# pyrs-schema Documentation

***Release 0.8.0***

**Csaba Palankai**

October 17, 2015

Contents

CHAPTER 1

# Contents

## 1.1 MicroService framework :: Schema

Project homepage: https://github.com/palankai/pyrs-schema
Documentation: http://pyrs-schema.readthedocs.org/
Issues: https://github.com/palankai/pyrs-schema/issues

### 1.1.1 What is this package for

I've used different python frameworks for data serialisation many times. Mostly when I had to implement an API for my work. I felt many times those frameworks did good job but not extensible enough. Also writing easily an API which is satisfy every expectations of projects, without coupled restrictions sometimes really hard.

### 1.1.2 Nutshell

```
from pyrs import schema

class UserSchema(schema.Object):
    version = schema.Version(version='1.0')
    username = schema.StringField(required=True)
    password = schema.StringField(required=True, tags=['writeonly'])
    email = schema.EmailField(title='Registered email address')

writer = schema.JSONWriter(UserSchema)
jsonstring = writer.write(data) # The validation also happen

schemawriter = schema.JSONSchemaWriter()
jsonschemastr = writer.write(UserSchema)
```

### 1.1.3 Features

- Easy schema definition
- Schema validation
- Decoupled serialisation, validation
- Extensible API

### 1.1.4 Installation

The code is tested with python 2.7, 3.3, 3.4.

```
$ pip install pyrs-schema
```

### 1.1.5 Dependencies

See requirements.txt. But The goal is less dependency as possible. The main dependency is the python jsonchema The validation is using that package.

Notice that even it's a JSON schema validator this work still can be used for any (compatible) schema validation.

### 1.1.6 Important caveats

This code is in beta version. I working hard on write stable as possible API in the first place but while this code in 0.x version you should expect some major modification on the API.

### 1.1.7 The ecosystem

This work is part of pyrs framework. The complete framework follow the same intention to implement flexible solution.

### 1.1.8 Contribution

I really welcome any comments! I would be happy if you fork my code or create pull requests. I've already really strong opinions what I want to achieve and how, though any help would be welcomed.

Feel free drop a message to me!

## 1.2 Base module

**class** pyrs.schema.base.**Base**(***attrs*)
Bases: *pyrs.schema.base.Schema*

**get_jsonschema**()

**class** pyrs.schema.base.**DeclarativeMetaclass**
Bases: type

**classmethod get_inherited**(*mcls*, *cls*, *name*, *base*, *remove_if_none=False*)

**classmethod update_attrs**(*mcls*, *attrs*, *name*, *clsname*)

**classmethod update_fields**(*mcls*, *attrs*, *name*, *base*)

**class** pyrs.schema.base.**Schema**(*_jsonschema=None*, ***attrs*)
Bases: object

**dialect**

**exclude_tags**

**exclusive**

**fieldname**

**get_attr**(*name*, *default=None*, *expected=None*)

**get_jsonschema**()

**getter**(*func*)

**has_attr**(*name*, *expected=None*)

**is_excluded**

**is_exclusive**

**logger**

**classmethod mixin**(*src*)

**parent**

**root**

**setter**(*func*)

**to_python**(*value*)
Convert the value to a real python object

**to_raw**(*value*)
Convert the value to a dict of primitives

**class** pyrs.schema.base.**SchemaDict**(*origin*, **args*, ***kwargs*)
Bases: dict

**class** pyrs.schema.base.**Set**(*items=None*)
Bases: set

pyrs.schema.base.**constraint**(*code*, *hint=None*)

## 1.3 Basic types

This module introduce the basic schema types.

