Strainer Documentation

Release 0.0.4

Alex Kessinger

Jul 22, 2017

Contents

Strainer is a different take on object serialization and validation in python.

It utilizes a functional style over classes.

A Strainer Example

```
import datetime
from strainer import (serializer, field, child,
                      formatters, validators,
                      ValidationException)

artist_serializer = serializer(
  field('name', validators=[validators.required()])
)

album_schema = serializer(
  field('title', validators=[validators.required()]),
  field('release_date',
        validators=[validators.required(), validators.datetime()],
        formatters=[formatters.format_datetime()]),
  child('artist', serializer=artist_serializer, validators=[validators.required()])
)

class Artist(object):
    def __init__(self, name):
        self.name = name

class Album(object):
    def __init__(self, title, release_date, artist):
        self.title = title
        self.release_date = release_date
        self.artist = artist

bowie = Artist(name='David Bowie')
album = Album(
    artist=bowie,
    title='Hunky Dory',
    release_date=datetime.datetime(1971, 12, 17)
)
```

Given that we can now serialize, deserialize, and validate data

```
>>> album_schema.serialize(album)
{'artist': {'name': 'David Bowie'},
 'release_date': '1971-12-17T00:00:00',
 'title': 'Hunky Dory'}
>>> album_schema.deserialize(album_schema.serialize(album))
{'artist': {'name': 'David Bowie'},
 'release_date': datetime.datetime(1971, 12, 17, 0, 0, tzinfo=<iso8601.Utc>),
 'title': 'Hunky Dory'}
>>> input = album_schema.serialize(album)
>>> del input['artist']
>>> album_schema.deserialize(input)
ValidationException: {'artist': ['This field is required']}
```

the example has been borrowed from Marshmallow

CHAPTER 1

Introduction to Strainer

Strainer was built with restful api's in mind. Here is an informal overview of how to use strainer in that domain.

The goal of this document is to give you enough technical specifics to understand how Strainer works, but this isn't intended to be a tutorial or reference. Once you have your bearings dive into the more technical parts of the documentation.

Background

Strainer was built to serialize rich Python objects into simple data structures. You might use Strainer with an object relation mapper like Django's ORM, or SQLAlchemy. So, first we are going to define some models that we will use for the rest of the introduction.

We are going to cover some aspects of creating an API that will track RSS feeds and their items. Here are two simple models that could represent RSS feeds and their items.

```
class Feed(object):
    def __init__(self, feed, name, items):
        self.feed = feed
        self.name = name
        self.items = items

class FeedItem(object):
    def __init__(self, title, pub_date):
        self.title = title
        self.pub_date = pub_date
```

We have the models, but now we want to create a JSON API for our models. We will need to serialize our models, which are rich python objects, into simple dicts so that we may convert them into JSON. First step is to create the serializer.

Create A Feed Serializer

To start, we will create serializers for each model. The job of a serializer is to take a rich python object and boil it down to a simple python dict that can be easily converted into JSON. Given the Feed model we just created, a serializer might look like this.

```
from strainer import serializer, field, formatters, validators

feed_serializer = serializer(
  field('feed', validators=[validators.required()]),
  field('name', validators=[validators.required()]),
)
```

This serializer will map the feed, and name attributes into a simple python dict. Now, we can nest the item serializer into the feed serializer, here's how.

```
from strainer import serializer, field, many, formatters, validators

feed_item_serializer = serializer(
  field('title', validators=[validators.required()]),
  field('pub_date', validators=[validators.required(), validators.datetime()],
        formatters=[formatters.format_datetime()]),
)

feed_serializer = serializer(
  field('feed', validators=[validators.required()]),
  field('name', validators=[validators.required()]),
  many('items', serializer=feed_item_serializer),
)
```

Using A Feed Serializer

We can now use the serializer. We first can instantiate some models, and then we will serialize them into dicts.

```
>>> import datetime
>>> feed_items = [FeedItem('A Title', datetime.datetime(2016, 11, 10, 10, 15))]
>>> feed_items += [FeedItem('Another Title', datetime.datetime(2016, 11, 10, 10, 20))]
>>> feed = Feed('http://example.org/feed.xml', 'A Blog', feed_items)
>>> feed_serializer.serialize(feed)
{'feed': 'http://example.org/feed.xml',
 'items': [{'pub_date': '2016-11-10T10:15:00', 'title': 'A Title'},
  {'pub_date': '2016-11-10T10:20:00', 'title': 'Another Title'}],
 'name': 'A Blog'}
```

At this point, if we had REST API, we could convert this simple data structure into JSON and return it as the response body.

