pycerberus Documentation

Release 0.6.99.20201111

Felix Schwarz

Nov 11, 2020

Contents

pycerberus is a library to check user data thoroughly so that you can protect your application from malicious (or just garbled) input data.

- **Remove stupid code which converts input values:** After values are validated, you can work with real Python types instead of strings - e.g. 42 instead of '42', convert database IDs to model objects transparently.
- **Implement custom validation rules:** Writing custom validators is straightforward, everything is well documented and pycerberus only uses very little Python magic.
- **Focus on your value-adding application code:** Save time by implementing every input validation rule only once, but 100% right instead of implementing a dozen different half-baked solutions.
- **Ready for global business:** i18n support (based on GNU gettext) is built in, adding custom translations is easy.
- **Tune it for your needs:** You can implement custom behavior in your validators, e.g. fetch translations from a database instead of using gettext or define custom translations for built-in validators.
- **Use it wherever you like:** pycerberus is used in a SMTP server, trac macros as well as web applications - there are no dependencies on a specific context like web development.

CHAPTER 1

Documentation

1.1 Installation and Setup

pycerberus is just a Python library which uses setuptools so it does not require a special setup. It has no dependencies besides the standard Python library. There are some optional packages which you can choose to install:

- **Babel** is the most convenient option to generate the gettext catalog files so you can see localized error messages.
- I'm using **nosetests** ("nose") to run all the automated tests.

pycerberus has been tested on **Python 2.6-2.7** as well as **Python 3**. To know more about Python 3 support (and its limitations), please read the section *Python 3 Support*.

1.2 Background

In every software you must check carefully that untrusted user input data matches your expectations. Unvalidated user input is a common source of security flaws. However many checks are repetitive and validation logic tends to be scattered all around the code. Because basic checks are duplicated, developers forget to check also for uncommon edge cases. Eventually there is often also some code to convert the input data (usually strings) to more convenient Python data types like int or bool.

pycerberus is a library that tackles these common problems and allows you to write tailored validators to perform additional checks. Furthermore the library also has built-in support for less common (but important) use cases like internationalization.

The library itself is heavily inspired by FormEncode by Ian Bicking. Therefore most of FormEncode's design rationale is directly applicable to pycerberus. However several things about FormEncode annoyed me so much that I decided to write my own library when I needed one for my SMTP server project pymta.

1.3 Philosophy and Design

1.3.1 Rules are declared explicitly: Separating policy from mechanism

pycerberus separates validation rules ("Validators") from the objects they validate against. It might be tempting to derive the validation rules from restrictions you specified earlier (e.g. from a class which is mapped by an ORM to a database). However that approach completely ignores that validation typically depends on context: In an API you have typically a lot more freedom in regard to allowed values compared to a public web interface where input needs to conform to a lot more checks. In a system where you declare the validation explicitly, this is possible. Also it is quite easy writing some code that generates a bottom line of validation rules automatically based on your ORM model and add additional restrictions depending on the context.

As pycerberus is completely context-agnostic (not being bundled with a specific framework), you can use it in many different places (e.g. web applications with different frameworks, server applications, check parameters in a library, ...).

Further reading: FormEncode's design rationale - most of the design ideas are also present in pycerberus.

1.4 Development Status

I'm using pycerberus currently (June 2019, version 0.6.99) in several projects and the basic API (based on "rich results") seems to be fine. One problem is that I have a lot of code on top of the open source version which should probably be open sourced as well (e.g. dependencies for formvalidators).

The code is a bit simple, does not have many validators but is pretty solid. The API for single validators is basically complete, i18n support is built in and there is decent documentation covering all important aspects. You can check multiple values (e.g. a web form) easily using a validation Schema ("compound validator").

In the future 'll try to increase the number of *built-in validators for specific domains* (e.g. localized numbers). Another interesting topic will be *integration into different frameworks* like TurboGears and trac.

However I have to say that I'm pretty satisfied with the current status so adding more features to pycerberus won't be my #1 priority in the next months. The current API and functionality was well-suited even when validating input parameters of a SMTP server so I think most use cases should be actually covered.