**class** pyrs.schema.types.**AllOf**(**_types*, ***attrs*)
Bases: *pyrs.schema.types.MultiSchema*

**to_python**(*value*)

**to_raw**(*value*)

**class** pyrs.schema.types.**Any**(***attrs*)
Bases: *pyrs.schema.base.Base*

**class** pyrs.schema.types.**AnyOf**(**_types*, ***attrs*)
Bases: *pyrs.schema.types.MultiSchema*

**to_python**(*value*)

**to_raw**(*value*)

**class** pyrs.schema.types.**Array**(***attrs*)
Bases: *pyrs.schema.base.Base*

Successful validation of an array instance with regards to these two keywords is determined as follows:

if "items" is not present, or its value is an object, validation of the instance always succeeds, regardless of the value of "additional"

if the value of "additional" is boolean value true or an object, validation of the instance always succeeds;

if the value of "additional" is boolean value false and the value of "items" is an array, the instance is valid if its size is less than, or equal to, the size of "items".

**Array specific options:**

**min_items:** An array instance is valid against "min_items" if its size is greater than, or equal to, the value of this keyword.

**max_items:** An array instance is valid against "max_items" if its size is less than, or equal to, the value of this keyword.

**unique_items:** If this keyword has boolean value false, the instance validates successfully. If it has boolean value true, the instance validates successfully if all of its elements are unique.

**get_jsonschema**()

**class** pyrs.schema.types.**Boolean**(**attrs*)
Bases: *pyrs.schema.base.Base*

**class** pyrs.schema.types.**Date**(**attrs*)
Bases: *pyrs.schema.types.String*

**to_python**(*value*)

**to_raw**(*value*)

**class** pyrs.schema.types.**DateTime**(**attrs*)
Bases: *pyrs.schema.types.String*

**to_python**(*value*)

**to_raw**(*value*)

**class** pyrs.schema.types.**Duration**(**attrs*)
Bases: *pyrs.schema.types.String*

**to_python**(*value*)

**to_raw**(*value*)

**class** pyrs.schema.types.**Email**(**attrs*)
Bases: *pyrs.schema.types.String*

**class** pyrs.schema.types.**Enum**(**attrs*)
Bases: *pyrs.schema.types.Any*

JSON generic enum class

**Parameters** **enum** (*list*) – list of possible values

**get_jsonschema**()
Ensure the generic schema, remove *types*

**Returns** Gives back the schema

**Return type** dict

**class** pyrs.schema.types.**Integer**(**attrs*)
Bases: *pyrs.schema.types.Number*

**Integer specific agruments:**

**maximum, exclusive_max:** The value of *maximum* MUST be a number. The value of *exclusive_max* MUST be a boolean. If "exclusiveMaximum" is present, "maximum" MUST also be present. Successful validation depends on the presence and value of *exclusive_max*. If it iss is not present, or has boolean value false, then the instance is valid if it is lower than, or equal to, the value of *maximum*. If *exclusive_max* has boolean value true, the instance is valid if it is strictly lower than the value of *maximum*

**minimum, exclusive_min:** The value of *minimum* MUST be a number. The value of *exclusive_min* MUST be a boolean. If "exclusiveMinimum" is present, "minimum" MUST also be present. Successful validation depends on the presence and value of *exclusive_min*. If it iss is not present, or has boolean value false, then the instance is valid if it is greater than, or equal to, the value of *minimum*. If *exclusive_min* is present and has boolean value true, the instance is valid if it is strictly greater than the value of *minimum*.

**multiple:** The value MUST be an number. This number MUST be strictly greater than 0. A numeric instance is valid against *multiple* if the result of the division of the instance by this keyword's value is an integer.

**class** pyrs.schema.types.**MultiSchema**(**_types*, ***attrs*)
Bases: *pyrs.schema.base.Schema*

**get_jsonschema**()

**class** pyrs.schema.types.**Not**(**_types*, ***attrs*)
Bases: *pyrs.schema.types.MultiSchema*

**class** pyrs.schema.types.**Null**(***attrs*)
Bases: *pyrs.schema.types.Any*

Generic null type

**get_jsonschema**()

**class** pyrs.schema.types.**Number**(***attrs*)
Bases: *pyrs.schema.base.Base*

**Number specific agruments:**

**maximum, exclusive_max:** The value of *maximum* MUST be a number. The value of *exclusive_max* MUST be a boolean. If *exclusive_max* is present, *maximum* MUST also be present. Successful validation depends on the presence and value of *exclusive_max*. If it iss is not present, or has boolean value false, then the instance is valid if it is lower than, or equal to, the value of *maximum*. If *exclusive_max* has boolean value true, the instance is valid if it is strictly lower than the value of *maximum*

