Lollipop Documentation

Release 0.2

Maxim Kulkin

August 12, 2016

Contents

Release version 0.2. (Changelog)

CHAPTER 1

Guide

1.1 Installation

lollipop requries Python >= 2.6 or <= 3.5. It has no external dependencies other than the Python standard library.

```
$ pip install lollipop
```

1.1.1 Bleeding Edge

To get latest development version of lollipop, run

```
$ pip install git+https://github.com/maximkulkin/lollipop.git
```

1.2 Quickstart

This guide will walk you through the basics of schema definition, data serialization, deserialization and validation.

1.2.1 Declaring Types

Let's start with a your application-level model:

```
class Person(object):
    def __init__(self, name, birthdate):
        self.name = name
        self.birthdate = birthdate

    def __repr__(self):
        return "<Person name={name} birthdate={birthdate}>".format(
            name=repr(self.name), birthdate=repr(self.birthdate),
        )
```

You want to create a JSON API to load and dump it. First you need to define a type for that data:

```
from lollipop.types import Object, String, Date

PersonType = Object({
    'name': String(),
```

```
    'birthdate': Date(),
})
```

1.2.2 Serializing data

To serialize your data, pass it to your type's `dump()` method:

```
from datetime import date
import json

john = Person(name='John', birthdate=date(1970, 02, 29))
john_data = PersonType.dump(john)
print json.dump(john_data, indent=2)
# {
#   "name": "John",
#   "birthdate": "1970-02-29"
# }
```

1.2.3 Deserializing data

To load data back, pass it to your type's `load()` method:

```
user_data = {
    "name": "Bill",
    "birthdate": "1994-08-12",
}

user = PersonType.load(user_data)
print user
# {"name": "Bill", "birthdate": date(1994, 08, 12)}
```

If you want to restore original data type, you can pass it's constructor function when you define your type:

```
PersonType = Object({
    'name': String(),
    'birthdate': Date(),
}, constructor=Person)

print PersonType.load({
    "name": "Bill",
    "birthdate": "1994-08-12",
})
# <Person name="Bill" birthdate=date(1994, 08, 12)>
```

To deserialize a list of objects, you can create a *List* instance with your object type as element type:

```
List(PersonType).load([
    {"name": "Bob", "birthdate": "1980-12-12"},
    {"name": "Jane", birthdate": "1991-08-04"},
])
# => [<Person name="Bob" birthdate=date(1980, 12, 12)>,
      <Person name="Jane" birthdate=date(1991, 08, 04)>]
```

1.2.4 Validation

By default all fields are required to have values, so if you accidentally forget to specify one, you will get a *ValidationError* exception:

```
from lollipop.errors import ValidationError

try:
    PersonType.load({"name": "Bob"})
except ValidationError as ve:
    print ve.messages  # => {"birthdate": "Value is required"}
```

The same applies to field types: if you specify value of incorrect type, you will get validation error.

If you want more control on your data, you can specify additional validators:

```
from lollipop.validators import Regexp

email_validator = Regexp(
    '(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$)',
    error='Invalid email',
)

UserType = Object({
    'email': String(validate=email_validator),
})

try:
    UserType.load({"email": "wasa"})
except ValidationError as ve:
    print ve.messages  # => {"email": "Invalid email"}
```

If you just need to validate date and not interested in result, you can use `validate()` method:

```
print UserType.load({"email": "wasa"})
# => {"email": "Invalid email"}

print UserType.load({"email": "wasa@example.com"})
# => None
```

You can define your own validators:

```
def validate_person(person):
    errors = ValidationErrorBuilder()
    if person.name == 'Bob':
        errors.add_error('name', 'Should not be called Bob')
    if person.age < 18:
        errors.add_error('age', 'Should be at least 18 years old')
    errors.raise_errors()

PersonType = Object({
    'name': String(),
    'birthdate': Date(),
}, validate=validate_person)

PersonType.validate({'name': 'Bob', 'age': 15})
# => {'name': 'Should not be called Bob',
#     'age': 'Should be at least 18 years old'}
```

or use *Predicate* validator and supply a True/False function to it.