1.5 Using Validators

In pycerberus "Validators" are used to specify validation rules which ensure that the input matches your expectations. Every basic validator validates a just single value (e.g. one specific input field in a web application). When the validation was successful, the validated and converted value is returned. If something is wrong with the data, an exception is raised:

```
from pycerberus.validators import IntegerValidator
IntegerValidator().process('42') # returns 42 as int
```

pycerberus puts conversion and validation together in one call because of two main reasons:

- As a user you need to convert input data (usually strings) anyway into a more sensible format (e.g. int). These lines of code are redundant because you declared in the validator already what the value should be.
- During the validation process, it is very easy to do also the conversion. In fact many validations are done just by trying to do a conversion and catch all exceptions that were raised during that process.

1.5.1 Validation Errors

Every validation error will trigger an exception, usually an `InvalidDataError`. This exception will contain a translated error message which can be presented to the user, a key so you can identify the exact error programmatically and the original, unmodified value:

```
from pycerberus.errors import InvalidDataError
from pycerberus.validators import IntegerValidator
try:
    IntegerValidator().process('foo')
except InvalidDataError, e:
    details = e.details()
    details.msg()          # u'Please enter a number.'
    details.key()          # 'invalid_number'
    details.value()        # 'foo'
    details.context()      # {}
```

1.5.2 Configuring Validators

You can configure the behavior of the validator when instantiating it. For example, if you pass `required=False` to the constructor, most validators will also accept `None` as a valid value:

```
IntegerValidator(required=True).process(None)   # -> validation error
IntegerValidator(required=False).process(None) # None
```

Validators support different configuration options which are explained along the validator description.

1.5.3 Context

All validators support an optional `context` argument (which defaults to an emtpy dict). It is used to plug validators into your application and make them aware of the overall system state: For example a validator must know which locale it should use to translate an error message to the correct language without relying on some global variables:

```
context = {'locale': 'de'}
validator = IntegerValidator()
validator.process('foo', context=context) # u'Bitte geben Sie eine Zahl ein.'
```

The context variable is especially useful when writing custom validators - locale is the only context information that pycerberus itself cares about.

1.6 Available validators

pycerberus contains some basic validators already. You can use them as they are or use them as a basis for more specialized validators. Below you find a list of all included validators.

class pycerberus.validators.basic_numbers.**IntegerValidator**(*self*)

Bases: *pycerberus.api.Validator*

convert(*value*, *context*)

Convert the input value to a suitable Python instance which is returned. If the input is invalid, raise an `InvalidDataError`.

is_empty(*value*, *context*)

Decide if the value is considered an empty value.

keys()
Return all keys defined by this specific validator class.

message_for_key(*key*, *context*)
Return a message for a specific key. Implement this method if you want to avoid calls to messages() which might be costly (otherwise implementing this method is optional).

messages()
Return all messages which are defined by this validator as a key/message dictionary. Alternatively you can create a class-level dictionary which contains these keys/messages.

You must declare all your messages here so that all keys are known after this method was called.

Calling this method might be costly when you have a lot of messages and returning them is expensive. You can reduce the overhead in some situations by implementing `message_for_key()`

revert_conversion(*value*, *context=None*)
Undo the conversion of `process()` and return a "string-like" representation. This method is especially useful for widget libraries like ToscaWigets so they can render Python data types in a human readable way. The returned value does not have to be an actual Python string as long as it has a meaningful unicode() result. Generally the validator should accept the return value in its '.process()' method.

validate(*value*, *context*)
Perform additional checks on the value which was processed successfully before (otherwise this method is not called). Raise an InvalidDataError if the input data is invalid.

You can implement only this method in your validator if you just want to add additional restrictions without touching the actual conversion.

This method must not modify the `converted_value`.

class pycerberus.validators.domain.**DomainNameValidator**(*self*)
Bases: *pycerberus.validators.string.StringValidator*

A validator to check if an domain name is syntactically correct.

keys()
Return all keys defined by this specific validator class.

message_for_key(*key*, *context*)
Return a message for a specific key. Implement this method if you want to avoid calls to messages() which might be costly (otherwise implementing this method is optional).

messages()
Return all messages which are defined by this validator as a key/message dictionary. Alternatively you can create a class-level dictionary which contains these keys/messages.

