JSON models Documentation

Release 2.2

Szczepan Cieślik

Dec 27, 2017

Contents

Contents:

CHAPTER 1

JSON models

jsonmodels is library to make it easier for you to deal with structures that are converted to, or read from JSON.

- Free software: BSD license
- Documentation: http://jsonmodels.rtfd.org
- Source: https://github.com/beregond/jsonmodels

1.1 Features

- Fully tested with Python 2.7, 3.2, 3.3, 3.4, 3.5, 3.6.
- Support for PyPy (see implementation notes in docs for more details).
- Create Django-like models:

```
from jsonmodels import models, fields, errors, validators


class Cat(models.Base):

    name = fields.StringField(required=True)
    breed = fields.StringField()
    love_humans = fields.IntField(nullable=True)


class Dog(models.Base):

    name = fields.StringField(required=True)
    age = fields.IntField()


class Car(models.Base):
```

```
    registration_number = fields.StringField(required=True)
    engine_capacity = fields.FloatField()
    color = fields.StringField()


class Person(models.Base):

    name = fields.StringField(required=True)
    surname = fields.StringField(required=True)
    nickname = fields.StringField(nullable=True)
    car = fields.EmbeddedField(Car)
    pets = fields.ListField([Cat, Dog], nullable=True)
```

- Access to values through attributes:

```
>>> cat = Cat()
>>> cat.populate(name='Garfield')
>>> cat.name
'Garfield'
>>> cat.breed = 'mongrel'
>>> cat.breed
'mongrel'
```

- Validate models:

```
>>> person = Person(name='Chuck', surname='Norris')
>>> person.validate()
None

>>> dog = Dog()
>>> dog.validate()
*** ValidationError: Field "name" is required!
```

- Cast models to python struct and JSON:

```
>>> cat = Cat(name='Garfield')
>>> dog = Dog(name='Dogmeat', age=9)
>>> car = Car(registration_number='ASDF 777', color='red')
>>> person = Person(name='Johny', surname='Bravo', pets=[cat, dog])
>>> person.car = car
>>> person.to_struct()
{
    'car': {
        'color': 'red',
        'registration_number': 'ASDF 777'
    },
    'surname': 'Bravo',
    'name': 'Johny',
    'nickname': None,
    'pets': [
        {'name': 'Garfield'},
        {'age': 9, 'name': 'Dogmeat'}
    ]
}

>>> import json
>>> person_json = json.dumps(person.to_struct())
```

- You don't like to write JSON Schema? Let *jsonmodels* do it for you:

```
>>> person = Person()
>>> person.to_json_schema()
{
    'additionalProperties': False,
    'required': ['surname', 'name'],
    'type': 'object',
    'properties': {
        'car': {
            'additionalProperties': False,
            'required': ['registration_number'],
            'type': 'object',
            'properties': {
                'color': {'type': 'string'},
                'engine_capacity': {'type': ''},
                'registration_number': {'type': 'string'}
            }
        },
        'surname': {'type': 'string'},
        'name': {'type': 'string'},
        'nickname': {'type': ['string', 'null']}
        'pets': {
            'items': {
                'oneOf': [
                    {
                        'additionalProperties': False,
                        'required': ['name'],
                        'type': 'object',
                        'properties': {
                            'breed': {'type': 'string'},
                            'name': {'type': 'string'}
                        }
                    },
                    {
                        'additionalProperties': False,
                        'required': ['name'],
                        'type': 'object',
                        'properties': {
                            'age': {'type': 'number'},
                            'name': {'type': 'string'}
                        }
                    },
                    {
                        'type': 'null'
                    }
                ]
            },
            'type': 'array'
        }
    }
}
```

