td>MIN_LENGTH</td>
<td>minlength</td>
</tr>
<tr>
<td>40</td>
<td>0x28</td>
<td>MAX_LENGTH</td>
<td>maxlength</td>
</tr>
<tr>
<td>65</td>
<td>0x41</td>
<td>REGEX_MISMATCH</td>
<td>regex</td>
</tr>
<tr>
<td>66</td>
<td>0x42</td>
<td>MIN_VALUE</td>
<td>min</td>
</tr>
<tr>
<td>67</td>
<td>0x43</td>
<td>MAX_VALUE</td>
<td>max</td>
</tr>
<tr>
<td>68</td>
<td>0x44</td>
<td>UNALLOWED_VALUE</td>
<td>allowed</td>
</tr>
<tr>
<td>69</td>
<td>0x45</td>
<td>UNALLOWED_VALUES</td>
<td>allowed</td>
</tr>
<tr>
<td>70</td>
<td>0x46</td>
<td>FORBIDDEN_VALUE</td>
<td>forbidden</td>
</tr>
<tr>
<td>71</td>
<td>0x47</td>
<td>FORBIDDEN_VALUES</td>
<td>forbidden</td>
</tr>
<tr>
<td>72</td>
<td>0x48</td>
<td>MISSING_MEMBERS</td>
<td>contains</td>
</tr>
</tbody>
</table>

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 definitions that are shipped with the `errors` module:
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Code (dec.)</th>
<th>Code (hex.)</th>
<th>Name</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>0x60</td>
<td>NORMALIZATION</td>
<td>None</td>
</tr>
<tr>
<td>97</td>
<td>0x61</td>
<td>COERCION_FAILED</td>
<td>coerce</td>
</tr>
<tr>
<td>98</td>
<td>0x62</td>
<td>RENAMING_FAILED</td>
<td>rename_handler</td>
</tr>
<tr>
<td>99</td>
<td>0x63</td>
<td>READONLY_FIELD</td>
<td>readonly</td>
</tr>
<tr>
<td>100</td>
<td>0x64</td>
<td>SETTING_DEFAULT_FAILED</td>
<td>default_setter</td>
</tr>
<tr>
<td>128</td>
<td>0x80</td>
<td>ERROR_GROUP</td>
<td>None</td>
</tr>
<tr>
<td>129</td>
<td>0x81</td>
<td>SCHEMA</td>
<td>schema</td>
</tr>
<tr>
<td>130</td>
<td>0x82</td>
<td>ITEMSRULES</td>
<td>itemsrules</td>
</tr>
<tr>
<td>131</td>
<td>0x83</td>
<td>KEYSRULES</td>
<td>keysrules</td>
</tr>
<tr>
<td>132</td>
<td>0x84</td>
<td>VALUESRULES</td>
<td>valuesrules</td>
</tr>
<tr>
<td>143</td>
<td>0x8f</td>
<td>ITEMS</td>
<td>items</td>
</tr>
<tr>
<td>144</td>
<td>0x90</td>
<td>LOGICAL</td>
<td>None</td>
</tr>
<tr>
<td>145</td>
<td>0x91</td>
<td>NONEOF</td>
<td>noneof</td>
</tr>
<tr>
<td>146</td>
<td>0x92</td>
<td>ONEOF</td>
<td>oneof</td>
</tr>
<tr>
<td>147</td>
<td>0x93</td>
<td>ANYOF</td>
<td>anyof</td>
</tr>
<tr>
<td>148</td>
<td>0x94</td>
<td>ALLOF</td>
<td>allof</td>
</tr>
</tbody>
</table>

Error Containers

class cerberus.errors.ErrorList
A list for ValidationError instances that can be queried with the in keyword for a particular ErrorDefinition.


    add (error: cerberus.errors.ValidationError) -> None
    Add an error to the tree.

    fetch_errors_from (path: Tuple[Hashable, ...]) -> cerberus.errors.ErrorList
    Returns all errors for a particular path.

    fetch_node_from (path: Tuple[Hashable, ...]) -> cerberus.errors.ErrorTreeNode
    Returns a node for a path.


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


    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

`cerberus.validator_factory` (name: str, bases: Union[type, Tuple[type], None] = None, namespace: Optional[Dict[KT, VT]] = None, validated_schema: bool = True) → type

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** – The name of the new class.
- **bases** – Class(es) with additional and overriding attributes.
- **namespace** – Attributes for the new class.
- **validated_schema** – Indicates that schemas that are provided to the validator are to be validated.

