flyforms Documentation

Release 1.0.0b1

Pavel Sizov

October 29, 2015

Contents

FlyForms is flexible and easy to use Python library that provide high-level API for data structures defining and validation. There is nothing superfluous, so it is lightweight and very fast.

CHAPTER 1

Installation

The easiest way to install FlyForms is using pip:

```
pip install flyforms
```

You can install FlyForms using setup.py also:

```
python setup.py install
```

CHAPTER 2

Feedback

Found a bug, or you have a suggestion for improvement? Please contact me at BitBucket

2.1 Introduction to FlyForms

2.1.1 Concept

There are main concepts of FlyForms. It based on

- **Validators** that check certain properties of the obtained data.
- **Fields** represent a set of rules to data validation via set of *Validator* instances.
- **Form** is the core container of FlyForms. Forms represent a collection of *Field* instances.

2.1.2 Quickstart

Defining Forms

Let's define our first form right away:

```
from flyforms.form import Form
from flyforms.fields import EmailField, StringField


class LoginForm(Form):
    email = EmailField()
    password = StringField(
        min_length=8,
        regex=r"^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#\$%\^&\*])",
        max_length=64
    )
```

When you create a form, you define the fields by the defining class variables for *Form* subclass which are instantiations of the fields. In this example, we have defined the authorization form consists of two fields which represent user email and password.

Extending Forms

As a normal Python object Forms have inheritance. So, if you need to extend your form you can easily do it:

```
class RegistrationForm(LoginForm):

    first_name = StringField(
        regex=r"^[A-Z].*$",
        min_length=3,
        max_length=64
    )

    last_name = StringField(
        regex=r"^[A-Z].*$",
        min_length=3,
        max_length=64
    )
```

Via subclassing, `RegistrationForm` has all fields defined in `LoginForm` and it's own. So you easily share common fields between forms.

Using Forms

Using a Forms is as simple as their definition. Let's see an usage example for `LoginForm` we defined earlier:

```
    form = LoginForm(
        email="qwerty@gmail.com",
        password="Qwerty_#123"
    )

    print(form.is_valid)   # >>> True
    print(form.errors)   # >>> {}

    print(form.password)   # >>> Qwerty_#123
    print(form.email)   # >>> qwerty@gmail.com
```

First, we instantiate the *Form*, providing it with data. While instantiation given data pass validation using defined fields validators. By the way, all the fields in the form are required, by default. You need to pass *required=False* to field's constructor if you want to discard it.

If the *Form* is submitted with wrong data, we get the following:

```
# coding=utf-8

from flyforms.form import Form
from flyforms.fields import EmailField, StringField


class LoginForm(Form):
    email = EmailField()
    password = StringField(
        min_length=8,
        regex=r"^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#\$%\^&\*])",
        max_length=64
    )


class RegistrationForm(LoginForm):
```

```
    first_name = StringField(
        regex=r"^[A-Z].*$",
        min_length=3,
        max_length=64
    )

    last_name = StringField(
        regex=r"^[A-Z].*$",
        min_length=3,
        max_length=64
    )


if __name__ == '__main__':
    form = LoginForm(
        email="qwerty@gmail.com",
        password="Qwerty_#123"
    )

    print(form.is_valid)  # >>> True
    print(form.errors)  # >>> {}

    print(form.password)  # >>> Qwerty_#123
    print(form.email)  # >>> qwerty@gmail.com
```

2.1.3 Further reading

For more information about FlyForms see *FlyForms reference*.

2.2 FlyForms reference

2.2.1 Forms

Class *Form* is the root class of FlyForms that provide the highest level API for data structures validation and mapping. All user-defined forms must inherit this class.

Form class

class flyforms.core.**Form**(***data*)

The root class for all Forms

Parameters **data** (*dict*) – additional data to form

When a Form is instantiated you can access given data via instance attributes or get everything at once using *to_python()* method

Properties

is_bound

Checks is Form instance bound. Returns True if there are no *UnboundField* instances in *_raw_data*. Otherwise, False.

is_valid

Checks is Form instance valid. Returns True if there are no errors. Otherwise, False.