- Validate models and use validators, that affect generated schema:

```
>>> class Person(models.Base):
...
...     name = fields.StringField(
...         required=True,
```

```
...             validators=[
...                 validators.Regex('^[A-Za-z]+$'),
...                 validators.Length(3, 25),
...             ],
...         )
...         age = fields.IntField(
...             nullable=True,
...             validators=[
...                 validators.Min(18),
...                 validators.Max(101),
...             ]
...         )
...         nickname = fields.StringField(
...             required=True,
...             nullable=True
...         )
...

>>> person = Person()
>>> person.age = 11
>>> person.validate()
*** ValidationError: '11' is lower than minimum ('18').
>>> person.age = None
>>> person.validate()
None

>>> person.age = 19
>>> person.name = 'Scott_'
>>> person.validate()
*** ValidationError: Value "Scott_" did not match pattern "^[A-Za-z]+$".

>>> person.name = 'Scott'
>>> person.validate()
None

>>> person.nickname = None
>>> person.validate()
*** ValidationError: Field is required!

>>> person.to_json_schema()
{
    "additionalProperties": false,
    "properties": {
        "age": {
            "maximum": 101,
            "minimum": 18,
            "type": ["number", "null"]
        },
        "name": {
            "maxLength": 25,
            "minLength": 3,
            "pattern": "/^[A-Za-z]+$/",
            "type": "string"
        },
        "nickname": {,
            "type": ["string", "null"]
        }
    },
```

```
    "required": [
        "nickname",
        "name"
    ],
    "type": "object"
}
```

For more information, please see topic about validation in documentation.

- Lazy loading, best for circular references:

```
>>> class Primary(models.Base):
...
...     name = fields.StringField()
...     secondary = fields.EmbeddedField('Secondary')

>>> class Secondary(models.Base):
...
...    data = fields.IntField()
...    first = fields.EmbeddedField('Primary')
```

You can use either *Model*, full path *path.to.Model* or relative imports *.Model* or *...Model*.

- Using definitions to generate schema for circular references:

```
>>> class File(models.Base):
...
...     name = fields.StringField()
...     size = fields.FloatField()

>>> class Directory(models.Base):
...
...     name = fields.StringField()
...     children = fields.ListField(['Directory', File])

>>> class Filesystem(models.Base):
...
...     name = fields.StringField()
...     children = fields.ListField([Directory, File])

>>> Filesystem.to_json_schema()
{
    "type": "object",
    "properties": {
        "name": {"type": "string"}
        "children": {
            "items": {
                "oneOf": [
                    "#/definitions/directory",
                    "#/definitions/file"
                ]
            },
            "type": "array"
        }
    },
    "additionalProperties": false,
    "definitions": {
        "directory": {
            "additionalProperties": false,
```

```
            "properties": {
                "children": {
                    "items": {
                        "oneOf": [
                            "#/definitions/directory",
                            "#/definitions/file"
                        ]
                    },
                    "type": "array"
                },
                "name": {"type": "string"}
            },
            "type": "object"
        },
        "file": {
            "additionalProperties": false,
            "properties": {
                "name": {"type": "string"},
                "size": {"type": "number"}
            },
            "type": "object"
        }
    }
}
```

- Compare JSON schemas:

```
>>> from jsonmodels.utils import compare_schemas
>>> schema1 = {'type': 'object'}
>>> schema2 = {'type': 'array'}
>>> compare_schemas(schema1, schema1)
True
>>> compare_schemas(schema1, schema2)
False
```

1.2 More

For more examples and better description see full documentation: http://jsonmodels.rtfd.org.

CHAPTER 2

Installation

At the command line:

```
$ easy_install jsonmodels
```

Or, if you have *virtualenvwrapper* installed:

```
$ mkvirtualenv jsonmodels
$ pip install jsonmodels
```

CHAPTER 3

Usage

To use JSON models in a project:

```
import jsonmodels
```

3.1 Creating models

To create models you need to create class that inherits from *jsonmodels.models.Base* (and *NOT* jsonmodels.models.PreBase to which although refers links in documentation) and have class attributes which values inherits from *jsonmodels.fields.BaseField* (so all other fields classes from *jsonmodels.fields*).

```
class Cat(models.Base):

    name = fields.StringField(required=True)
    breed = fields.StringField()


class Dog(models.Base):

    name = fields.StringField(required=True)
    age = fields.IntField()


class Car(models.Base):

    registration_number = fields.StringField(required=True)
    engine_capacity = fields.FloatField()
    color = fields.StringField()


class Person(models.Base):
```

```
    name = fields.StringField(required=True)
    surname = fields.StringField(required=True)
    car = fields.EmbeddedField(Car)
    pets = fields.ListField([Cat, Dog])
```