You must declare all your messages here so that all keys are known after this method was called.

Calling this method might be costly when you have a lot of messages and returning them is expensive. You can reduce the overhead in some situations by implementing `message_for_key()`

validate(*value*, *context*)
Perform additional checks on the value which was processed successfully before (otherwise this method is not called). Raise an InvalidDataError if the input data is invalid.

You can implement only this method in your validator if you just want to add additional restrictions without touching the actual conversion.

This method must not modify the `converted_value`.

class pycerberus.validators.email.**EmailAddressValidator**(*self*)
Bases: *pycerberus.validators.domain.DomainNameValidator*

A validator to check if an email address is syntactically correct.

Please note that there is no clear definition of an 'email address'. Some parts are defined in consecutive RFCs, there is a notion of 'string that is accepted by a MTA' and last but not least a fuzzy 'general expectation' what an email address should be about.

Therefore this validator is currently extremly simple and does not handle internationalized local parts/domains.

For the future I envision some extensions here:

- support internationalized domain names (possibly also encode to/ decode from idna) if specified by flag
- More flexible structure if there must be a second-level domain

Something that should not happen in this validator:

- Open SMTP connections to check if an account exists
- specify default domains if missing

These things can be implemented in derived validators

keys()
Return all keys defined by this specific validator class.

message_for_key(*key*, *context*)
Return a message for a specific key. Implement this method if you want to avoid calls to messages() which might be costly (otherwise implementing this method is optional).

messages()
Return all messages which are defined by this validator as a key/message dictionary. Alternatively you can create a class-level dictionary which contains these keys/messages.

You must declare all your messages here so that all keys are known after this method was called.

Calling this method might be costly when you have a lot of messages and returning them is expensive. You can reduce the overhead in some situations by implementing `message_for_key()`

validate(*emailaddress*, *context*)
Perform additional checks on the value which was processed successfully before (otherwise this method is not called). Raise an InvalidDataError if the input data is invalid.

You can implement only this method in your validator if you just want to add additional restrictions without touching the actual conversion.

This method must not modify the `converted_value`.

class pycerberus.validators.foreach.**ForEach**(*self*)
Bases: *pycerberus.api.Validator*

Apply a validator to every item of an iterable (like map). Also you can specify the allowed min/max number of items in that iterable.

convert(*values*, *context*)
Convert the input value to a suitable Python instance which is returned. If the input is invalid, raise an `InvalidDataError`.

handle_validator_result(*converted_value*, *result*, *context*, *errors=None*, *nr_new_errors=None*)

keys()
Return all keys defined by this specific validator class.

message_for_key(*key*, *context*)
Return a message for a specific key. Implement this method if you want to avoid calls to messages() which might be costly (otherwise implementing this method is optional).

messages()
Return all messages which are defined by this validator as a key/message dictionary. Alternatively you can create a class-level dictionary which contains these keys/messages.

You must declare all your messages here so that all keys are known after this method was called.

Calling this method might be costly when you have a lot of messages and returning them is expensive. You can reduce the overhead in some situations by implementing `message_for_key()`

new_result(*initial_value*)

class pycerberus.validators.string.**StringValidator**(*self*)
Bases: *pycerberus.api.Validator*

convert(*value*, *context*)
Convert the input value to a suitable Python instance which is returned. If the input is invalid, raise an `InvalidDataError`.

is_empty(*value*, *context*)
Decide if the value is considered an empty value.

keys()
Return all keys defined by this specific validator class.

message_for_key(*key*, *context*)
Return a message for a specific key. Implement this method if you want to avoid calls to messages() which might be costly (otherwise implementing this method is optional).

messages()
Return all messages which are defined by this validator as a key/message dictionary. Alternatively you can create a class-level dictionary which contains these keys/messages.