Attributes

_raw_data
Normal Python `dict` contains all Form data (even unbind fields)

_fields
Python `set` contains all defined fields names

_meta
Instance of *FormMetaOptions* contains Form meta information

Methods

to_python()
Changed in version 1.0.0.

New in version 0.3.0.

Get form data as a `dict`

Returns dictionary that contains bound data of valid form

Raises UnboundForm if *is_valid* returns False

classmethod **validate**(***schema*)
Class method provides data validation without *Form* instantiation by calling *Field.validate()* method of all declared fields

Parameters **schema** – data to validate

Returns boolean flag is data valid for this *Form*

For more information about *validate()* method see *In flight data validation*.

Defining Forms

Forms defining is quite simply process. All you need is to make a subclass of *Form* and define fields as class attributes. If you need to extend Forms, inheritance is available. New Form will contain all fields of the parent form as well as it's own.

```
from flyforms import Form, IntField


class GrandParent(Form):
    grandparent_field = IntField()


class Parent(GrandParent):
    parent_field = IntField()


class Child(Parent):
    child_field = IntField()


if __name__ == '__main__':
    print(Child._fields)  # >> set(['parent_field', 'child_field', 'grandparent_field'])
```

By the way, multiple Forms inheritance is also possible. It will be done in full compliance with the Python MRO:

```
from flyforms import Form, IntField, EmailField


class GrandParent(Form):
    grandparent = IntField()


class Mother(GrandParent):
    mother = IntField()
    parent = EmailField()


class Father(GrandParent):
    father = IntField()
    parent = IntField()


class Child(Mother, Father):
    child = IntField()


if __name__ == '__main__':
    print(Child._fields)  # >> set(['parent', 'mother', 'grandparent', 'child', 'father'])
    print(Child.parent)  # >> <flyforms.fields.EmailField object at ...>
```

Using Forms

Typical Forms usage schema looks like:

```
class MyForm(Form):
    # Your form definition

if __name__ == '__main__':
    f = MyForm(**data)  # form instantiation
    if f.is_valid:
        # do something with form
    else:
        # handle form errors
```

Note: Verification form by `is_valid` is a more generalized than `is_bound`. In general, Form may be *bound* but *not valid*. Therefore, we recommend you to use `is_valid` property for Form bound verification.

For a better understanding of the internal structure of the `Form` class see *Low-level API* section.

When Form instantiated, you can access bind data within form instance attributes:

```
class MyForm(Form):
    field = StringField()

if __name__ == '__main__':
    f = MyForm(**data)  # form instantiation
    print(f.field)  # >> bound value will be printed
```

FlyForms `Field` API allow you to set default values for form fields. You can use it together with passing `required=False` to field constructor (for more information about `Field` API see *Fields*). If you do not pass the value of not required field during form instantiation, you'll got the default value:

```
class MyForm(Form):
    field = StringField(required=False, default="Hello!")

if __name__ == '__main__':
    f = MyForm()  # form instantiation
    print(f.field)  # >> Hello!
```

But if you'll pass `required=False` to field constructor without passing a default value, you'll got an unbound field:

```
class MyForm(Form):
    field = StringField(required=False)

if __name__ == '__main__':
    f = MyForm()  # form instantiation
    print(f.field)  # >> <UnboundStringField(field, errors: {})>
```

By default, representation of unbound fields provided by *UnboundField*.

If you want to access all bound data, use *to_python()* method that returns `dict` which contains all bound data. But if there are any errors, *UnboundForm* exception will be raised.

class `flyforms.core.`**UnboundForm**
Raises when you try to serialize an *Form* bound with invalid data

FlyForms also provides you to dump Form data into json string within the `to_json()` method.

In flight data validation

If you already have defined Form and you want to just validate a data structure within that, you can use *Form.validate()* class method.

Goal

The main goal is that no new objects will be created when you call *Form.validate()*.