**Returns** The created class.

3.10.8 Schema Validation Schema

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

```python
ChainMap({
    'coerce': {'oneof': [{'type': ('Callable',)},
                         {'itemsrules': {'oneof': [{'type': ('Callable',)},
                                                   {'allowed': (),
                                                    'type': ('string',)}],
                         'type': ('Iterable',)},
                        {'allowed': (),
                         'type': ('string',)}],
    'default': {'nullable': True},
    'default_setter': {'oneof': [{'type': ('Callable',)},
                                   {'allowed': (),
                                    'type': ('string',)}],
                      'type': ('Iterable',)},
    'purge_unknown': {'type': ('Iterable',)},
    'rename': {'type': ('Iterable',)},
    'rename_handler': {'oneof': [{'type': ('Callable',)},
                                 {'itemsrules': {'oneof': [{'type': ('Callable',)},
                                                          {'allowed': (),
                                                           'type': ('string',)}],
                                 'type': ('Iterable',)},
                                {'allowed': (),
                                 'type': ('string',)}],
                  'type': ('Iterable',)},
    'allof': {'logical': 'allof', 'type': ('Iterable',)},
    'allow_unknown': {'oneof': [{'type': ('Callable',)},
                                 {'itemsrules': {'oneof': [{'type': ('Callable',)},
                                                          {'allowed': (),
                                                           'type': ('string',)}],
                                 'type': ('Iterable',)},
                                {'allowed': (),
                                 'type': ('string',)}],
                  'type': ('Iterable',)},
    'anyof': {'logical': 'anyof', 'type': ('Iterable',)},
    'check_with': {'oneof': [{'type': ('Callable',)},
                             {'itemsrules': {'oneof': [{'type': ('Callable',)},
                                                      {'allowed': (),
                                                       'type': ('string',)}],
                             'type': ('Iterable',)},
                            {'allowed': (),
                             'type': ('string',)}],
                  'type': ('Iterable',)}
})
```

(continues on next page)
3.11 Frequently Asked Questions

3.11.1 Can I use Cerberus to validate objects?

Yes. See Validating user objects with Cerberus.

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).

3.11. Frequently Asked Questions 49
3.12.1 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.

3.12.2 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.