You must declare all your messages here so that all keys are known after this method was called.

Calling this method might be costly when you have a lot of messages and returning them is expensive. You can reduce the overhead in some situations by implementing `message_for_key()`

validate(*value*, *context*)
Perform additional checks on the value which was processed successfully before (otherwise this method is not called). Raise an InvalidDataError if the input data is invalid.

You can implement only this method in your validator if you just want to add additional restrictions without touching the actual conversion.

This method must not modify the `converted_value`.

1.7 Writing your own validators

After all, using only built-in validators won't help you much: You'll need custom validation rules which means that you need to write your own validators.

pycerberus comes with two classes that can serve as a good base when you start writing a custom validator: The `BaseValidator` only provides the absolutely required set of API so you have maximum freedom. The `Validator` class itself is inherited from the `BaseValidator` and defines a more sophisticated API and i18n support. Usually you should use the `Validator` class.

1.7.1 BaseValidator

class pycerberus.api.**BaseValidator**(*self*)

The BaseValidator implements only the minimally required methods. Therefore it does not put many constraints on you. Most users probably want to use the `Validator` class which already implements some commonly used features.

You can pass `messages` a dict of messages during instantiation to overwrite messages specified in the validator without the need to create a subclass.

copy()

Return a copy of this instance.

keys()

Return all keys defined by this specific validator class.

message_for_key(*key*, *context*)

Return a message for a specific key. Implement this method if you want to avoid calls to messages() which might be costly (otherwise implementing this method is optional).

messages()

Return all messages which are defined by this validator as a key/message dictionary. Alternatively you can create a class-level dictionary which contains these keys/messages.

You must declare all your messages here so that all keys are known after this method was called.

Calling this method might be costly when you have a lot of messages and returning them is expensive. You can reduce the overhead in some situations by implementing `message_for_key()`

process(*value*, *context=None*)

This is the method to validate your input. The validator returns a (Python) representation of the given input `value`.

In case of errors a `InvalidDataError` is thrown.

raise_error(*key*, *value*, *context*, *errorclass=<class 'pycerberus.errors.InvalidDataError'>*, ***values*)

Raise an InvalidDataError for the given key.

revert_conversion(*value*, *context=None*)

Undo the conversion of `process()` and return a "string-like" representation. This method is especially useful for widget libraries like ToscaWigets so they can render Python data types in a human readable way. The returned value does not have to be an actual Python string as long as it has a meaningful unicode() result. Generally the validator should accept the return value in its '.process()' method.

1.7.2 Validator

class pycerberus.api.**Validator**(*self*)

The Validator is the base class of most validators and implements some commonly used features like required values (raise exception if no value was provided) or default values in case no value is given.

This validator splits conversion and validation into two separate steps: When a value is `process()``ed, the validator first calls ``convert()` which performs some checks on the value and eventually returns the converted value. Only if the value was converted correctly, the `validate()` function can do additional checks on the converted value and possibly raise an Exception in case of errors. If you only want to do additional checks (but no conversion) in your validator, you can implement `validate()` and simply assume that you get the correct Python type (e.g. int).

Of course if you can also raise a `ValidationError` inside of `convert()` - often errors can only be detected during the conversion process.

By default, a validator will raise an `InvalidDataError` if no value was given (unless you set a default value). If `required` is False, the default is None. All exceptions thrown by validators must be derived from `ValidationError`. Exceptions caused by invalid user input should use `InvalidDataError` or one of the subclasses.

If `strip` is True (default is False) and the input value has a `strip()` method, the input will be stripped before it is tested for empty values and passed to the `convert()`/`validate()` methods.

In order to prevent programmer errors, an exception will be raised if you set `required` to True but provide a default value as well.

convert(*value*, *context*)
Convert the input value to a suitable Python instance which is returned. If the input is invalid, raise an `InvalidDataError`.

empty_value(*context*)
Return the 'empty' value for this validator (usually None).

is_empty(*value*, *context*)
Decide if the value is considered an empty value.

messages()
Return all messages which are defined by this validator as a key/message dictionary. Alternatively you can create a class-level dictionary which contains these keys/messages.