Usage

```
# coding=utf-8

from flyforms.form import Form
from flyforms.fields import EmailField, StringField


class LoginForm(Form):
    email = EmailField()
    password = StringField(
        min_length=8,
        regex=r"^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#\$%\^&\*])",
        max_length=64
    )


if __name__ == '__main__':

    # Valid data
    r1 = LoginForm.validate(email="smith@gmail.com", password="Qw3rT!y_")
    print(r1)  # >>>  True
```

```
    # Bad data
    r2 = LoginForm.validate(email="smith@gmail.com", password="Qwerty")
    print(r2)  # >>>  False
```

Unbound fields rendering

By default, all unbound fields are replaced with *UnboundField* instances. You can customize it using *FormMetaOptions.unbound_field_render* in *FormMetaOptions* definition for your Form.

class flyforms.core.**UnboundField**(*name*, *field_type*, *errors*)
Field without value

Parameters

- **name** – field name
- **field_type** – field class
- **errors** – field errors

Forms customization

You can define an Meta class in your Form definition to customize it's behaviour.

API

class flyforms.core.**FormMetaOptions**(***kwargs*)
Provides customization of *Form* instances behaviour.

skip_extra
(default: False) if not defined, all extra field will be interpreted as errors during Form instantiation

unbound_field_render
(default: *UnboundField*) all unbound Form fields will be replaced with instances of this class

Usage

```
# coding=utf-8
from flyforms import Form, IntField


class MyUnboundField(object):

    def __init__(self, *args, **kwargs):
        pass


class MyForm(Form):

    class Meta:
        skip_extra = True
        unbound_field_render = MyUnboundField

    field = IntField()
    not_required = IntField(required=False)


if __name__ == '__main__':
    f = MyForm(**{
```

```
        "field": 1,
        "extra_field1": None,
        "extra_field2": object()
    })

    print(f.is_valid)  # >> True
    print(f.to_python())  # >> {'field': 1}
    print(f.not_required)  # >> <__main__.MyUnboundField object at ...>
```

Note: There is no inheritance of `Meta` class. It'll have an effect only in a form where it has been defined. **But** if you use *EmbeddedFormField* with form in which the `Meta` class defined, it *FormMetaOptions.skip_extra* attribute will be used to customize it binding process.

Low-level API

Warning: This section provides information about core classes API of FlyForms. You should not use it, but understanding. It is necessary to extending and form behavior customization in some specific cases.

class `flyforms.core.`**FormMeta**
The metaclass for Form and it's subclasses. It's main responsibility - find all declared fields in the form and it's bases. It also replaces the declared fields with *FormField* descriptor.

class `flyforms.core.`**FormField**(*name*, *field_obj*)
The descriptor for fields in *Form*. It's behavior depends on whether the form is instantiated or not.

Parameters

- **name** – declared field class attribute name
- **field_obj** – instance of declared field

__get__(*instance*, *owner*)
If form is instantiated returns bound data, otherwise - instance of declared field

__set__(*instance*, *value*)
Calls *Field.bind()* method and puts the result to *Form._raw_data*.

If *Field.bind()* returns `UNSET` value or there are errors (second return value is not `None`) an instance of *UnboundField* will be put into *Form._raw_data*.

If form is instantiated `AttributeError` will be raised.

Form data manipulations

Since version 1.0.0 FlyForms provides you to load and dump your Form data to JSON format. We decided to bring this functionality into separate functions, collected in module *flyforms.form*. For JSON encoding and decoding we use `json` module. Eventually, the data cached in Form constitute an ordinary Python `dict`, so we decided to avoid complicating.

`flyforms.form.`**to_json**(*form*, ***kwargs*)
New in version 1.0.0.

Dump Form data to json string

Parameters

- **form** – instance of *Form* subclass

- **kwargs** – additional arguments passed to json.dumps

Returns encoded json-string

`flyforms.form.`**`from_json`**(*form_cls*, *json_str*, ***kwargs*)

Creates and returns new Form instance bound with data from passed json string

Parameters

- **form_cls** – *Form* subclass
- **json_str** – json string
- **kwargs** – additional arguments for `json.loads`

Returns bound Form instantiated from `form_cls`

`flyforms.form.`**`validate_json`**(*form_cls*, *json_str*, ***kwargs*)

Validates given json string data within passed *Form* subclass by calling it's *Form.validate()* classmethod.

