peewee-validates

Release 1.0.8

Oct 25, 2019

Contents

A simple and flexible model and data validator for Peewee ORM.

CHAPTER 1

Requirements

- python >= 3.3
- peewee >= 2.8.2 (including Peewee 3)
- python-dateutil >= 2.5.0

CHAPTER 2

Installation

This package can be installed using pip:

```
pip install peewee-validates
```

CHAPTER 3

Usage

Here's a quick teaser of what you can do with peewee-validates:

```
import peewee
from peewee_validates import ModelValidator

class Category(peewee.Model):
    code = peewee.IntegerField(unique=True)
    name = peewee.CharField(null=False, max_length=250)

obj = Category(code=42)

validator = ModelValidator(obj)
validator.validate()

print(validator.errors)

# {'name': 'This field is required.', 'code': 'Must be a unique value.'}
```

In fact, there is also a generic validator that does not even require a model:

```
from peewee_validates import Validator, StringField

class SimpleValidator(Validator):
    name = StringField(required=True, max_length=250)
    code = StringField(required=True, max_length=4)

validator = SimpleValidator(obj)
validator.validate({'code': 'toolong'})

print(validator.errors)

# {'name': 'This field is required.', 'code': 'Must be at most 5 characters.'}
```

CHAPTER 4

Documentation

Check out the Full Documentation for more details.

CHAPTER 5

Contents

5.1 Basic Validation

The very basic usage is a validator class that looks like this:

```
from peewee_validates import Validator, StringField, validate_not_empty

class SimpleValidator(Validator):
    first_name = StringField(validators=[validate_not_empty()])

validator = SimpleValidator()
```

This tells us that we want to validate data for one field (first_name).

Each field has an associated data type. In this case, using StringField will coerce the input data to `str`.

After creating an instance of our valitdator, then we call the `validate()` method and pass the data that we want to validate. The result we get back is a boolean indicating whether all validations were successful.

The validator then has two dictionaries that you mway want to access: `data` and `errors`.

`data` is the input data that may have been mutated after validations.

`errors` is a dictionary of any error messages.

```
data = {'first_name': ''}
validator.validate(data)

print(validator.data)
# {}

print(validator.errors)
# {'first_name': 'This field is required'}
```

In this case we can see that there was one error for `first_name`. That's because we gave it the `validate_not_empty()` validator but did not pass any data for that field. Also notice that the `data` dict is

empty because the validators did not pass.

When we pass data that matches all validators, the `errors` dict will be empty and the `data` dict will be populated:

```
data = {'first_name': 'Tim'}
validator.validate(data)

print(validator.data)
# {'first_name': 'Tim'}

print(validator.errors)
# {}
```

The `data` dict will contain the values after any validators, type coersions, and any other custom modifiers. Also notice that we are able to reuse the same validator instance while passing a new data dict.

5.1.1 Data Type Coersion

One of the first processes that happens when data validation takes place is data type coersion.

There are a number of different fields built-in. Check out the full list in the API Documentation.

Here's an example of a field. This just duplicates the functionality of `IntegerField` to show you an as example.

```
class CustomIntegerField(Field):
    def coerce(self, value):
        try:
            return int(value)
        except (TypeError, ValueError):
            raise ValidationError('coerce_int')

class SimpleValidator(Validator):
    code = CustomIntegerField()

validator = SimpleValidator()
validator.validate({'code': 'text'})

validator.data
# {}

validator.errors
# {'code': 'Must be a valid integer.'}
```

5.2 Available Validators

There are a bunch of built-in validators that can be accessed by importing from `peewee_validates`.

- `validate_email()` - validate that data is an email address
- `validate_equal(value)` - validate that data is equal to `value`
- `validate_function(method, **kwargs)` - runs `method` with field value as first argument and `kwargs` and alidates that the result is truthy
- `validate_length(low, high, equal)` - validate that length is between `low` and `high` or equal to `equal`

- `validate_none_of(values)` - validate that value is not in `values`. `values` can also be a callable that returns values when called
- `validate_not_empty()` - validate that data is not empty
- `validate_one_of(values)` - validate that value is in `values`. `values` can also be a callable that returns values when called
- `validate_range(low, high)` - validate that value is between `low` and `high`
- `validate_regexp(pattern, flags=0)` - validate that value matches `patten`
- `validate_required()` - validate that the field is present

5.3 Custom Validators

A field validator is just a method with the signature `validator(field, data)` where field is a `Field` instance and `data` is the data dict that is passed to `validate()`.