Validation

This is a great start to building a JSON API, but now we want to reverse the process and accept JSON. When we accept input from the outside, we first need to validate that it well-formed before we begin to work with it.

Since, we have already described our data, including what makes it valid, we can use our existing serializer, just in reverse. So, let's say we are going to create feed item, we can do the following

```
feed_item = {
    'title': 'A Title',
    'pub_date': '2016-11-10T10:15:00',
}
print feed_item_serializer.deserialize(feed_item)
# {'pub_date': datetime.datetime(2016, 11, 10, 10, 15, tzinfo=<iso8601.Utc>), 'title
↪': 'A Title'}
```

At this point, we could take that deserialized input and instantiate a FeedItem oject. If we were using an ORM we could then persist that object to the database.

Error Reporting

Data will not always be valid, and when it isn't valid we should be able to report those errors back the user agent. So, we need a way to catch and present errors.

```
from strainer import ValidationException

feed_item = {
  'title': 'A Title',
}

try:
  feed_item_serializer.deserialize(feed_item)
except ValidationException, e:
  print e.errors

# {'pub_date': ['This field is required']}
```

Here, we catch any possible validation exceptions. When a ValidationException is thrown there is a property on the exception called errors. That will have the reasons why the input is invalid. In a format that is ready to be returned as an API response.

CHAPTER 2

Structures

Strainer exists to convert data structures comprised of rich python objects into simple datastructures ready to be converted into something suitable for HTTP resposes. It also exsists to take those simple data structures back to rich python types, and validate that the data is what it's suppose to be.

The meat of that serialization is strainers structures. They descrbe the entire process from serialization, to validation, to deserialization.

The Basics of Structures

All structures return a *Translator* object. *Translator* objects have only two methods. *.serialize* will turn rich python objects into simple python data structures, and *.deserialize* will validate, and turn simple data structures into righ python types.

You can compose comples serializers by combining a number of structures.

The Field

A field is the smallest structure. It maps one attribute, and one value. That value can be a list, but everything inside the list needs to be the same type.

A field shouldn't be used by its self, but you can define a field by it's self.

```
from strainer import field

a_field = field('a')
```

During serialization this field will map the attribute *a* from a python object to the key *a* in a dict. During deserialization it will map a key *a* from the input to a key *a* in the ouput and validate that the value is correct.

Target Field

Sometimes, the field name in the output isn't always the same as the attribute name in the input. So, you can pass a second optional argument to achieve different names.

```
from strainer import field

a_field = field('a', target_field='z')
```

Now *a_field* will serialize the attribute *a* to the field *z* in the output, and during deserialization the reverse will happen. All structures have the target_field argument.

Validators

When deserializing a structure you can have a series of validators run, validtors server two functions. The first is too convert incoming data into the correct form if possible, and the second is to validate that the incoming data is correct. Validators are always run when deserialization is called, and they are only run during deserialization. Validators are called in order.

```
from strainer import field, validators

a_field = field('a', validators=[validators.required(), validators.string(max_
↪length=10)])
```

Read more about validators see, *Validators*.

Multiple Values

It is possible to declare a field as a list instead of single value. If you do so each value in the list will be validated as a single value. If any fail, the validation will fail.

```
from strainer import multiple_field, validators

a_field = multiple_field('a')
```

Custom Attribute Getter

The default method for geting attributes from objects is to use the *operator.attrgetter* function. You can pass in a different function.