It is useful when you don't need to load data from json to your Form.

Parameters

- **form_cls** – *Form* subclass
- **json_str** – json string
- **kwargs** – additional arguments for `json.loads`

Returns bound Form instantiated from `form_cls`

2.2.2 Fields

Fields represent a set of rules for data validation within set of *Validator* instances. When you define your custom *Form* subclass you define Fields as its class attributes. This is the most common usage of Fields, but nothing prevents you to use them standalone.

Field class

class `flyforms.fields.`**`Field`**(*required=True*, *null=False*, *validators=()*, *default=UNSET*)

This is the base class for all builtin and user defined Fields.

Parameters

- **required** (*bool*) – boolean flag is this field value may be `UNSET`
- **null** (*bool*) – boolean flag is field value may be `None`
- **validators** (*list of callable*) – the additional validators for field
- **default** (*instance of value_types*) – the default value of the field

Raises TypeError if passed arguments invalid

Class attributes

`wrapper`

type which should be returned by *bind()* method default realization (may be `None`)

`value_types`

valid types of field value

Attributes

required
Boolean flag passed to constructor

default
The default value for Field, which will be used when no data available to bind

---

Note: You should specify *default* only together with `required=False`, otherwise *default* value will be skipped

---

base_validators
List of attached by processing construction arguments `Validators`

custom_validators
Iterable object contains `custom_validators` passed to constructor

Property

validators
Changed in version 1.0.0.

Generator that returns `base_validators` and `custom_validators` items successively

Methods

validate(*value*)
Validates given value via defined set of `Validators`

bind(*value*)
New in version 0.2.0.

Changed in version 1.0.0.

Validates given value via defined set of `Validators`, wraps it into *wrapper* and returns wrapped value and `None` in second position. If some errors occurred returns an `UNSET` and this errors

If value is mutable obj (for example `list`) it'll be converted to immutable (for example `tuple`)

Parameters **value** – the value to bind

Returns bound value and occurred errors (if there were no errors - `None` will be returned in second position)

Hooks

static field_binding_hook(*method*)
Hook to avoid code duplicating in *Field.bind()* method realizations that provides checking the value to `None` and `UNSET`

You may use it as decorator for *Field.bind()* method in your custom fields

static field_validation_hook(*method*)
Hook to avoid code duplicating in *Field.validate()* method realizations that provides checking the value to `None` and `UNSET`

You may use it as decorator for *Field.validate()* method in your custom fields

Builtin fields summary

This section provides summary for all builtin Fields and API for its base classes. For more information about each Field, see it's API reference.

Note: Since version 1.0.0 inheritance model for builtin Fields has been changed.

Field	Base class	Reflected type	Bound value type
StringField	*SelectField*	Python 2.x: `basestring` Python 3.x: `str`	Python 2.x `unicode` Python 3.x: `str`
IntField		`int`	
BooleanField		`bool`	
FloatField	*IntField*	`float`	`float`
EmailField	*StringField*	same with *StringField*	
Ip4Field	*StringField*	same with *StringField*	
ListField	*SequenceField*	`Iterable`	`tuple`
ArrayField			
DatetimeField	*Field*	same with *StringField*	
DictField		`dict`	`FrozenDict`
EmbeddedFormField		For more information see *Embedded Forms*	

class `flyforms.fields.`**`SelectField`**(*choices=()*, ***kwargs*)

New in version 1.0.0.

This is the base class for all Fields that reflect single values such as `int`, `float`, `bool` and etc. It provides to specify list of valid values (choices). Any passed values which are not in the defined choices will cause validation fails.