3.12.3 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:

```python
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 In Development

New

- Classes can be used as constraint for the `type` rule (#374)
- The abstract base classes of the standard library’s `collections.abc` module are available as named types for the `type` rule (#374)
- Generic type aliases from the `typing` module can be used as constraints for the `type` rule, including parametrized ones a.k.a. compound types (#374)
- Support for Python 3.5 is removed.

Changed

- Most of Python’s builtin data types are available as named constraint for the `type` rule. They do not match their builtin subtypes, e.g. `True` isn’t recognized as "integer", this also applies to `datetime.date` objects. (#374)
Fixed

- WARNING Unexpected keys present on Black: python_version (#494)

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


• 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: ‘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’ keyword 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


• 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 Šemančík).
• 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.
- Not-nullable fields are validated independently 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 called 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).
- ‘minlength’ 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 length (Martijn Vermaat).
- ‘allowed’ rule on strings (Martijn 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 2.0 from any 1.x version

#### 3.14.1 Major Changes and Deprecations

**Rules**

A few rules have been renamed:

<table>
<thead>
<tr>
<th>Old name</th>
<th>New name</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyschema</td>
<td>keysrules</td>
</tr>
<tr>
<td>schema for sequences</td>
<td>itemsrules</td>
</tr>
<tr>
<td>validator</td>
<td>check_with</td>
</tr>
<tr>
<td>valueschema</td>
<td>valuesrules</td>
</tr>
</tbody>
</table>

As a consequence custom validators’ methods that are referenced by name in constraints for the `check_with` rule must be prefixed with `_check_with_` instead of `_validator_`.

**Errors**

A few constant names of error definitions in the `errors` module have been renamed:

<table>
<thead>
<tr>
<th>Old name</th>
<th>New name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD_ITEMS</td>
<td>ITEMS</td>
</tr>
<tr>
<td>BAD_TYPE</td>
<td>TYPE</td>
</tr>
<tr>
<td>EMPTY_NOT_ALLOWED</td>
<td>EMPTY</td>
</tr>
<tr>
<td>MAPPING_SCHEMA</td>
<td>SCHEMA</td>
</tr>
<tr>
<td>NOT_NONNULL</td>
<td>NULLABLE</td>
</tr>
<tr>
<td>SEQUENCE_SCHEMA</td>
<td>ITEMSRULES</td>
</tr>
</tbody>
</table>
The constant `BAD_TYPE_FOR_SCHEMA` has been removed permanently.

**Type checking**

The use of methods prefixes with `_validate_type_` for checking the type of a value has been abandoned. See *Extending Cerberus* how to define custom types with `TypeDefinition` objects. If you used such methods to test more than the type of the value, use the `check_with` rules instead.

### 3.15 Upgrading to Cerberus 1.0 from prior versions

#### 3.15.1 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.

#### 3.15.2 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 Pythoniverse¹, 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.

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:

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

1.0:

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

3.16 Authors

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

¹ compare dictionary
3.16.1 Core maintainers

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

3.16.2 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
A full, up-to-date list of contributors is available from git with:

git shortlog -sne

### 3.17 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:

#### 3.17.1 Blog

New releases are usually announced on my Website.

#### 3.17.2 Twitter

I often tweet about new features and releases of Cerberus. Follow @nicolaiaroci.
3.17.3 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.

3.17.4 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.

3.17.5 GitHub repository

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

3.18 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.
Copyright Notice

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

__call__()  (cerberus.errors.BaseErrorHandler method), 44
__init__()  (cerberus.errors.BaseErrorHandler method), 44
__iter__()  (cerberus.errors.BaseErrorHandler method), 44
__weakref__  (cerberus.errors.BaseErrorHandler attribute), 44

A
add()  (cerberus.base.Registry method), 43
add()  (cerberus.errors.BaseErrorHandler method), 44
add()  (cerberus.errors.ErrorTree method), 47
all()  (cerberus.base.Registry method), 43
allow_unknown  (cerberus.Validator attribute), 41

B
BaseErrorHandler  (class in cerberus.errors), 44
BasicErrorHandler  (class in cerberus.errors), 45

C
child_errors  (cerberus.errors.ValidationError attribute), 45
clear()  (cerberus.base.Registry method), 43
clear_caches()  (cerberus.Validator class method), 42
code  (cerberus.errors.ValidationError attribute), 45
constraint  (cerberus.errors.ValidationError attribute), 45

definitions_errors  (cerberus.errors.ValidationError attribute), 45
document_path  (cerberus.errors.ValidationError attribute), 45
DocumentError, 47
DocumentErrorTree  (class in cerberus.errors), 47

E
emit()  (cerberus.errors.BaseErrorHandler method), 44
end()  (cerberus.errors.BaseErrorHandler method), 45
ErrorDefinition  (class in cerberus.errors), 45
ErrorList  (class in cerberus.errors), 47
effects  (cerberus.Validator attribute), 42
ErrorTree  (class in cerberus.errors), 47
extend()  (cerberus.base.Registry method), 43
extend()  (cerberus.errors.BaseErrorHandler method), 45

F
fetch_errors_from()  (cerberus.errors.ErrorTree method), 47
fetch_node_from()  (cerberus.errors.ErrorTree method), 47
field  (cerberus.errors.ValidationError attribute), 45

G
get()  (cerberus.base.Registry method), 44

I
ignore_none_values  (cerberus.Validator attribute), 42
info  (cerberus.errors.ValidationError attribute), 45
is_child  (cerberus.Validator attribute), 42
is_group_error  (cerberus.errors.ValidationError attribute), 45
is_logic_error  (cerberus.errors.ValidationError attribute), 45
is_normalization_error  (cerberus.errors.ValidationError attribute), 45

N
normalized()  (cerberus.Validator method), 42

P
purge_readonly  (cerberus.Validator attribute), 42
purge_unknown (cerberus.Validator attribute), 42

R
Registry (class in cerberus.base), 43
remove() (cerberus.base.Registry method), 44
require_all (cerberus.Validator attribute), 42
root_allow_unknown (cerberus.Validator attribute), 42
root_document (cerberus.Validator attribute), 42
root_require_all (cerberus.Validator attribute), 42
root_schema (cerberus.Validator attribute), 42
rule (cerberus.errors.ValidationError attribute), 46
rules_set_registry (cerberus.Validator attribute), 42
RulesSetRegistry (class in cerberus.schema), 44

S
schema (cerberus.Validator attribute), 43
schema_path (cerberus.errors.ValidationError attribute), 46
schema_registry (cerberus.Validator attribute), 43
SchemaError, 47
SchemaErrorTree (class in cerberus.errors), 47
SchemaRegistry (class in cerberus.schema), 44
start() (cerberus.errors.BaseErrorHandler method), 45

T
TypeDefininition (class in cerberus), 44

V
validate() (cerberus.Validator method), 43
validated() (cerberus.Validator method), 43
ValidationErrors (class in cerberus.errors), 45
Validator (class in cerberus), 41
validator_factory() (in module cerberus), 48
value (cerberus.errors.ValidationError attribute), 46