You must declare all your messages here so that all keys are known after this method was called.

Calling this method might be costly when you have a lot of messages and returning them is expensive. You can reduce the overhead in some situations by implementing `message_for_key()`

process(*value*, *context=None*)
This is the method to validate your input. The validator returns a (Python) representation of the given input `value`.

In case of errors a `InvalidDataError` is thrown.

raise_error(*key*, *value*, *context*, *errorclass=<class 'pycerberus.errors.InvalidDataError'>*, *error_dict=None*, *error_list=()*, ***values*)
Raise an InvalidDataError for the given key.

validate(*converted_value*, *context*)
Perform additional checks on the value which was processed successfully before (otherwise this method is not called). Raise an InvalidDataError if the input data is invalid.

You can implement only this method in your validator if you just want to add additional restrictions without touching the actual conversion.

This method must not modify the `converted_value`.

1.7.3 Miscellaneous

pycerberus uses a deprecated library called *simple_super* so you can just say *self.super()* in your custom validator classes. This will call the super implementation with just the same parameters as your method was called.

Validators need to be thread-safe as one instance might be used several times. Therefore you must not add additional attributes to your validator instance after you called Validator's constructor. To prevent unexperienced programmers falling in that trap, a ''Validator" will raise an exception if you try to set an attribute. If you don't like this behavior and you really know what you are doing, you can issue `validator.set_internal_state_freeze(False)` to disable that protection.

1.7.4 Putting all together - A simple validator

Now it's time to put it all together. This validator demonstrates most of the API as explained so far:

```
class UnicodeValidator(Validator):

    def __init__(self, max=None):
        self.super()
        self._max_length = max

    def messages(self):
        return {
                 'invalid_type': _(u'Validator got unexpected input (expected string,␣
↪got %(classname)s).'),
                 'too_long': _(u'Please enter at maximum %(max_length) characters.')
                }
    # Alternatively you could also declare a class-level variable:
    # messages = {...}

    def convert(self, value, context):
        try:
            return unicode(value, 'UTF-8')
        except Exception:
            classname = value.__class__.__name__
            self.raise_error('invalid_type', value, context, classname=classname)

    def validate(self, converted_value, context):
        if self._max_length is None:
            return
        if len(converted_value) > self._max_length:
            self.raise_error('too_long', converted_value, context, max_length=self._
↪max_length)
```

The validator will convert all input to unicode strings (using the UTF-8 encoding). It also checks for a maximum length of the string.

You can see that all the conversion is done in `convert()` while additional validation is encapsulated in `validate()`. This can help you keeping your methods small.

In case there is an error the `error()` method will raise an `InvalidDataError`. You select the error message to show by passing a string constant `key` which identifies the message. The key can be used later to adapt the user interface without relying the message itself (e.g. show an additional help box in the user interface if the user typed in the wrong password).

The error messages are declared in the `messages()`. You'll notice that the message strings can also contain variable parts. You can use these variable parts to give the user some additional hints about what was wrong with the data.

1.8 Internationalization

Modern applications must be able to handle different languages. Internationalization (i18n) in pycerberus refers to validating locale-dependent input data (e.g. different decimal separator characters) as well as validation errors in different languages. The former aspect is not yet covered by default but you should be able to write custom validators easily.

All messages from validators included in pycerberus can be translated in different languages using the standard gettext library. The language of validation error messages will be chosen depending on the locale which is given in the state dictionary,

i18n support in pycerberus is a bit broader than just translating existing error messages. i18n becomes interesting when you write your own validators (based on the ones that come with pycerberus) and your translations need to play along with the built-in ones:

- Translate only the messages you defined, keep the existing pycerberus translations.
- If you don't like the existing pycerberus translations, you can define your own without even changing a single line or file in pycerberus.
- Specify additional translation options per validator class (e.g. a different gettext domain or a different directory where your translations are stored).
- Even though pycerberus uses the well-known gettext mechanism to retrieve translations, you can use any other source as well (e.g. a database or a XML file).