Parameters

- **`required`** (*bool*) – boolean flag is this field required or may be `UNSET`
- **`null`** (*bool*) – boolean flag is field value may be `None`
- **`validators`** (*list of callable*) – the additional validators for field
- **`default`** (*instance of value_types*) – the default value of the field
- **`choices`** (*Iterable*) – iterable object contains possible values of this field

class `flyforms.fields.`**`SequenceField`**(*min_length=None*, *max_length=None*, *item_validators=()*, ***kwargs*)

New in version 1.0.0.

This is the base class for all Fields that reflect iterable values such as list, tuple and etc. It provides to specify such as minimum and maximum possible iterable length and validators for each item.

Parameters

- **`required`** (*bool*) – boolean flag is this field required or may be `UNSET`
- **`null`** (*bool*) – boolean flag is field value may be `None`
- **`validators`** (*list of callable*) – the additional validators for field
- **`default`** (*instance of value_types*) – the default value of the field
- **`min_length`** (*int*) – minimum iterable length
- **`max_length`** (*int*) – maximum iterable length
- **`item_validators`** (*list of Callable*) – the additional validators for every item

Builtin fields API

class `flyforms.fields.`**`StringField`**(*min_length=None*, *max_length=None*, *regex=''*, ***kwargs*)

Reflects Python strings

Parameters

- **required** (*bool*) – boolean flag is this field required
- **null** (*bool*) – boolean flag is field value may be `None`
- **min_length** (*int or None*) – the minimum length of the string
- **max_length** (*int or None*) – the maximum length of the string
- **regex** (*str or regexp*) – the regular expression to validate
- **choices** (*iterable*) – iterable object contains possible values of this field
- **validators** (*list of callable*) – the additional validators for field
- **default** (*instance of value_types*) – the default value of the field

class `flyforms.fields.`**EmailField**(***kwargs*)
Reflects Python string corresponding to an email

Parameters

- **required** (*bool*) – boolean flag is this field required
- **null** (*bool*) – boolean flag is field value may be `None`
- **choices** (*iterable*) – iterable object contains possible values of this field
- **validators** (*list of callable*) – the additional validators for field
- **default** (*instance of value_types*) – the default value of the field

class `flyforms.fields.`**Ip4Field**(***kwargs*)
Reflects Python string corresponding to an IPv4 address

Parameters

- **required** (*bool*) – boolean flag is this field required
- **null** (*bool*) – boolean flag is field value may be `None`
- **choices** (*iterable*) – iterable object contains possible values of this field
- **validators** (*list of callable*) – the additional validators for field
- **default** (*instance of value_types*) – the default value of the field

class `flyforms.fields.`**IntField**(*min_value=None*, *max_value=None*, ***kwargs*)
Reflects Python `int` values

Parameters

- **required** (*bool*) – boolean flag is this field required
- **null** (*bool*) – boolean flag is field value may be `None`
- **min_value** – the minimum valid value
- **max_value** – the maximum valid value
- **choices** (*iterable*) – iterable object contains possible values of this field
- **validators** (*list of callable*) – the additional validators for field
- **default** (*instance of value_types*) – the default value of the field

class `flyforms.fields.`**FloatField**(*min_value=None*, *max_value=None*, ***kwargs*)
Reflects Python `float` values

Parameters

- **required** (*bool*) – boolean flag is this field required
- **null** (*bool*) – boolean flag is field value may be `None`
- **min_value** – the minimum valid value
- **max_value** – the maximum valid value
- **choices** (*iterable*) – iterable object contains possible values of this field
- **validators** (*list of callable*) – the additional validators for field
- **default** (*instance of value_types*) – the default value of the field

class `flyforms.fields.`**BooleanField**(***kwargs*)

Reflects Python `bool` values

Parameters

- **required** (*bool*) – boolean flag is this field value may be `UNSET`
- **null** (*bool*) – boolean flag is field value may be `None`
- **validators** (*list of callable*) – the additional validators for field
- **default** (*instance of value_types*) – the default value of the field

class `flyforms.fields.`**ListField**(*jsonify=True*, ***kwargs*)