Validating cross-field dependencies is easy:

```
def validate_person(person):
    if person.name == 'Bob' and person.age < 18:
        raise ValidationError('All Bobs should be at least 18 years old')
```

1.2.5 Changing The Way Accessing Object Data

When you define an *Object* type, by default it will retrieve object data by accessing object's attributes with the same name as name of the field you define. Most often it is what you want. However sometimes you might want to obtain data differently. To do that, you define object's fields not with *Type* instances, but with *Field* instances.

To access attribute with a different name, use *AttributeField*:

```
MyObject = namedtuple('MyObject', ['other_field'])

MyObjectType = Object({
    'field1': AttributeField(String(), attribute='other_field'),
})
```

To get data from a method instead of an attribute, use *MethodField*:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def get_name(self):
        return self.first_name + ' ' + self.last_name

PersonType = Object({
    'name': MethodField(String(), method='get_name'),
})
```

1.3 Object schemas

1.3.1 Declaration

Object schemas are defined with *Object* class by passing it a dictionary mapping field names to *Type* instances.

So given an object

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

You can define it's type like this:

```
from lollipop.types import Object, String, Integer

PersonType = Object({
    'name': String(),
    'age': Integer(),
})
```

It will allow serializing Person types to Python's basic types (that you can use to serialize to JSON) or validate that basic Python data:

```
PersonType.dump(Person('John', 38))
# => {"name": "John", "age": 38}

PersonType.validate({"name": "John"})
# => {"age": "Value is required"}

PersonType.load({"name": "John", "age": 38})
# => {"name": "John", "age": 38}
```

Yet it loads to same basic type dict instead of real object. To fix that, you need to provide a data constructor to type object:

```
PersonType = Object({
    'name': String(),
    'age': Integer(),
}, constructor=Person)

PersonType.load({"name": "John", "age": 38})
# => Person(name="John", age=38)
```

Constructor function should take field values as keyword arguments and return constructed object.

1.3.2 Value extraction

When you serialize (dump) objects, field values are expected to be object attributes. But library actually allows controlling that. This is done with *Field* class instances. When you define your object and pass types for it's fields, what really happens is those types are wrapped with a *Field* subclass objects. The actual object fields are defined like this:

```
PersonType = Object({
    'name': AttributeField(String()),
    'age': AttributeField(Integer()),
})
```

Passing just a *Type* instances for field types is just a shortcut to wrap them all with a default field type which is *AttributeField*. You can change default field type with `Object.default_field_type` argument:

```
PersonType = Object({
    'name': String(),
    'age': Integer(),
}, default_field_type=AttributeField)
```

And you can actually mix fields defined with just *Type* with fields defined with *Field*. The first ones will be wrapped with default field type while the later ones will be used as is.

AttributeField is probably the one that would be used most of the time. It extracts value for serialization from object attribute with the same name as the field name. You can change the name of attribute to extract value from:

```
Person = namedtuple('Person', ['full_name'])

PersonType = Object({'name': AttributeField(String(), attribute='full_name')})

PersonType.dump(Person('John Doe'))  # => {'name': 'John Doe'}
```

Other useful instances are *MethodField* which calls given method on the object to get value instead of getting attribute, *FunctionField* which uses given function on a serialized object to get value, *ConstantField* which

always serializes to given constant value. For last one there is another shortcut: if you provide a value for a field which is not *Type* and not *Field* then it will be wrapped with a *ConstantField*.