If we want to implement a validator that makes sure the name is always "tim" we could do it like this:

```
def always_tim(field, data):
    if field.value and field.value != 'tim':
        raise ValidationError('not_tim')

class SimpleValidator(Validator):
    name = StringField(validators=[always_tim])

validator = SimpleValidator()
validator.validate({'name': 'bob'})

validator.errors
# {'name': 'Validation failed.'}
```

That's not a very pretty error message, but I'll show you soon how to customize that.

Now let's say you want to implement a validator that checks the length of the field. The length should be configurable. So we can implement a validator that accepts a parameter and returns a validator function. We basically wrap our actual validator function with another function. That looks like this:

```
def length(max_length):
    def validator(field, data):
        if field.value and len(field.value) > max_length:
            raise ValidationError('too_long')
    return validator

class SimpleValidator(Validator):
    name = StringField(validators=[length(2)])

validator = SimpleValidator()
validator.validate({'name': 'bob'})

validator.errors
# {'name': 'Validation failed.'}
```

5.4 Custom Error Messages

In some of the previous examples, we saw that the default error messages are not always that friendly. Error messages can be changed by settings the `messages` attribute on the `Meta` class. Error messages are looked up by a key, and optionally prefixed with the field name.

The key is the first argument passed to `ValidationError` when an error is raised.

```
class SimpleValidator(Validator):
    name = StringField(required=True)

    class Meta:
        messages = {
            'required': 'Please enter a value.'
        }
```

Now any field that is required will have the error message "please enter a value". We can also change this for specific fields by prefixing with field name:

```
class SimpleValidator(Validator):
    name = StringField(required=True)
    color = StringField(required=True)

    class Meta:
        messages = {
            'name.required': 'Enter your name.',
            'required': 'Please enter a value.',
        }
```

Now the `name` field will have the error message "Enter your name." but all other required fields will use the other error message.

5.5 Excluding/Limiting Fields

It's possible to limit or exclude fields from validation. This can be done at the class level or when calling `validate()`.

This will only validate the `name` and `color` fields when `validate()` is called:

```
class SimpleValidator(Validator):
    name = StringField(required=True)
    color = StringField(required=True)
    age = IntegerField(required=True)

    class Meta:
        only = ('name', 'color')
```

And similarly, you can override this when `validate()` is called:

```
validator = SimpleValidator()
validator.validate(data, only=('color', 'name'))
```

Now only `color` and `name` will be validated, ignoring the definition on the class.

There's also an `exclude` attribute to exclude specific fields from validation. It works the same way that `only` does.

5.6 Model Validation

You may be wondering why this package is called peewee-validates when nothing we have discussed so far has anything to do with Peewee. Well here is where you find out. This package includes a ModelValidator class for using the validators we already talked about to validate model instances.

```
import peewee
from peewee_validates import ModelValidator

class Category(peewee.Model):
    code = peewee.IntegerField(unique=True)
    name = peewee.CharField(max_length=250)

obj = Category(code=42)

validator = ModelValidator(obj)
validator.validate()
```

In this case, the ModelValidator has built a Validator class that looks like this:

```
unique_code_validator = validate_model_unique(
    Category.code, Category.select(), pk_field=Category.id, pk_value=obj.id)

class CategoryValidator(Validator):
    code = peewee.IntegerField(
        required=True,
        validators=[unique_code_validator])
    name = peewee.StringField(required=True, max_length=250)
```

Notice the many things that have been defined in our model that have been automatically converted to validator attributes:

- name is required string
- name must be 250 character or less
- code is required integer
- code must be a unique value in the table

We can then use the validator to validate data.

By default, it will validate the data directly on the model instance, but you can always pass a dictionary to `validates` that will override any data on the instance.

```
obj = Category(code=42)
data = {'code': 'notnum'}

validator = ModelValidator(obj)
validator.validate(data)

validator.errors
# {'code': 'Must be a valid integer.'}
```

This fails validation because the data passed in was not a number, even though the data on the instance was valid.

You can also create a subclass of `ModelValidator` to use all the other things we have shown already:

```
import peewee
from peewee_validates import ModelValidator

class CategoryValidator(ModelValidator):
    class Meta:
        messages = {
            'name.required': 'Enter your name.',
            'required': 'Please enter a value.',
        }

validator = ModelValidator(obj)
validator.validate(data)
```

When validations is successful for ModelValidator, the given model instance will have been mutated.

```
validator = ModelValidator(obj)

obj.name
# 'tim'

validator.validate({'name': 'newname'})

obj.name
# 'newname'
```

5.6.1 Field Validations

Using the ModelValidator provides a couple extra goodies that are not found in the standard Validator class.