Reflects iterable Python objects

Parameters

- **required** (*bool*) – boolean flag is this field required or may be `UNSET`
- **null** (*bool*) – boolean flag is field value may be `None`
- **min_length** (*int*) – minimum iterable length
- **max_length** (*int*) – maximum iterable length
- **item_validators** (*list of Callable*) – the additional validators for every item
- **jsonify** (*bool*) – if True all items should be one of `jsonify_types`
- **validators** (*list of callable*) – the additional validators for field
- **default** (*instance of value_types*) – the default value of the field

Usage

```
# coding=utf-8
from flyforms.form import Form
from flyforms.fields import ListField
from flyforms.common import UNSET


class ListForm(Form):
    jsonify_list = ListField(min_length=2, max_length=5)
    common_list = ListField(min_length=3, jsonify=False)

if __name__ == '__main__':
    form = ListForm(
        jsonify_list=["Hello!", 2.5, 0],
        common_list=[object(), 500, "... world!", UNSET]
    )
```

```
    print(form.is_valid)    # >>> True
    print(form.errors)    # >>> {}
```

class flyforms.fields.**ArrayField**(*item_type*, *jsonify=True*, ***kwargs*)

New in version 0.2.0.

Reflects iterable objects where each item same type

Parameters

- **item_type** – type of each item in the list
- **required** (*bool*) – boolean flag is this field required or may be UNSET
- **null** (*bool*) – boolean flag is field value may be None
- **min_length** (*int*) – minimum iterable length
- **max_length** (*int*) – maximum iterable length
- **item_validators** (*list of Callable*) – the additional validators for every item
- **jsonify** (*bool*) – if True all items should be one of jsonify_types
- **validators** (*list of callable*) – the additional validators for field
- **default** (*instance of value_types*) – the default value of the field

Usage

```
# ArrayField usage
from flyforms.form import Form
from flyforms.fields import StringField, ArrayField


class CommentForm(Form):
    login = StringField()
    comment = StringField(max_length=256)
    tags = ArrayField(item_type=str)

if __name__ == '__main__':
    f = CommentForm(
        login="YourLogin",
        comment="Your comment",
        tags=["schema", "python", "json"]  # <-- list
    )

    print(f.is_valid)  # >>> True
    print(f.errors)  # >>> {}
    print(f.to_python())

    # Given list was wrapped into tuple
    print(type(f.tags))  # >>> <type 'tuple'>
```

class flyforms.fields.**DatetimeField**(*required=True*, *fmt='%Y-%m-%d %H:%M:%S'*, *now=False*, *null=False*, *validators=()*)

New in version 0.3.0.

A datetime field.

Parse string contains datetime via datetime.strptime

Parameters

- **required** (*bool*) – boolean flag is this field value may be `UNSET`
- **fmt** (*str*) – datetime format
- **now** (*bool*) – if passed the default value will be datetime.now()
- **null** (*bool*) – boolean flag is field value may be `None`
- **validators** (*list of callable*) – the additional validators for field

class `flyforms.fields.`**DictField**(*schema*, ***kwargs*)

New in version 0.3.0.

Reflects Python dict

Parameters

- **schema** (*dict*) – template to dict validation
- **required** (*bool*) – boolean flag is this field value may be `UNSET`
- **null** (*bool*) – boolean flag is field value may be `None`
- **validators** (*list of callable*) – the additional validators for field
- **default** (*instance of value_types*) – the default value of the field

Usage

```
# DictField usage
from flyforms.form import Form
from flyforms.fields import DictField, EmailField, StringField


class MyForm(Form):
    cred = DictField(
        schema={
            "email": EmailField(),
            "password": StringField(
                min_length=8,
                regex=r"^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#\$%\^&\*])",
                max_length=64
            )
        }
    )


if __name__ == '__main__':
    f = MyForm(
        cred={
            "email": "qwerty@gmail.com",
            "password": "Qwerty_#123"
        }  # <--- dict
    )

    print(f.is_valid)  # >>> True
    print(f.errors)  # >>> {}
    print(f.cred)  # >>> {'password': 'Qwerty_#123', 'email': 'qwerty@gmail.com'}
    print(type(f.cred))  # >>> <class 'flyforms.common.FrozenDict'>  <--- !!!
```