All i18n support in pycerberus aims to provide custom validators with a nice, simple-to-use API while maintaining the flexibility that serious applications need.

1.8.1 Get translated error messages

If you want to get translated error messages from a validator, you set the correct ''context''. formencode looks for a key named 'locale' in the context dictionary:

```
validator = IntegerValidator()
validator.process('foo', context={'locale': 'en'}) # u'Please enter a number.'
validator.process('foo', context={'locale': 'de'}) # u'Bitte geben Sie eine Zahl ein.'
```

1.8.2 Internal gettext details

Usually you don't have to know much about how pycerberus uses gettext internally. Just for completeness: The default domain is 'pycerberus'. By default translations (.mo files) are loaded from `pycerberus.locales`, with a fall back to the system-wide locale dir ''/usr/share/locale''.

1.8.3 Translate your custom messages

To translate messages from a custom validator, you need to declare them in the messages() method and mark the message strings as translatable:

```
from pycerberus.api import Validator
from pycerberus.i18n import _

class MyValidator(Validator):
    def messages(self):
        return {
                 'foo': _('A message.'),
                 'bar': _('Another message.'),
                }

    # your validation logic ...
```

Afterwards you just have to start the usual gettext process. I always use Babel because it provides the very convenient `pybabel` tool which simplifies the workflow a lot:

- Collect the translatable strings in a po template (.pot) file, e.g. `pybabel extract . --output=mymessages.pot`

- Create the initial po file for your new locale (only needed once): `pybabel init --domain=pycerberus --input-file=mymessages.pot --locale=<locale ID> --output-dir=locales/`
- After every change to a translatable string in your source code, you need to recreate the pot file (see first step) and update the po file for your locale: `pybabel update --domain=pycerberus --input-file=mymessages.pot --output-dir=locales/`
- Translate the messages for every locale.
- Compile the final po file into a mo file, e.g. `pybabel compile --domain=pycerberus --directory=locales/`

1.8.4 Override existing messages and translations

Assume your custom validator is a subclass of a built-in validator but you don't like the built-in translation. Of course you can replace pycerberus' mo files directly. However there is also another way where you don't have to change pycerberus itself:

```
class CustomValidatorThatOverridesTranslations(Validator):

    def messages(self):
        return {'empty': _('My custom message if the value is empty'),
                'custom': _('A custom message')}

    # ...
```

This validator will use a different message for the 'empty' error and you can define custom translations for this key in your own .po files.

1.8.5 Modify gettext options (locale dir, domain)

The gettext library is configurable, e.g. in which directory your .mo files are located and which domain (.mo filename) should be used. In pycerberus this is configurable by validator:

```
class ValidatorWithCustomGettextOptions(Validator):

    def messages(self):
        return {'custom': _('A custom message')}

    def translation_parameters(self, context):
        return {'domain': 'myapp', 'localedir': '/home/foo/locale'}

    # ...
```

These translation parameters are passed directly to the ''gettext" call so you can read about the available options in the gettext documentation. Your parameter will be applied for all messages which were declared in your validator class (but not in others). So you can modify the parameters for your own validator but keep all the existing parameters (and translations) for built-in validators.

1.8.6 Retrieve translations from a different source (e.g. database)

Sometimes you don't want to use gettext. For instance you could store translations in a relational database so that your users can update the messages themselves without fiddling with gettext tools:

```
class ValidatorWithNonGettextTranslation(FrameworkValidator):

    def messages(self):
        return {'custom': _('A custom message')}

    def translate_message(self, key, native_message, translation_parameters, context):
        # fetch the translation for 'native_message' from somewhere
        translated_message = get_translation_from_db(native_message)
        return translated_message
```

You can use this mechanism to plug in arbitrary translation systems into gettext. Your translation mechanism is (again) only applied to keys which were defined by your specific validator class. If you want to use your translation system also for keys which were defined by built-in validators, you need to re-define these keys in your class as shown in the previous section.