Uniqueness

If the Peewee field was defined with `unique=True` then a validator will be added to the field that will look up the value in the database to make sure it's unique. This is smart enough to know to exclude the current instance if it has already been saved to the database.

Foreign Key

If the Peewee field is a `ForeignKeyField` then a validator will be added to the field that will look up the value in the related table to make sure it's a valid instance.

Many to Many

If the Peewee field is a `ManyToManyField` then a validator will be added to the field that will look up the values in the related table to make sure it's valid list of instances.

Index Validation

If you have defined unique indexes on the model like the example below, they will also be validated (after all the other field level validations have succeeded).

```
class Category(peewee.Model):
    code = peewee.IntegerField(unique=True)
    name = peewee.CharField(max_length=250)

    class Meta:
        indexes = (
            (('name', 'code'), True),
        )
```

5.7 Field Overrides

If you need to change the way a model field is validated, you can simply override the field in your custom class. Given the following model:

```
class Category(peewee.Model):
    code = peewee.IntegerField(required=True)
```

This would generate a field for `code` with a required validator.

```
class CategoryValidator(ModelValidator):
    code = IntegerField(required=False)

validator = CategoryValidator(category)
validator.validate()
```

Now `code` will not be required when the call to `validate` happens.

5.8 Overriding Behaviors

5.8.1 Cleaning

Once all field-level data has been validated during `validate()`, the resulting data is passed to the `clean()` method before being returned in the result. You can override this method to perform any validations you like, or mutate the data before returning it.

```
class MyValidator(Validator):
    name1 = StringField()
    name2 = StringField()

    def clean(self, data):
        # make sure name1 is the same as name2
        if data['name1'] != data['name2']:
            raise ValidationError('name_different')
        # and if they are the same, uppercase them
        data['name1'] = data['name1'].upper()
        data['name2'] = data['name2'].upper()
        return data

    class Meta:
        messages = {
            'name_different': 'The names should be the same.'
        }
```

5.8.2 Adding Fields Dynamically

If you need to, you can dynamically add a field to a validator instance. They are stored in the `_meta.fields` dict, which you can manipulate as much as you want.

```
validator = MyValidator()
validator._meta.fields['newfield'] = IntegerField(required=True)
```

5.9 API Documentation

5.9.1 Validator Classes

class Validator

A validator class. Can have many fields attached to it to perform validation on data.

class Meta

A meta class to specify options for the validator. Uses the following fields:

```
messages = {}

only = []

exclues = []
```

clean(*data*)

Clean the data dictionary and return the cleaned values.

Parameters **data** – Dictionary of data for all fields.

Returns Dictionary of "clean" values.

Return type dict

clean_fields(*data*)

For each field, check to see if there is a clean_<name> method. If so, run that method and set the returned value on the self.data dict. This happens after all validations so that each field can act on the cleaned data of other fields if needed.

Parameters **data** – Dictionary of data to clean.

Returns None

initialize_fields()

The dict self.base_fields is a model instance at this point. Turn it into an instance attribute on this meta class. Also intitialize any other special fields if needed in sub-classes.

Returns None

validate(*data=None*, *only=None*, *exclude=None*)

Validate the data for all fields and return whether the validation was successful. This method also retains the validated data in `self.data` so that it can be accessed later.

This is usually the method you want to call after creating the validator instance.

Parameters

- **data** – Dictionary of data to validate.
- **only** – List or tuple of fields to validate.
- **exclude** – List or tuple of fields to exclude from validation.

Returns True if validation was successful. Otherwise False.

class ModelValidator(*instance*)

A validator class based on a Peewee model instance. Fields are automatically added based on the model instance, but can be customized.

Parameters **instance** – Peewee model instance to use for data lookups and field generation.

convert_field(*name*, *field*)

Convert a single field from a Peewee model field to a validator field.

Parameters

- **name** – Name of the field as defined on this validator.
- **name** – Peewee field instance.

Returns Validator field.

initialize_fields()
Convert all model fields to validator fields. Then call the parent so that overwrites can happen if necessary for manually defined fields.