This will attempt to fetch a key from a dict instead of using attrgetter.

```
from strainer import field

a_field = field('a', attr_getter=lambda x: x.get('a'))
```

Format A Value For Serialization

By default the value that is fetched from the attribute of the object is passed forward as-is, but you can format values for serialization by passing in a list of formatters.

```
from strainer import field, validators, formatters

a_field = field('a', validators=[validators.datetime()], formatters=[formatters.
↪format_datetime()])
```

Read more about formatters, see , *Formatters*.

The Dict Field

The dict_field is almost exactly like the field, except that it will attempt to get a key from a dict instead of an attribute from an object.

```
from strainer import dict_field

a_field = dict_field('a')
```

The Child

When creating a serializer, often one will need to model one object nested in another object. This is where the *child* strucutre comes handy. It allows you to nest one serializer in another.

```
from strainer import serializer, field, child

c_serializer = serializer(
  field('c1'),
)

a_serializer = serializer(
  field('b'),
  child('c', serializer=c_serializer),
)
```

Target Field

Sometimes, the field name in the output isn't always the same as the attribute name in the input. So, you can pass a second optional argument to achieve different names.

```
from strainer import serializer, field

c_serializer = serializer(
  field('c1'),
)

a_serializer = serializer(
  field('b'),
  child('c', target_field='a', serializer=c_serializer),
)
```

Now *a_serializer* will serialize the attribute *c* to the field *a* in the output, and during deserialization the reverse will happen.

Validators

Just like the regular field, you can apply validations to a child structure. These validators run before the inner object is deserialized it's self.

In this example you may want to require that the child object exists.

```
from strainer import serializer, field, validators

c_serializer = serializer(
  field('c1'),
)

a_serializer = serializer(
  field('b'),
  child('c', validators=[validators.required()], serializer=c_serializer),
)
```

The Many

The Many structure is like the Child structure. It allows you to nest objects. The Many though allows you to nest an array of values instead of one. Like the child strucutre you can also use validators.

```
from strainer import many, serializer, field, validators

c_serializer = serializer(
  field('c1'),
)

a_serializer = serializer(
  field('b'),
  many('c', validators=[validators.required()], serializer=c_serializer),
)
```

One thing to keep in mind is that the passed validators to many will be passed all the data in the target key. That way you can perform validation over the whole structure. For instance you could limit the length of a list. The full validation will happen before the data is passed to the serialier.

The Serializer

A serializer is composed of any number of Translators, usually produce by other structures like field, child, and many. The serializer returns a translator object that can serializer, and deserialize.

```
from strainer import serializer, field

a_serializer = serializer(
  field('a'),
  field('b'),
)
```

CHAPTER 3

Validators

Validators convert incoming data into the correct format, and also raise excpetions if data is invalid.

Current Validators

integer

Will validate that a value is an integer.

```
>>> from strainer import validators
>>> int_validators = validators.integer()
>>> int_validators('1')
1
```

You can also optionally, clamp an integer to bounds

```
>>> from strainer import validators
>>> int_validators = validators.integer(bounds=(2, 10))
>>> int_validators('1')
2
```

string

Will validate that a value is a string

```
>>> from strainer import validators
>>> string_validators = validators.string()
>>> string_validators(1)
'1'
```

You can also apply a *max_length*. If the string is longer then the *max_length* an exception will be thrown.

```
>>> from strainer import validators
>>> string_validators = validators.string(max_length=100)
```

required

Will validate that a value exists and that it is not falsey. It will accept *0*, but raise an exception on *False*, *None*, *''*, *[]*, and *{}*.

boolean

Will coerce value into either a *True*, or *False* value. *0*, *False*, *None*, *''*, '[]', and *{}* would all count as *False* values, anything else would be *True*.

datetime

This validator will attempt to parse an ISO 8601 string into a python datetime object.

The default timezone is UTC, but you can modify that by passing a *default_tzinfo*.

Custom Validators

A validtora returns a function that will be used to validate a value during serialization. You can use the *export_validator* function to create a custom validation function.

```
from strainer import validators, ValidationException

@validators.export_validator
def my_silly_validators(value, context=None):
    if value == 'An apple':
        raise ValidationException("An apple is not silly")

    return '%s is silly.' % (value)
```

CHAPTER 4

Formatters

Formmatters help fields prepare values for serializaiton. Most formatters accept a value, and a context and return a formatted value.