3.2 Usage

After that you can use it as normal object. You can pass kwargs in constructor or `jsonmodels.models.PreBase.populate()` method.

```
>>> person = Person(name='Chuck')
>>> person.name
'Chuck'
>>> person.surname
None
>>> person.populate(surname='Norris')
>>> person.surname
'Norris'
>>> person.name
'Chuck'
```

3.3 Validation

You can specify which fields are *required*, if required value is absent during `jsonmodels.models.PreBase.validate()` the `jsonmodels.error.ValidationError` will be raised.

```
>>> bugs = Person(name='Bugs', surname='Bunny')
>>> bugs.validate()

>>> dafty = Person()
>>> dafty.validate()
*** ValidationError: Field is required!
```

Note that required fields are not raising error if no value was assigned during initialization, but first try of accessing will raise it.

```
>>> dafty = Person()
>>> dafty.name
*** ValidationError: Field is required!
```

Also validation is made every time new value is assigned, so trying assign *int* to *StringField* will also raise an error:

```
>>> dafty.name = 3
*** ValidationError: ('Value is wrong, expected type "basestring"', 3)
```

During casting model to JSON or JSONSchema explicite validation is always called.

3.3.1 Validators

Validators can be passed through *validators* keyword, as a single validator, or list of validators (so, as you may be expecting, you can't pass object that extends *List*).

You can try to use validators shipped with this library. To get more details see *jsonmodels.validators*. Shipped validators affect generated schema out of the box, to use full potential JSON schema gives you.

3.3.2 Custom validators

You can always specify your own validators. Custom validator can be object with *validate* method (which takes precedence) or function (or callable object).

Each validator **must** raise exception to indicate validation didn't pass. Returning values like *False* won't have any effect.

```
>>> class RangeValidator(object):
...
...   def __init__(self, min, max):
...     # Some logic here.
...
...   def validate(self, value):
...     # Some logic here.

>>> def some_validator(value):
...   # Some logic here.

>>> class Person(models.Base):
...
...   name = fields.StringField(required=True, validators=some_validator)
...   surname = fields.StringField(required=True)
...   age = fields.IntField(
...     Car, validators=[some_validator, RangeValidator(0, 100)])
```

If your validator have method *modify_schema* you can use it to affect generated schema in any way. Given argument is schema for single field. For example:

```
>>> class Length(object):
...
... def validate(self, value):
...     # Some logic here.
...
... def modify_schema(self, field_schema):
...     if self.minimum_value:
...         field_schema['minLength'] = self.minimum_value
...
...     if self.maximum_value:
...         field_schema['maxLength'] = self.maximum_value
```

3.4 Casting to Python struct (and JSON)

Instance of model can be easy casted to Python struct (and thanks to that, later to JSON). See `jsonmodels.models.PreBase.to_struct()`.

```
>>> cat = Cat(name='Garfield')
>>> dog = Dog(name='Dogmeat', age=9)
>>> car = Car(registration_number='ASDF 777', color='red')
>>> person = Person(name='Johny', surname='Bravo', pets=[cat, dog])
>>> person.car = car
```

```
>>> person.to_struct()
# (...)
```

Having Python struct it is easy to cast it to JSON.

```
>>> import json
>>> person_json = json.dumps(person.to_struct())
```

3.5 Creating JSON schema for your model

JSON schema, although it is far more friendly than XML schema still have something in common with its old friend: people don't like to write it and (probably) they shouldn't do it or even read it. Thanks to *jsonmodels* it is possible to you to operate just on models.

```
>>> person = Person()
>>> schema = person.to_json_schema()
```

And thats it! You can serve then this schema through your API or use it for validation incoming data.

CHAPTER 4

Implementation notes

Below you can read some implementation specific quirks you should know/remember about when you are using *jsonmodels* (especially on production servers/applications).

4.1 PyPy

PyPy is supported, although there is one problem with garbage collecting: **PyPy's weakref implementation is not stable, so garbage collecting may not work, which may cause memory leak** (values for nonexistent objects may still be preserved, since descriptors are for fields implementation).

All others features are fully supported.