**minimum, exclusive_min:** The value of *minimum* MUST be a number. The value of *exclusive_min* MUST be a boolean. If *exclusive_min* is present, *minimum* MUST also be present. Successful validation depends on the presence and value of *exclusive_min*. If it iss is not present, or has boolean value false, then the instance is valid if it is greater than, or equal to, the value of *minimum*. If *exclusive_min* is present and has boolean value true, the instance is valid if it is strictly greater than the value of *minimum*.

**multiple:** The value MUST be an number. This number MUST be strictly greater than 0. A numeric instance is valid against *multiple* if the result of the division of the instance by this keyword's value is an integer.

**get_jsonschema**()

**class** pyrs.schema.types.**Object**(*extend=None*, ***attrs*)
Bases: *pyrs.schema.base.Base*

Declarative schema object

**Object specific attributes:**

**additional:** boolean value: enable or disable extra items on the object schema: items which are valid against the schema allowed to extend **false by default**

**min_properties:** An object instance is valid against *min_properties* if its number of properties is greater than, or equal to, the value.

**max_properties:** An object instance is valid against *max_properties* if its number of properties is less than, or equal to, the value.

**pattern:** Should be a dict where the keys are valid regular excpressions and the values are schema instances. The object instance is valid if the extra properties (which are not listed as property) valid against the schema while name is match on the pattern.

Be careful, the pattern sould be explicit as possible, if the pattern match on any normal property the validation should be successful against them as well.

A normal object should looks like the following:

```
class Translation(types.Object):
    keyword = types.String()
    value = types.String()

    class Attrs:
        additional = False
        patterns = {
            'value_[a-z]{2}': types.String()
        }
```

**extend**(*properties*)
Extending the exist same with new properties. If you want to extending with an other schema, you should use the other schame *properties*

**fields**

**get_jsonschema**()

**to_python**(*value*)
Convert the value to a real python object

**to_raw**(*value*)
Convert the value to a JSON compatible value

**class** pyrs.schema.types.**OneOf**(**types*, ***attrs*)
Bases: *pyrs.schema.types.AnyOf*

**class** pyrs.schema.types.**Password**(***attrs*)
Bases: *pyrs.schema.types.String*

**get_jsonschema**()

**to_raw**(*value*)

**class** pyrs.schema.types.**Ref**(***attrs*)
Bases: *pyrs.schema.types.Any*

**get_jsonschema**()

**class** pyrs.schema.types.**String**(***attrs*)
Bases: *pyrs.schema.base.Base*

**String specific arguments:**

**pattern:** The value of this keyword MUST be a string. This string SHOULD be a valid regular expression, according to the ECMA 262 regular expression dialect. A string instance is considered valid if the regular expression matches the instance successfully. Recall: regular expressions are not implicitly anchored.

**minlen (int >=0):** The value of this keyword MUST be an integer. This integer MUST be greater than, or equal to, 0. A string instance is valid against this keyword if its length is greater than, or equal to, the value of this keyword.

**maxlen (int >=minlen):** The value of this keyword MUST be an integer. This integer MUST be greater than, or equal to, 0. A string instance is valid against this keyword if its length is less than, or equal to, the value of this keyword.

**blank (bool):** The value of *blank* MUST be a boolean. Successful validation depends on presence and value of *min_len*. If *min_len* is present and its value is greather than 0 this keyword has no effect. If *min_len* is not present or its value is 0 the value of *min_len* will be set to 1.