Current Formatters

format_datetime

This formatter will take a datetime, or a date object and convert it into an ISO8601 string representation.

```
>>> import datetime
>>> from strainer import formatters
>>> dt_formatter = formatters.format_datetime()
>>> dt_formatter(datetime.datetime(1984, 6, 11))
'1984-06-11T00:00:00'
```

Custom Formatters

A formatter returns a function that will be used to format a value before serialization, you could build a silly formatter like this.

```
def custom_formatter():
    def _my_formatter(value, context=None):
        return '%s is silly.' % (value)

    return _my_formatter

my_formatter = custom_formatter()
print my_formatter('A clown')
# A clown is silly
```

In practice it's probably better to use the export_formatter decorator. It's as simple way to create a formatter.

```
from strainer import formatters

@formatters.export_formatter
def my_silly_formatter(value, context=None):
    return '%s is silly.' % (value)
```

It's clear, and their is less nesting.

CHAPTER 5

API

Structure

Use these structures to build up a serializers.

Every structure returns an object that has two methods. *serialize* returns objects ready to be encoded into JSON, or other formats. *deserialize* will validate and return objects ready to be used internally, or it will raise a validation excepton.

class `strainer.structure.`**`Translator`**(*serialize*, *deserialize*)
Translator is an internal data structure that holds a reference to a serialize and deserialize function. All structures return a translator.

`strainer.structure.`**`child`**(*source_field*, *target_field=None*, *serializer=None*, *validators=None*, *attr_getter=None*, *full_validators=None*)
A child is a nested serializer.

`strainer.structure.`**`dict_field`**(**args*, ***kwargs*)
dict_field is just like field except that it pulls attributes out of a dict, instead of off an object.

`strainer.structure.`**`field`**(*source_field*, *target_field=None*, *validators=None*, *attr_getter=None*, *formatters=None*)
Constructs an indvidual field for a serializer, this is on the order of one key, and one value.

The field determines the mapping between keys internaly, and externally. As well as the proper validation at the level of the field.

```
>>> from collections import namedtuple
>>> Aonly = namedtuple('Aonly', 'a')
>>> model = Aonly('b')
>>> one_field = field('a')
>>> one_field.deserialize(model)
{'a': 'b'}
```

Parameters

- **source_field** (*str*) – What attribute to get from a source object
- **target_field** (*str*) – What attribute to place the value on the target, optional. If optional target is equal to source_field
- **validators** (*list*) – A list of validators that will be applied during deserialization.
- **formaters** (*list*) – A list of formaters that will be applied during serialization.
- **attr_getter** (*function*) – Overrides the default method for getting the soure_field off of an object

strainer.structure.**many**(*source_field*, *target_field=None*, *serializer=None*, *validators=None*, *attr_getter=None*)
Many allows you to nest a list of serializers

strainer.structure.**serializer**(**fields*)
This function creates a serializer from a list fo fields

Validators

Validators are functions that validate data.

strainer.validators.**boolean**(**args*, ***kwargs*)
Converts a field into a boolean

strainer.validators.**datetime**(**args*, ***kwargs*)
validates that a a field is an ISO 8601 string, and converts it to a datetime object.

strainer.validators.**integer**(**args*, ***kwargs*)
converts a value to integer, applying optional bounds

strainer.validators.**required**(**args*, ***kwargs*)
validates that a field exists in the input

strainer.validators.**string**(**args*, ***kwargs*)
converts a value into a string, optionally with a max length

Formatters

Formatters are functions that transform data.

strainer.formatters.**format_datetime**(**args*, ***kwargs*)
Formats a value as an iso8601 datetime

Exceptions

This is just a set of utilities to help take a deserialized dict and turn it into JSON. It handles things like datetime objects.

exception strainer.exceptions.**ValidationException**(*errors*)
This exception keeps track of all the exceptions thrown during validations

Python Module Index

S

Index