```
# Following lines are equivalent
Object({'answer': ConstantField(Any(), 42)}).dump(object())   # => {'answer': 42}
Object({'answer': 42}).dump(object())   # => {'answer': 42}
```

1.3.3 Object Schema Inheritance

To be able to allow reusing parts of schema, you can supply a base `Object`:

```
BaseType = Object({'base': String()})
InheritedType = Object(BaseType, {'foo': Integer()})

# is the same as
InheritedType = Object({'base': String(), 'foo': Integer()})
```

You can actually supply multple base types which allows using them as mixins:

```
TimeStamped = Object({'created_at': DateTime(), 'updated_at': DateTime()})

BaseType = Object({'base': String()})
InheritedType = Object([BaseType, TimeStamped], {'foo': Integer()})
```

1.3.4 Polymorphic types

Sometimes you need a way to serialize and deserialize values of different types put in the same list. Or maybe you value can be of either one of given types. E.g. you have a graphical application which operates with objects of different shapes:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Shape(object):
    pass

class Circle(Shape):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

class Rectangle(Shape):
    def __init__(self, left_top, right_bottom):
        self.left_top = left_top
        self.right_bottom = right_bottom

PointType = Object({'x': Integer(), 'y': Integer()}, constructor=Point)

CircleType = Object({
    'center': PointType,
    'radius': Integer
}, constructor=Circle)

RectangleType = Object({
```

```
    'left_top': PointType,
    'right_bottom': PointType,
}, constructor=Rectangle)
```

To support that library provides a special type - *OneOf*:

```
def with_type_annotation(subject_type, type_name):
    return Object(subject_type, {'type': type_name},
                  constructor=subject_type.constructor)

AnyShapeType = OneOf(
    {
        'circle': with_type_annotation(CircleType, 'circle'),
        'rectangle': with_type_annotation(RectangleType, 'rectangle'),
    },
    dump_hint=lambda obj: obj.__class__.__name__.lower(),
    load_hint=dict_value_hint('type'),
)

dumped = List(AnyShapeType).dump([
    Circle(Point(5, 8), 4), Rectangle(Point(1, 10), Point(10, 1))
])
# => [
#   {'type': 'circle',
#    'center': {'x': 5, 'y': 8},
#    'radius': 4},
#   {'type': 'rectangle',
#    'left_top': {'x': 1, 'y': 10},
#    'right_bottom': {'x': 10, 'y': 1}}]

List(AnyShapeType).load(dumped)
# => [Circle(Point(5, 8), 4), Rectangle(Point(1, 10), Point(10, 1))]
```

OneOf uses user supplied functions to determine which particular type to use during serialization/deserialization. It helps returning proper error messages. If you're not interested in providing detailed error message, you can just supply all types as a list. *OneOf* will try to use each of them in given order returning first successfull result. If all types return errors it will provide generic error message. Here is example of library's error messages schema:

```
ErrorMessagesType = OneOf([
    String(), List(String()), Dict('ErrorMessages')
], name='ErrorMessages')
```

1.4 Validation

Validation allows to check that data is consistent. It is run on raw data before it is deserialized. E.g. `DateTime` deserializes string to `datetime.datetime` so validations are run on a string before it is parsed. In *Object* validations are run on a dictionary of fields but after fields themselves were already deserialized. So if you had a field of type `DateTime` your validator will get a dictionary with `datetime` object.

Validators are just callable objects that take one or two arguments (first is the data to be validated, second (optional) is the operation context) and raise *ValidationError* in case of errors. Return value of validator is always ignored.

To add validator or validators to a type, you pass them to type contructor's `validate` argument:

```
def is_odd(data):
    if data % 2 == 0:
        raise ValidationError('Value should be odd')
```

```
MyNumber = Integer(validate=is_odd)
MyNumber.load(1)  # => returns 1
MyNumber.load(2)  # => raises ValidationError('Value should be odd')
```

In simple cases you can create a *Predicate* validator for which you need to specify a boolean function and error message:

```
is_odd = Predicate(lambda x: x % 2 != 0, 'Value should be odd')

MyNumber = Integer(validate=is_odd)
```

In more complex cases where you need to parametrize validator with some data it is more convenient to create a validator class:

```
from lollipop.validators import Validator

class GreaterThan(Validator):
    default_error_messages = {
        'greater': 'Value should be greater than {value}'
    }

    def __init__(self, value, **kwargs):
        super(GreaterThan, self).__init__(**kwargs)
        self.value = value

    def __call__(self, data):
        if data <= self.value:
            self._fail('greater', data=data, value=self.value)
```