Returns None

perform_index_validation(*data*)
Validate any unique indexes specified on the model. This should happen after all the normal fields have been validated. This can add error messages to multiple fields.

Returns None

save(*force_insert=False*)
Save the model and any related many-to-many fields.

Parameters **force_insert** – Should the save force an insert?

Returns Number of rows impacted, or False.

validate(*data=None*, *only=None*, *exclude=None*)
Validate the data for all fields and return whether the validation was successful. This method also retains the validated data in `self.data` so that it can be accessed later.

If data for a field is not provided in `data` then this validator will check against the provided model instance.

This is usually the method you want to call after creating the validator instance.

Parameters

- **data** – Dictionary of data to validate.
- **only** – List or tuple of fields to validate.
- **exclude** – List or tuple of fields to exclude from validation.

Returns True if validation is successful, otherwise False.

5.9.2 Fields

class Field(*required=False*, *default=None*, *validators=None*)
Base class from which all other fields should be derived.

Parameters

- **default** – Is this field required?
- **default** – Default value to be used if no incoming value is provided.
- **validators** – List of validator functions to run.

class StringField(*required=False*, *max_length=None*, *min_length=None*, *default=None*, *validators=None*)
A field that will try to coerce value to a string.

Parameters

- **required** – Is this field required?

- **default** – Default value to be used if no incoming value is provided.
- **validators** – List of validator functions to run.
- **max_length** – Maximum length that should be enfocred.
- **min_length** – Minimum length that should be enfocred.

class FloatField(*required=False*, *low=None*, *high=None*, *default=None*, *validators=None*)
A field that will try to coerce value to a float.

Parameters

- **required** – Is this field required?
- **default** – Default value to be used if no incoming value is provided.
- **validators** – List of validator functions to run.
- **low** – Lowest value that should be enfocred.
- **high** – Highest value that should be enfocred.

class IntegerField(*required=False*, *low=None*, *high=None*, *default=None*, *validators=None*)
A field that will try to coerce value to an integer.

Parameters

- **required** – Is this field required?
- **default** – Default value to be used if no incoming value is provided.
- **validators** – List of validator functions to run.
- **low** – Lowest value that should be enfocred.
- **high** – Highest value that should be enfocred.

class DecimalField(*required=False*, *low=None*, *high=None*, *default=None*, *validators=None*)
A field that will try to coerce value to a decimal.

Parameters

- **required** – Is this field required?
- **default** – Default value to be used if no incoming value is provided.
- **validators** – List of validator functions to run.
- **low** – Lowest value that should be enfocred.
- **high** – Highest value that should be enfocred.

class DateField(*required=False*, *low=None*, *high=None*, *default=None*, *validators=None*)
A field that will try to coerce value to a date. Can accept a date object, string, or anything else that can be converted by *dateutil.parser.parse*.

Parameters

- **required** – Is this field required?
- **default** – Default value to be used if no incoming value is provided.
- **validators** – List of validator functions to run.
- **low** – Lowest value that should be enfocred.
- **high** – Highest value that should be enfocred.

class TimeField(*required=False*, *low=None*, *high=None*, *default=None*, *validators=None*)
A field that will try to coerce value to a time. Can accept a time object, string, or anything else that can be converted by *dateutil.parser.parse*.

Parameters

- **required** – Is this field required?
- **default** – Default value to be used if no incoming value is provided.
- **validators** – List of validator functions to run.
- **low** – Lowest value that should be enfocred.
- **high** – Highest value that should be enfocred.

class DateTimeField(*required=False*, *low=None*, *high=None*, *default=None*, *validators=None*)
A field that will try to coerce value to a datetime. Can accept a datetime object, string, or anything else that can be converted by *dateutil.parser.parse*.

Parameters

- **required** – Is this field required?
- **default** – Default value to be used if no incoming value is provided.
- **validators** – List of validator functions to run.
- **low** – Lowest value that should be enfocred.
- **high** – Highest value that should be enfocred.

class BooleanField(*required=False*, *default=None*, *validators=None*)
A field that will try to coerce value to a boolean. By default the values is converted to string first, then compared to these values: values which are considered False: ('0', '{}', 'none', 'false') And everything else is True.

class ModelChoiceField(*query*, *lookup_field*, *required=False*, ***kwargs*)
A field that allows for a single value based on a model query and lookup field.

Parameters

- **query** – Query to use for lookup.
- **lookup_field** – Field that will be queried for the value.

class ManyModelChoiceField(*query*, *lookup_field*, *required=False*, ***kwargs*)
A field that allows for multiple values based on a model query and lookup field.