CHAPTER 5

jsonmodels

5.1 jsonmodels package

5.1.1 Submodules

5.1.2 jsonmodels.builders module

Builders to generate in memory representation of model and fields tree.

class jsonmodels.builders.**Builder**(*parent=None*, *nullable=False*)
Bases: object

add_definition(*builder*)

count_type(*type*)

get_builder(*type*)

static maybe_build(*value*)

register_type(*type*, *builder*)

class jsonmodels.builders.**EmbeddedBuilder**(**args*, ***kwargs*)
Bases: *jsonmodels.builders.Builder*

add_type_schema(*schema*)

build()

is_definition

class jsonmodels.builders.**ListBuilder**(**args*, ***kwargs*)
Bases: *jsonmodels.builders.Builder*

add_type_schema(*schema*)

build()

is_definition

class jsonmodels.builders.**ObjectBuilder**(*model_type*, **args*, ***kwargs*)
Bases: *jsonmodels.builders.Builder*

add_field(*name*, *field*, *schema*)

build()

build_definition(*add_defintitions=True*, *nullable=False*)

is_definition

is_root

type_name

class jsonmodels.builders.**PrimitiveBuilder**(**args*, ***kwargs*)
Bases: *jsonmodels.builders.Builder*

build()

set_type(*type*)

5.1.3 jsonmodels.collections module

class jsonmodels.collections.**ModelCollection**(*field*)
Bases: list

ModelCollection is list which validates stored values.

Validation is made with use of field passed to *__init__* at each point, when new value is assigned.

append(*value*)

5.1.4 jsonmodels.errors module

exception jsonmodels.errors.**FieldNotFound**
Bases: exceptions.RuntimeError

exception jsonmodels.errors.**FieldNotSupported**
Bases: exceptions.ValueError

exception jsonmodels.errors.**ValidationError**
Bases: exceptions.RuntimeError

5.1.5 jsonmodels.fields module

class jsonmodels.fields.**BaseField**(*required=False*, *nullable=False*, *help_text=None*, *validators=None*, *default=None*)
Bases: object

Base class for all fields.

get_default_value()
Get default value for field.

Each field can specify its default.

parse_value(*value*)
Parse value from primitive to desired format.

Each field can parse value to form it wants it to be (like string or int).

to_struct(*value*)
Cast value to Python structure.

types = None

validate(*value*)

validate_for_object(*obj*)

class jsonmodels.fields.**BoolField**(*required=False*, *nullable=False*, *help_text=None*, *validators=None*, *default=None*)
Bases: *jsonmodels.fields.BaseField*

Bool field.

parse_value(*value*)
Cast value to *bool*.

types = (<type 'bool'>,)

class jsonmodels.fields.**DateField**(*str_format=None*, **args*, ***kwargs*)
Bases: *jsonmodels.fields.StringField*

Date field.

default_format = '%Y-%m-%d'

parse_value(*value*)
Parse string into instance of *date*.

to_struct(*value*)
Cast *date* object to string.

types = (<type 'datetime.date'>,)

class jsonmodels.fields.**DateTimeField**(*str_format=None*, **args*, ***kwargs*)
Bases: *jsonmodels.fields.StringField*

Datetime field.

parse_value(*value*)
Parse string into instance of *datetime*.

to_struct(*value*)
Cast *datetime* object to string.

types = (<type 'datetime.datetime'>,)

class jsonmodels.fields.**EmbeddedField**(*model_types*, **args*, ***kwargs*)
Bases: *jsonmodels.fields.BaseField*

Field for embedded models.

parse_value(*value*)
Parse value to proper model type.

to_struct(*value*)

validate(*value*)

class jsonmodels.fields.**FloatField**(*required=False*, *nullable=False*, *help_text=None*, *validators=None*, *default=None*)
Bases: *jsonmodels.fields.BaseField*

Float field.

types = (<type 'float'>, <type 'int'>)

class jsonmodels.fields.**IntField**(*required=False*, *nullable=False*, *help_text=None*, *validators=None*, *default=None*)

Bases: *jsonmodels.fields.BaseField*

Integer field.

parse_value(*value*)

Cast value to *int*, e.g. from string or long

types = (<type 'int'>,)

class jsonmodels.fields.**ListField**(*items_types=None*, **args*, ***kwargs*)

Bases: *jsonmodels.fields.BaseField*

List field.

parse_value(*values*)

Cast value to proper collection.

to_struct(*values*)

types = (<type 'list'>,)

validate(*value*)

validate_single_value(*item*)

class jsonmodels.fields.**StringField**(*required=False*, *nullable=False*, *help_text=None*, *validators=None*, *default=None*)