The last example demonstrates how you can support customizing error messages in your validators: there is a default error message keyed with string 'greater' and users can override it when creating validator with supplying new set of error messages in validator constructor:

```
message = 'Should be greater than answer to the Ultimate Question of Life, the Universe, and Everythi
Integer(validate=GreaterThan(42, error_messages={'greater': message}))
```

1.5 Custom Types

To build a custom type object you can inherit from *Type* and implement functions `load(data, **kwargs)` and `dump(value, **kwargs)`:

```
from lollipop.types import MISSING, String
try:
    from urlparse import urlparse, urljoin
except ImportError:
    from urllib.parse import urlparse, urljoin

class URL(String):
    def _load(self, data, *args, **kwargs):
        loaded = super(URL, self)._load(data, *args, **kwargs)
        return urlparse(loaded)

    def _dump(self, value, *args, **kwargs):
        dumped = urljoin(value)
        return super(URL, self)._dump(dumped, *args, **kwargs)
```

Other variant is to take existing type and extend it with some validations while allowing users to add more validations:

```
from lollipop import types, validators

EMAIL_REGEXP = r"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$)"

class Email(types.String):
    def __init__(self, *args, **kwargs):
        super(Email, self).__init__(*args, **kwargs)
        self._validators.insert(0, validators.Regexp(EMAIL_REGEXP))
```

CHAPTER 2

API Reference

2.1 API Reference

2.1.1 Data

lollipop.types.**MISSING = <MISSING>**
Special singleton value (like None) to represent case when value is missing.

2.1.2 Types

class lollipop.types.**Type**(*validate=None*, **args*, ***kwargs*)
Base class for defining data types.

Parameters **validate** (*list*) – A validator or list of validators for this data type. Validator is a callable that takes serialized data and raises *ValidationError* if data is invalid. Validator return value is ignored.

dump(*value*, *context=None*)
Serialize data to primitive types. Raises *ValidationError* if data is invalid.

Parameters

- **value** – Value to serialize.
- **context** – Context data.

load(*data*, *context=None*)
Deserialize data from primitive types. Raises *ValidationError* if data is invalid.

Parameters

- **data** – Data to deserialize.
- **context** – Context data.

validate(*data*, *context=None*)
Takes serialized data and returns validation errors or None.

Parameters

- **data** – Data to validate.
- **context** – Context data.

class lollipop.types.**Any**(*validate=None*, **args*, ***kwargs*)
Any type. Does not transform/validate given data.

class lollipop.types.**String**(*validate=None*, **args*, ***kwargs*)
A string type.

class lollipop.types.**Integer**(*validate=None*, **args*, ***kwargs*)
An integer type.

num_type
alias of `int`

class lollipop.types.**Float**(*validate=None*, **args*, ***kwargs*)
A float type.

num_type
alias of `float`

class lollipop.types.**Boolean**(*validate=None*, **args*, ***kwargs*)
A boolean type.

class lollipop.types.**List**(*item_type*, ***kwargs*)
A homogenous list type.

Example:

```
List(String()).load(['foo', 'bar', 'baz'])
```

Parameters

- **item_type** (*Type*) – Type of list elements.
- **kwargs** – Same keyword arguments as for *Type*.

class lollipop.types.**Tuple**(*item_types*, ***kwargs*)
A heterogenous list type.

Example:

```
Tuple([String(), Integer(), Boolean()]).load(['foo', 123, False])
```

Parameters

- **item_types** (*list*) – List of item types.
- **kwargs** – Same keyword arguments as for *Type*.

class lollipop.types.**Dict**(*value_types=<Any>*, ***kwargs*)
A dict type. You can specify either a single type for all dict values or provide a dict-like mapping object that will return proper Type instance for each given dict key.