Nested DictField usage

```
# DictField nested usage
from flyforms.form import Form
from flyforms.fields import DictField, ListField, StringField, EmailField
from pprint import pprint


class MyForm(Form):
    field = DictField(
        schema={
            "list_field": ListField(),
            "nested_dict": DictField(
                schema={
                    "field1": EmailField(),
                    "field2": StringField(),
                    "nested_dd": DictField(
                        schema={
                            "email": EmailField(required=False)
                        }
                    )
                }
            )
        }
    )

if __name__ == '__main__':
    f = MyForm(
        field={
            "list_field": [0, 1, "Hello world!"],
            "nested_dict": {
                "field1": "qwerty@qwerty.com",
                "field2": "Hello world!",
                "nested_dd": {
                    "email": "qwerty@qwerty.com"
                }
            }
        }
    )

    print(f.is_valid)  # >>> True
    pprint(f.to_python())
```

Embedded Forms

Since version 1.0.0 FlyForms provides mechanism of encapsulation one Form to an other within *EmbeddedFormField*.

class flyforms.fields.**EmbeddedFormField**(*form_cls*, *required=True*, *null=False*)

Usage

```
# coding=utf-8
from flyforms.form import Form
from flyforms.fields import EmbeddedFormField, EmailField, StringField


class EmbeddedForm(Form):
    email = EmailField()
    password = StringField(
```

```
        min_length=8,
        regex=r"^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#\$%\^&\*])",
        max_length=64
    )


class ContainerForm(Form):
    first_name = StringField(
        regex=r"^[A-Z].*$",
        min_length=3,
        max_length=64
    )

    last_name = StringField(
        regex=r"^[A-Z].*$",
        min_length=3,
        max_length=64
    )

    login_data = EmbeddedFormField(EmbeddedForm, null=True)

if __name__ == '__main__':
    f = ContainerForm(**{
        "first_name": "John",
        "last_name": "Smith",
        "login_data": {
            "email": "smith@gmail.com",
            "password": "Qw3rT!y_"
        }
    })

    print(f.is_bound)  # >>> True
    print(f.is_valid)  # >>> True
    print(f.errors)  # >>> {}
    print(f.to_python())
    print(f.login_data)  # >>> {'password': 'Qw3rT!y_', 'email': 'smith@gmail.com'}
    print(f.login_data.password)  # >>> Qw3rT!y_
```

Custom Fields

Todo

finish

2.2.3 Validators

Generally, Validator is a callable object that validates given value with it's internal logic. If value is not valid *ValidationError* will be raised.

class flyforms.validators.**ValidationError**

Raised when a validator fails to validate it's input.

Validator class

All builtin Validators inherit *Validator* or it's subclass *SimpleValidator*.

class flyforms.validators.**Validator**
The abstract root class for all Validators.

class flyforms.validators.**SimpleValidator**
The Validator's subclass with only one validation case. Given value should satisfies condition in validation_case() method

Builtin validators API

class flyforms.validators.**RequiredValidator**
Validates is given value not UNSET object

class flyforms.validators.**TypeValidator**(*value_types*)
Validates is given value instance of passed value_types

Parameters **value_types** – list of possible value types

class flyforms.validators.**EntryValidator**(*iterable*, *item_type=None*)
Validates is given value in specified during initialization iterable object

Parameters **iterable** – the iterable object

Raise TypeError if given object is not iterable

class flyforms.validators.**MinValueValidator**(*min_value*, *strong=True*)
Validates is given value greater than specified during initialization value

Parameters

- **min_value** – the minimum valid value
- **strong** (*bool*) – boolean flag should be comparison strict or not

class flyforms.validators.**MaxValueValidator**(*max_value*, *strong=True*)
Validates is given value less than specified during initialization value

Parameters

- **max_value** – the maximum valid value
- **strong** (*bool*) – boolean flag should be comparison strict or not

class flyforms.validators.**MinLengthValidator**(*min_length*, *strong=True*)
Validates the minimum object length