**get_jsonschema**()

**class** pyrs.schema.types.**Time**(**attrs*)
Bases: *pyrs.schema.types.String*

**to_python**(*value*)

**to_raw**(*value*)

**class** pyrs.schema.types.**TimeDelta**(**attrs*)
Bases: *pyrs.schema.types.Number*

**to_python**(*value*)

**to_raw**(*value*)

**class** pyrs.schema.types.**Version**(**attrs*)
Bases: *pyrs.schema.types.String*

**expected_version**

**getvalue**(*value*)

**split_version**(*version*)

**validate_version**(*version*)

## 1.4 Schema IO

This module introduce the base classes for reading and writing data based on schema. The preferred way is using reader is writer rather than using the schema itself. It gives more flexibility and more extensibility.

**class** pyrs.schema.schemaio.**JSONFormReader**(*schema*)
Bases: *pyrs.schema.schemaio.JSONReader*

**read**(*data*)

**class** pyrs.schema.schemaio.**JSONReader**(*schema*)
Bases: *pyrs.schema.schemaio.Reader*

**read**(*data*)

**class** pyrs.schema.schemaio.**JSONSchemaDictValidator**(*schema*)
Bases: *pyrs.schema.schemaio.JSONSchemaValidator*

**validate**(*data*)

**class** pyrs.schema.schemaio.**JSONSchemaValidator**(*schema*)
Bases: *pyrs.schema.schemaio.Validator*

**validate**(*data*)

**class** pyrs.schema.schemaio.**JSONSchemaWriter**
Bases: *pyrs.schema.schemaio.SchemaWriter*

**extract**(*schema*)

**write**(*schema*)

**class** pyrs.schema.schemaio.**JSONWriter**(*schema*)
Bases: *pyrs.schema.schemaio.Writer*

**write**(*data*)

**class** pyrs.schema.schemaio.**Reader**(*schema*)
Bases: *pyrs.schema.schemaio.SchemaIO*

Reader abstract class At least the *read* method should be implemented

```
sw = Reader(CustomSchema())
data = sw.write(<custom datastructure>)
```

**read**(*data*)
with *self.schema* select the proper schema and read the data, validate the input and gives back the decoded value

**class** pyrs.schema.schemaio.**SchemaIO**(*schema*)
Bases: object

The schema IO gives chance to Schema remain independent from the serialisation method. Even the schema provide conversion still just based on primitive values.

**class** pyrs.schema.schemaio.**SchemaWriter**
Bases: object

Abstract implementation of schema writer. The main purpose of this class to ensure different useage of the schema. Add extra value if it's necessary which can't be implemented by the schema itself.

**write**(*schema*)

**class** pyrs.schema.schemaio.**Validator**(*schema*)
Bases: *pyrs.schema.schemaio.SchemaIO*

Abstract base class of validators.

**validate**(*data*)

**class** pyrs.schema.schemaio.**Writer**(*schema*)
Bases: *pyrs.schema.schemaio.SchemaIO*

Writer abstract class At least the *write* method should be implemented

```
sw = Writer(CustomSchema())
encoded_data = sw.write({'custom': 'value'})
```

**write**(*data*)
With *self.schema* select the proper schema, encode the given data then gives it back.

pyrs.schema.schemaio.**select_json_validator**(*schema*)

## 1.5 Formats

pyrs.schema.formats.**date_format_checker**(*instance*)

pyrs.schema.formats.**datetime_format_checker**(*instance*)

pyrs.schema.formats.**duration_format_checker**(*instance*)

pyrs.schema.formats.**format_checker**(*name*, *raises=()*)

pyrs.schema.formats.**parse_datetime**(*datetimestring*)
Parses ISO 8601 date-times into datetime.datetime objects. This function uses parse_date and parse_time to do the job, so it allows more combinations of date and time representations, than the actual ISO 8601:2004 standard allows.

pyrs.schema.formats.**time_format_checker**(*instance*)

## 1.6 Exceptions

**class ValidationError**
Simple import from jsonschema errors

**exception** pyrs.schema.exceptions.**ConstraintError**(*message=None*, *against=None*)
Bases: Exception

**exception** pyrs.schema.exceptions.**FormatError**(*message='Unrecognised input format'*, *value=None*)
Bases: *pyrs.schema.exceptions.SchemaError*

Cover serialization and deserialization errors and related parse errors. It would be raised when the object cannot be converted.