Example:

```
Dict(Integer()).load({'key0': 1, 'key1': 5, 'key2': 15})

Dict({'foo': String(), 'bar': Integer()}).load({
    'foo': 'hello', 'bar': 123,
})
```

Parameters

- **value_type** (*dict*) – A single *Type* for all dict values or mapping of allowed keys to *Type* instances.
- **kwargs** – Same keyword arguments as for *Type*.

class lollipop.types.**OneOf**(*types*, *load_hint=<function type_name_hint>*, *dump_hint=<function type_name_hint>*, **args*, ***kwargs*)

Example:

```
class Foo(object):
    def __init__(self, foo):
        self.foo = foo

class Bar(object):
    def __init__(self, bar):
        self.bar = bar

FooType = Object({'foo': String()}, constructor=Foo)
BarType = Object({'bar': Integer()}, constructor=Bar)

def object_with_type(name, subject_type):
    return Object(subject_type, {'type': name},
                  constructor=subject_type.constructor)

FooBarType = OneOf({
    'Foo': object_with_type('Foo', FooType),
    'Bar': object_with_type('Bar', BarType),
}, dump_hint=type_name_hint, load_hint=dict_value_hint('type'))

List(FooBarType).dump([Foo(foo='hello'), Bar(bar=123)])
# => [{'type': 'Foo', 'foo': 'hello'}, {'type': 'Bar', 'bar': 123}]

List(FooBarType).load([{'type': 'Foo', 'foo': 'hello'},
                       {'type': 'Bar', 'bar': 123}])
# => [Foo(foo='hello'), Bar(bar=123)]
```

lollipop.types.**type_name_hint**(*data*)

Returns type name of given value.

To be used as a type hint in *OneOf*.

lollipop.types.**dict_value_hint**(*key*, *mapper=None*)

Returns a function that takes a dictionary and returns value of particular key. The returned value can be optionally processed by `mapper` function.

To be used as a type hint in *OneOf*.

class lollipop.types.**Field**(*field_type*, **args*, ***kwargs*)

Base class for describing *Object* fields. Defines a way to access object fields during serialization/deserialization. Usually it extracts data to serialize/deserialize and call `self.field_type.load()` to do data transformation.

Parameters **field_type** (`Type`) – Field type.

dump(*name*, *obj*, **args*, ***kwargs*)

Serialize data to primitive types. Raises *ValidationError* if data is invalid.

Parameters

- **name** (*str*) – Name of attribute to serialize.
- **obj** – Application object to extract serialized value from.

load(*name*, *data*, **args*, ***kwargs*)
Deserialize data from primitive types. Raises *ValidationError* if data is invalid.

Parameters

- **name** (*str*) – Name of attribute to deserialize.
- **data** – Raw data to get value to deserialize from.

class lollipop.types.**ConstantField**(*field_type*, *value*, **args*, ***kwargs*)
Field that always serializes to given value and does not deserialize.

Parameters

- **field_type** (Type) – Field type.
- **value** – Value constant for this field.

class lollipop.types.**AttributeField**(*field_type*, *attribute=None*, **args*, ***kwargs*)
Field that corresponds to object attribute.

Parameters

- **field_type** (Type) – Field type.
- **attribute** (*str*) – Use given attribute name instead of field name defined in object type.

class lollipop.types.**MethodField**(*field_type*, *method*, **args*, ***kwargs*)
Field that is result of method invocation.

Example:

```
class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def get_name(self):
        return self.first_name + ' ' + self.last_name

PersonType = Object({
    'name': MethodField(String(), 'get_name'),
}, constructor=Person)
```

Parameters

- **field_type** (Type) – Field type.
- **method** (*str*) – Method name. Method should not take any arguments.

class lollipop.types.**FunctionField**(*field_type*, *function*, **args*, ***kwargs*)
Field that is result of function invocation.