1.9 Using Validation Schemas

Especially in web development you often get multiple values from a form and you want to validate all these values easily. This is where "compound validators" aka "schemas" come into play. A schema contains multiple validators, one validator for every field. There's nothing special about these validators - they are just validators like the ones I explained in the previous section. Every field validator only cares about a single value and does not see the rest of the values. Also a schema is technically just a special validator.

You can define a schema like this:

```
from pycerberus.schema import SchemaValidator
from pycerberus.validators import IntegerValidator, StringValidator

schema = SchemaValidator()
schema.add('id', IntegerValidator())
schema.add('name', StringValidator())
```

Afterwards the schema behaves most like all basic validators - instead of a single input value they just get a dictionary:

```
validated_values = schema.process({'id': '42', 'name': 'Foo Bar'})
```

If you declared a validator for a key which is not present in the input dict, the validator will get its 'empty' value instead:

```
id_required = SchemaValidator()
id_required.add('id', IntegerValidator(required=False))
id_required.process({}) # -> {'id': None}

id_optional = SchemaValidator()
id_optional.add('id', IntegerValidator(required=True))
id_optional.process({}) # raises an Exception because id None is not acceptable
```

Do not mix up the 'default' value with the 'empty' value:

```
IntegerValidator(default=42)
```

The 'default' value in this case is 42 but the 'empty' value is still None.

Please note that Schemas are 'secure by default' which means that the returned dictionary contains only values that were validated. If you did not add a validator for a specific key, this key won't be included in the result.

If you need to ensure that no values with unknown keys are passed to the schema (even if those would be just dropped), you can specify this when instantiating the schema: `Schema(allow_additional_parameters=False)`. The schema will raise an exception if it finds any unknown keys.

The equivalent "declarative schema" (see next section) is:

```
class MySchema(SchemaValidator):
    allow_additional_parameters=False
    # ...
```

1.9.1 Declarative Schemas

Schemas can be an important part in your application security. Also they define some kind of interface (which parameters does your application expect). Besides the algorithmic way to build a schema there is a 'declarative' way so that you can review and audit your schemas easily:

```
class MySchema(SchemaValidator):
    id   = IntegerValidator()
    name = StringValidator()

# using it...
schema = MySchema()
```

It's absolutely the same schema but the definition is way easier to read.

1.9.2 Schema Error Handling

All schema validators are executed even if one of the previous validators failed. Because of that you can display the user all errors at once:

```
schema = SchemaValidator()
schema.add('id', IntegerValidator())
schema.add('name', StringValidator())
try:
    schema.process({'id': 'invalid', 'name': None})
except InvalidDataError, e:
    e.error_dict()    # {'id': <id validation error>, 'name': <id validation error>}
    e.error_for('id') # id validation error
```

1.9.3 Validating multiple fields in a Schema

Sometimes you need to validate multiple fields in a schema - e.g. you need to check in a 'change password' action that the password is entered the same twice. Or you need to check that a certain value is higher than another value in the form. That's where *formvalidators* come into play.

formvalidators are validators like all other field validators but they get the complete field dict as input, not a single item. Also formvalidators are run *after* all field validators successfully validated the input - therefore you have access to reasonably sane values, already converted to a handy Python data type. Opposite to simple field validators, the validation process fails immediately if one formvalidator fails.

You can add formvalidators to a form like this:

```
class NumbersMatch(Validator):
    def validate(self, fields, context):
        if fields['a'] != fields['b' ]:
            self.raise_error('no_match', fields, context=context)
schema.add_formvalidator(NumbersMatch)
```

Of course there is also a declarative way to use form validators:

```
class MySchema(SchemaValidator):
    # ...
    formvalidators = (NumbersMatch, )
```

1.10 Schema inheritance - build multi-page forms without duplication

Validation schemas are an important piece of information: On the one hand they can serve as a kind of API specification (which parameters are accepted by your application) and on the other hand they are important for security audits (which constraints are put on your input values). Obviously this is something that you want to get right - duplicating this information only increases the likelyhood of bugs.