**exception** pyrs.schema.exceptions.**ParseError**(*message*, *value*, *error='ParseError'*)
Bases: *pyrs.schema.exceptions.SchemaError*

Cover serialization and deserialization errors and related parse errors. It would be raised when the object cannot be converted.

**exception** pyrs.schema.exceptions.**SchemaError**(*message*, *value*, *error=None*)
Bases: Exception

Core exception, you can use it to catch all kind of errors. Unlikely to raised directly. Could contain multiple errors.

**exception** pyrs.schema.exceptions.**ValidationError**(*message*, *value*, *invalid*, *against*, *path=None*)
Bases: *pyrs.schema.exceptions.ValidationErrors*

Cover a single validation error

**exception** pyrs.schema.exceptions.**ValidationErrors**(*message*, *value*, *errors=None*)
Bases: *pyrs.schema.exceptions.SchemaError*

Cover the validation errors.

## 1.7 Changelog

The [!] sign marks the incompatible changes.

### 1.7.1 0.8

This release aim to improve the usability and flexibility. The declared schema easily perambulate through with `parent` and `root`. This are set on any child schema **after** the schema initialised. This improvement helps to build complex schemas, complex dependency without context. You can also introduce special behaviour like `dialect` which is any time gets its value from the top level schema.

#### Features

- Add `logger` to Schema
- `Any` type introduced
- Got rid of context argument
- The get_jsonschema gives back a Wrapper which contains the schema object itself as `origin` attribute
- Default value handling
- Custom constraints introduced
- Mixin introduced to extend the functionality in any 3rd party module
- Make available the `parent` and `root` in any Schema

#### Fixes

- The excluded fields will be no shown up in the result

#### Minor changes

- `include` has been renamed to `exclusive` [!]
- `dialect` introduced as Schema field
- Using the class rather than instance of Schema is not supported [!]
- Password field introduced
- Email field introduced
- Version field introduced (future changes possible!)

### 1.7.2 0.7.3

#### Fixes

- Fix JSONReader / Form reader regarding dict input, dict schema

### 1.7.3 0.7.2

#### Fixes

- Get rid of the validation on FormReader

### 1.7.4 0.7.1

#### Minor modifications

- `exceptions.*` also imported in the `pyrs.schema`

### 1.7.5 0.7

This release a major release with lots of modification on API. One of the main concept is decouple responsibilities of schema, means serialisation, deserialisation, validation. Regarding this I removed the `load`, `dump`, `validate` functions from schema. Also make possible the further improvement the `get_schema` renamed to `get_jsonschema`.

#### Major improvements

- `JSONSchemaValidator` introduced
- `JSONReader` introduced
- `JSONFormReader` introduced
- `JSONWriter` introduced
- `validate` removed from `Schema` (and the whole validation) [!]
- `load` removed from `Schema` [!]
- `dump` removed from `Schema` [!]
- `to_dict` renamed to `to_raw` [!]
- `get_schema` renamed to `get_jsonschema` [!]
- Changed to MIT license

#### Minor modifications

- `ValidationErrors` introduced and has became the unified umbrella error
- `SchemaWriter` and `JSONSchemaWriter` introduced
- `JSONSchemaDictValidator` introduced (for dict based schemas)
- Cover document changed regarding the new API
- Number types functionality extended
- String type functionality extended
- More tests for date and time related types
- Doc and test removed from build
- Changelog introduced

## 1.8 License

CHAPTER 2

# Indices and tables

- genindex
- modindex
- search

## p

## A



## B



## C



## D



## E



## F



## G



## H



## I



## J