Parameters

- **min_length** – the minimum valid length
- **strong** (*bool*) – boolean flag should be comparison strict or not

class flyforms.validators.**MaxLengthValidator**(*max_length*, *strong=True*)
Validates the maximum object length

Parameters

- **max_length** – the maximum valid length
- **strong** (*bool*) – boolean flag should be comparison strict or not

class `flyforms.validators.`**`RegexValidator`**(*regex*, *flags=0*)
Validates string matching with regular expression

Parameters

- **`regex`** – the regular expression
- **`flags`** – flags passed to re.match function

class `flyforms.validators.`**`EmailValidator`**
Validates an email address via simple regex.

class `flyforms.validators.`**`Ip4AddressValidator`**
Validates an IPv4 address via simple regex.

2.3 FlyForms Changelog

2.3.1 Version 1.0.0

Beta 1 version released 29.10.2015

Release notes

This is the first logically completed and stable working version of FlyForms. We held a global refactoring and code optimization and add many new features. Unfortunately, this version is fully incompatible with previous releases. So if you have used the library before, you will have to rewrite the some your code. You can see list of all incompatible changes below. And we hope to get feedback about the new release on BitBucket.

New features

- new class *`FormMetaOptions`* that provides customization of *`Form`* instances behaviour
- new field *`EmbeddedFormField`* that provide to encapsulate one *`Form`* instance into the other
- new fields inheritance model changed (for more information see *Fields*)
- new core exception class: *`UnboundForm`*
- new customization model for unbound form fields (see *`UnboundField`* and *`FormMetaOptions`*)
- Field `validators` is generator now, not a `chain`
- documentation improvement and refactoring

Incompatible changes

- new module `flyforms.core`
- classes *`Field`*, *`Form`*, *`FormMeta`* and descriptor *`FormField`* were moved to `flyforms.core`
- property `SimpleValidator.positive_case` was removed; use `SimpleValidator.validation_case` instead
- property `Form.data` was removed; use *`Form.to_python()`* method instead
- validator `TypedValidator` was renamed to *`TypeValidator`*

- validators `ItemTypedValidator` and `JsonItemTypedValidator` were removed; use *`TypeValidator`*
- methods `Validator.validate()` and `Validator.is_valid()` were removed
- attribute `Form.raw_data` was changed to private *`Form._raw_data`*
- method `Field.is_valid()` was removed
- behavior of method *`Form.to_python()`* has been changed

2.3.2 Version 0.3.0

Released 20.10.2015

- new basic `Fields`: *`DatetimeField`* and *`DictField`*
- property `SimpleValidator.positive_case` was renamed to `validation_case` in *`SimpleValidator`*
- property `SimpleValidator.positive_case` was deprecated and will be removed in v1.0.0
- new method `to_python()` in *`Form`*
- property `Form.data` was deprecated and will be removed in v1.0.0 use `to_python()` method
- other minor improvements

2.3.3 Version 0.2.0

Released 19.10.2015

- issue tracker is available on BitBucket
- new method for *`Field.bind()`* which returns an *immutable bound* value
- new basic `Fields`: *`ListField`* and *`ArrayField`*
- new `Validators`: `ItemTypedValidator` and `JsonItemTypedValidator`
- methods *`Field.validate()`* and `Field.is_valid()` were deprecated and will be removed in v1.0.0
- core descriptor *`FormField`* now uses *`Field.bind()`* instead *`Field.validate()`*
- new module `flyforms.common`
- other minor improvements

2.3.4 Version 0.1.1

Released 14.10.2015.

FlyForms:

- bug with `default` argument for *`Field`* instances fixed
- source tarball added to distribution in addition to wheel

Documentation:

- new section *FlyForms reference* instead just API
- section *Advanced usage* removed

- other minor improvements

2.3.5 Version 0.1.0

Released 12.10.2015.

- Initial release.

2.4 Indices and tables

- genindex
- modindex
- search

f

Symbols

A

B

C

D

E

F

I

L

M

R

S

T

U

V

W