The issue becomes especially annoying when you have a web application with a complex form (e.g. a new user registration process) that you want to split in multiple steps on different pages so that your users won't drop out immediately when they see the huge form. It is good HTTP/ReST design practice to keep state on the client side. Therefore you pass fields from previous pages in hidden input fields to the next and for the final page it looks like there was one big form. This also has the advantage that you can shuffle the fields on the different pages without changing real logic.

With that approach your pretty much settled - however you need a separate validation schema for every single page which is a huge duplication. With pycerberus you can avoid that by using ''schema inheritance'':

```
class FirstPage(SchemaValidator):
    id = IntegerValidator()

    formvalidators = (SomeValidator(), )

class SecondPage(FirstPage):
    # this schema contains also 'id' validator
    name = StringValidator()

    # formvalidators are implicitely appended so actually this schema has
    # these formvalidators: (SomeValidator(), AnotherValidator(), )
    formvalidators = (AnotherValidator(), )

class FinalPage(SecondPage):
    # this schema contains also 'id' and 'name' validators
    age = IntegerValidator()

    # This page contains again both formvalidators
```

As you can see, every page adds some validators while keeping the old ones. This eliminates the duplication problem described above,

What happens if SecondPage declares a different validator for 'id'? In this case it will just replace the ''IntegerValidator()" declared by ''FirstPage''!

1.11 Parse Input with Schemas

In the common "web form" use case you already get parameters mapped to keys. That's usually the job of your web framework. However sometimes it's not that easy: Before you can do input validation, you need to parse the user input from a string and convert that into a dict.

This is where ''PositionalArgumentsParsingSchema()" might help you: This schema takes a string and extracts several parameters from it. So you can use it to transform `"foo, 42"` into `dict(name="foo", value=42)`.

class pycerberus.schemas.**PositionalArgumentsParsingSchema**(*self*)

This schema parses a string containing arguments within a specified order and returns a dict where each of these parameters is mapped to a specific key for easy retrieval.

You specify the order of parameters (and the keys) in the class-level attribute `parameter_order`:

```
class ConfigListSchema(PositionalArgumentsParsingSchema):
    first_key = StringValidator()
    second_key = IntegerValidator()
    parameter_order = (first_key, second_key)
```

By default the items are separated by comma though you can override in the method `separator_pattern()`. If there are more items than keys specified, this schema will behave like any other schema (depending if you set the class-level attribute `allow_additional_parameters`).

aggregate_values(*parameter_names*, *arguments*, *context*)

This method can manipulate or aggregate parsed arguments. In this class, it's just a noop but sub classes can override this method to do more interesting stuff.

keys()

Return all keys defined by this specific validator class.

message_for_key(*key*, *context*)

Return a message for a specific key. Implement this method if you want to avoid calls to messages() which might be costly (otherwise implementing this method is optional).

messages()

Return all messages which are defined by this validator as a key/message dictionary. Alternatively you can create a class-level dictionary which contains these keys/messages.

You must declare all your messages here so that all keys are known after this method was called.

Calling this method might be costly when you have a lot of messages and returning them is expensive. You can reduce the overhead in some situations by implementing `message_for_key()`

process(*value*, *context=None*)

This is the method to validate your input. The validator returns a (Python) representation of the given input `value`.

In case of errors a `InvalidDataError` is thrown.

This schema is used for example in pymta. to parse the SMTP command strings. Also I used it in my OhlohWidgets-Macro: Trac macros can get parameters but these are passed as a single string so the schema takes care of separating these arguments.

1.12 Support for Python 3

pycerberus supports Python 3 out of the box (tested with Python 3.4+) as well as Python 2.7. pycerberus 0.6 uses a unified source tree (pycerberus 0.5 relied on 2to3 and some custom hacks).

As far as I am aware the Python 3 version of pycerberus behaves exactly like the Python 2 version.

CHAPTER 2

Getting Help

There is a mailing list where you ask for help or discuss new features

CHAPTER 3

License

pycerberus is licensed under the MIT license. As there are no other dependencies (besides Python itself), you can easily use pycerberus in proprietary as well as GPL applications.

Python Module Index

p

Index

N

P

R

S

V