Bases: *jsonmodels.fields.BaseField*

String field.

types = (<type 'basestring'>,)

class jsonmodels.fields.**TimeField**(*str_format=None*, **args*, ***kwargs*)

Bases: *jsonmodels.fields.StringField*

Time field.

parse_value(*value*)

Parse string into instance of *time*.

to_struct(*value*)

Cast *time* object to string.

types = (<type 'datetime.time'>,)

5.1.6 jsonmodels.models module

class jsonmodels.models.**Base**(***kwargs*)

Bases: object

Base class for all models.

get_field(*field_name*)

Get field associated with given attribute.

classmethod iterate_over_fields()

Iterate through fields and values.

populate(***kw*)

Populate values to fields. Skip non-existing.

classmethod to_json_schema()
Generate JSON schema for model.

to_struct()
Cast model to Python structure.

validate()
Explicitly validate all the fields.

5.1.7 jsonmodels.parsers module

Parsers to change model structure into different ones.

jsonmodels.parsers.**build_json_schema**(*value*, *parent_builder=None*)

jsonmodels.parsers.**build_json_schema_object**(*cls*, *parent_builder=None*)

jsonmodels.parsers.**build_json_schema_primitive**(*cls*, *parent_builder*)

jsonmodels.parsers.**to_json_schema**(*cls*)
Generate JSON schema for given class.

Parameters **cls** – Class to be casted.

Return type dict

jsonmodels.parsers.**to_struct**(*model*)
Cast instance of model to python structure.

Parameters **model** – Model to be casted.

Return type dict

5.1.8 jsonmodels.utilities module

class jsonmodels.utilities.**PythonRegex**(*regex*, *flags*)
Bases: tuple

flags
Alias for field number 1

regex
Alias for field number 0

jsonmodels.utilities.**compare_schemas**(*one*, *two*)
Compare two structures that represents JSON schemas.

For comparison you can't use normal comparison, because in JSON schema lists DO NOT keep order (and Python lists do), so this must be taken into account during comparison.

Note this wont check all configurations, only first one that seems to match, which can lead to wrong results.

Parameters

- **one** – First schema to compare.
- **two** – Second schema to compare.

Return type *bool*

jsonmodels.utilities.**convert_ecma_regex_to_python**(*value*)
Convert ECMA 262 regex to Python tuple with regex and flags.

If given value is already Python regex it will be returned unchanged.

Parameters **value** (*string*) – ECMA regex.

Returns 2-tuple with *regex* and *flags*

Return type namedtuple

jsonmodels.utilities.**convert_python_regex_to_ecma**(*value*, *flags=[]*)
Convert Python regex to ECMA 262 regex.

If given value is already ECMA regex it will be returned unchanged.

Parameters

- **value** (*string*) – Python regex.
- **flags** (*list*) – List of flags (allowed flags: *re.I*, *re.M*)

Returns ECMA 262 regex

Return type str

jsonmodels.utilities.**is_ecma_regex**(*regex*)
Check if given regex is of type ECMA 262 or not.

Return type bool

5.1.9 jsonmodels.validators module

Predefined validators.

class jsonmodels.validators.**Enum**(**choices*)
Bases: object

Validator for enums.

modify_schema(*field_schema*)

validate(*value*)

class jsonmodels.validators.**Length**(*minimum_value=None*, *maximum_value=None*)
Bases: object

Validator for length.

modify_schema(*field_schema*)
Modify field schema.

validate(*value*)
Validate value.

class jsonmodels.validators.**Max**(*maximum_value*, *exclusive=False*)
Bases: object

Validator for maximum value.

modify_schema(*field_schema*)
Modify field schema.

validate(*value*)
Validate value.

class jsonmodels.validators.**Min**(*minimum_value*, *exclusive=False*)
Bases: object

Validator for minimum value.

modify_schema(*field_schema*)
Modify field schema.

validate(*value*)
Validate value.

class jsonmodels.validators.**Regex**(*pattern*, ***flags*)
Bases: object

Validator for regular expressions.

FLAGS = {'ignorecase': 2, 'multiline': 8}

modify_schema(*field_schema*)
Modify field schema.

validate(*value*)
Validate value.