Parameters

- **query** – Query to use for lookup.
- **lookup_field** – Field that will be queried for the value.

5.9.3 Field Validators

This module includes some basic validators, but it's pretty easy to write your own if needed.

All validators return a function that can be used to validate a field. For example:

```
validator = peewee_validates.validate_required()

# Raises ValidationError since no data was provided for this field.
field = StringField()
```

(continues on next page)

(continued from previous page)

```
validator(field, {})

# Does not raise any error since default data was provided.
field = StringField(default='something')
validator(field, {})
```

validate_email()
Validate the field is a valid email address.

Raises `ValidationError('email')`

validate_equal(*value*)
Validate the field value is equal to the given value. Should work with anything that supports '==' operator.

Parameters **value** – Value to compare.

Raises `ValidationError('equal')`

validate_function(*method*, ***kwargs*)
Validate the field matches the result of calling the given method. Example:

```
def myfunc(value, name):
    return value == name

validator = validate_function(myfunc, name='tim')
```

Essentially creates a validator that only accepts the name 'tim'.

Parameters

- **method** – Method to call.
- **kwargs** – Additional keyword arguments passed to the method.

Raises `ValidationError('function')`

validate_length(*low=None*, *high=None*, *equal=None*)
Validate the length of a field with either low, high, or equal. Should work with anything that supports len().

Parameters

- **low** – Smallest length required.
- **high** – Longest length required.
- **equal** – Exact length required.

Raises `ValidationError('length_low')`

Raises `ValidationError('length_high')`

Raises `ValidationError('length_between')`

Raises `ValidationError('length_equal')`

validate_matches(*other*)
Validate the field value is equal to another field in the data. Should work with anything that supports '==' operator.

Parameters **value** – Field key to compare.

Raises `ValidationError('matches')`

validate_model_unique(*lookup_field*, *queryset*, *pk_field=None*, *pk_value=None*)
Validate the field is a unique, given a queryset and lookup_field. Example:

```
validator = validate_model_unique(User.email, User.select())
```

Creates a validator that can validate the uniqueness of an email address.

Parameters

- **lookup_field** – Peewee model field that should be used for checking existing values.
- **queryset** – Queryset to use for lookup.
- **pk_field** – Field instance to use when excluding existing instance.
- **pk_value** – Field value to use when excluding existing instance.

Raises `ValidationError('unique')`

validate_none_of(*values*)
Validate that a field is not in one of the given values.

Parameters **values** – Iterable of invalid values.

Raises `ValidationError('none_of')`

validate_not_empty()
Validate that a field is not empty (blank string).

Raises `ValidationError('empty')`

validate_one_of(*values*)
Validate that a field is in one of the given values.

Parameters **values** – Iterable of valid values.

Raises `ValidationError('one_of')`

validate_range(*low=None*, *high=None*)
Validate the range of a field with either low, high, or equal. Should work with anything that supports '>' and '<' operators.

Parameters

- **low** – Smallest value required.
- **high** – Longest value required.

Raises `ValidationError('range_low')`

Raises `ValidationError('range_high')`

Raises `ValidationError('range_between')`

validate_regexp(*pattern*, *flags=0*)
Validate the field matches the given regular expression. Should work with anything that supports '==' operator.

Parameters

- **pattern** – Regular expresion to match. String or regular expression instance.
- **pattern** – Flags for the regular expression.

Raises `ValidationError('equal')`

validate_required()
Validate that a field is present in the data.

Raises `ValidationError('required')`

5.10 Changelog

5.10.1 1.0.7

- Fix compatibility with Peewee 3

5.10.2 1.0.6

- Make compatible with Peewee 3

5.10.3 1.0.5

- Fixed issue with converting DecimalField.
- Fixed issue with converting IntegerField.
- Fixed issue with using a list of non-string in one_of_validator.

5.10.4 1.0.4

- Fixed issue with ModelValidator field overrides.

5.10.5 1.0.3

- Fixed issue coercing numeric fields.

5.10.6 1.0.2

- Fixed issue when passing dicts and validating unique indexes.

5.10.7 1.0.1

- Fixed issue where `None` values were failing validators.

5.10.8 1.0.0

- Major rewrite and first release with proper documentation.

CHAPTER 6

License

The MIT License (MIT)

Copyright (c) 2016 Tim Shaffer

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Index