5.1.10 Module contents

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/beregond/jsonmodels/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

6.1.4 Write Documentation

JSON models could always use more documentation, whether as part of the official JSON models docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/beregond/jsonmodels/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *jsonmodels* for local development.

1. Fork the *jsonmodels* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/jsonmodels.git
```

3. Install your local copy into a virtualenv. Assuming you have *virtualenvwrapper* installed, this is how you set up your fork for local development:

```
$ mkvirtualenv jsonmodels
$ cd jsonmodels/
$ pip install -r requirements.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass the tests, including testing other Python versions with tox:

```
$ pytest
$ tox
```

   To get tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.3, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/beregond/jsonmodels/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ pytest -k test_jsonmodels
```

CHAPTER 7

Credits

7.1 Development Lead

- Szczepan Cieślik <szczepan.cieslik@gmail.com>

7.2 Contributors

(In alphabetical order)

- Chris Targett <chris.targett@xlevus.net>
- Dima Kuznetsov <dima.kuznetsov@toganetworks.com>
- Johannes Garimort <johannes.garimort@gmx.net>
- Omer Anson <omer.anson@toganetworks.com>
- Pavel Lipchak <kazemat92@gmail.com>
- Vorren

CHAPTER 8

History

8.1 2.2 (2017-08-21)

- Fixed time fields, when value is not required.
- Dropped support for python 2.6
- Added support for python 3.6
- Added nullable param for fields.
- Improved model representation.

8.2 2.1.5 (2017-02-01)

- Fixed DateTimefield error when value is None.
- Fixed comparing models without required values.

8.3 2.1.4 (2017-01-24)

- Allow to compare models based on their type and fields (rather than their reference).

8.4 2.1.3 (2017-01-16)

- Fixed generated schema.
- Improved JSON serialization.

8.5 2.1.2 (2016-01-06)

- Fixed memory leak.

8.6 2.1.1 (2015-11-15)

- Added support for Python 2.6, 3.2 and 3.5.

8.7 2.1 (2015-11-02)

- Added lazy loading of types.
- Added schema generation for circular models.
- Improved readability of validation error.
- Fixed structure generation for list field.

8.8 2.0.1 (2014-11-15)

- Fixed schema generation for primitives.

8.9 2.0 (2014-11-14)

- Fields now are descriptors.
- Empty required fields are still validated only during explicite validations.

8.9.1 Backward compatibility breaks

- Renamed _types to types in fields.
- Renamed _items_types to items_types in ListField.
- Removed data transformers.
- Renamed module *error* to *errors*.
- Removed explicit validation - validation occurs at assign time.
- Renamed *get_value_replacement* to *get_default_value*.
- Renamed modules *utils* to *utilities*.

8.10 1.4 (2014-07-22)

- Allowed validators to modify generated schema.
- Added validator for maximum value.

- Added utilities to convert regular expressions between Python and ECMA formats.
- Added validator for regex.
- Added validator for minimum value.
- By default "validators" property of field is an empty list.

8.11 1.3.1 (2014-07-13)

- Fixed generation of schema for BoolField.

8.12 1.3 (2014-07-13)

- Added new fields (BoolField, TimeField, DateField and DateTimeField).
- ListField is always not required.
- Schema can be now generated from class itself (not from an instance).

8.13 1.2 (2014-06-18)

- Fixed values population, when value is not dictionary.
- Added custom validators.
- Added tool for schema comparison.

8.14 1.1.1 (2014-06-07)

- Added possibility to populate already initialized data to EmbeddedField.
- Added *compare_schemas* utility.

8.15 1.1 (2014-05-19)

- Added docs.
- Added json schema generation.
- Added tests for PEP8 and complexity.
- Moved to Python 3.4.
- Added PEP257 compatibility.
- Added help text to fields.

8.16 1.0.5 (2014-04-14)

- Added data transformers.

8.17 1.0.4 (2014-04-13)

- List field now supports simple types.

8.18 1.0.3 (2014-04-10)

- Fixed compatibility with Python 3.
- Fixed *str* and *repr* methods.

8.19 1.0.2 (2014-04-03)

- Added deep data initialization.

8.20 1.0.1 (2014-04-03)

- Added *populate* method.

8.21 1.0 (2014-04-02)

- First stable release on PyPI.

8.22 0.1.0 (2014-03-17)

- First release on PyPI.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

L

M

O

P

R

S

T

V