Example:

```
class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

def get_name(person):
    return person.first_name + ' ' + person.last_name

PersonType = Object({
```

```
        'name': FunctionField(String(), get_name),
    }, constructor=Person)
```

Parameters

- **field_type** (*Type*) – Field type.
- **function** (*callable*) – Function that takes source object and returns field value.

class lollipop.types.**Object**(*bases_or_fields=None*, *fields=None*, *constructor=<type 'dict'>*, *default_field_type=<class 'lollipop.types.AttributeField'>*, *allow_extra_fields=True*, *only=None*, *exclude=None*, ***kwargs*)

An object type. Serializes to a dict of field names to serialized field values. Parametrized with field names to types mapping. The way values are obtained during serialization is determined by type of field object in `fields` mapping (see *ConstantField*, *AttributeField*, *MethodField* for details). You can specify either *Field* object, a *Type* object or any other value. In case of *Type*, it will be automatically wrapped with a default field type, which is controlled by `default_field_type` constructor argument. In case of any other value it will be transformed into *ConstantField*.

Example:

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

PersonType = Object({
    'name': String(),
    'age': Integer(),
}, constructor=Person)
PersonType.load({'name': 'John', 'age': 42})
# => Person(name='John', age=42)
```

Parameters

- **base_or_fields** – Either *Object* instance or fields (See `fields` argument). In case of fields, the actual fields argument should not be specified.
- **fields** – List of name-to-value tuples or mapping of object field names to *Type*, *Field* objects or constant values.
- **contructor** (*callable*) – Deserialized value constructor. Constructor should take all fields values as keyword arguments.
- **default_field_type** (*Field*) – Default field type to use for fields defined by their type.
- **allow_extra_fields** (*bool*) – If False, it will raise *ValidationError* for all extra dict keys during deserialization. If True, will ignore all extra fields.
- **only** (*list*) – List of field names to include in this object. All other fields (own or inherited) won't be used.
- **exclude** (*list*) – List of field names to exclude from this object. All other fields (own or inherited) will be included.
- **kwargs** – Same keyword arguments as for *Type*.

class lollipop.types.**Optional**(*inner_type*, *load_default=None*, *dump_default=None*, ***kwargs*)
A wrapper type which makes values optional: if value is missing or None, it will not transform it with an inner type but instead will return None (or any other configured value).

Example:

```
UserType = Object({
    'email': String(),           # by default types require valid values
    'name': Optional(String()),  # value can be omitted or None
    'role': Optional(            # when value is omitted or None, use given value
        String(validate=AnyOf(['admin', 'customer'])),
        load_default='customer',
    ),
})
```

Parameters

- **inner_type** (Type) – Actual type that should be optional.
- **load_default** – Value to use when value is missing on deserialization.
- **dump_default** – Value to use when value is missing on serialization.
- **kwargs** – Same keyword arguments as for *Type*.

class lollipop.types.**LoadOnly**(*inner_type*)
A wrapper type which proxies loading to inner type but always returns *MISSING* on dump.

Example:

```
UserType = Object({
    'name': String(),
    'password': LoadOnly(String()),
})
```

Parameters **inner_type** (Type) – Data type.

class lollipop.types.**DumpOnly**(*inner_type*)
A wrapper type which proxies dumping to inner type but always returns *MISSING* on load.

Example:

```
UserType = Object({
    'name': String(),
    'created_at': DumpOnly(DateTime()),
})
```

Parameters **inner_type** (Type) – Data type.

2.1.3 Validators

class lollipop.validators.**Validator**(*error_messages=None*, **args*, ***kwargs*)
Base class for all validators.

Validator is used by types to validate data during deserialization. Validator class should define `__call__` method with either one or two arguments. In both cases, first argument is value being validated. In case of two arguments, the second one is the context. If given value fails validation, `__call__` method should raise *ValidationError*. Return value is always ignored.

class lollipop.validators.**Predicate**(*predicate*, *error=None*, ***kwargs*)
Validator that succeeds if given predicate returns True.

Parameters

- **predicate** (*callable*) – Predicate that takes value and returns True or False. One- and two-argument predicates are supported. First argument in both cases is value being validated. In case of two arguments, the second one is context.
- **error** (*str*) – Error message in case of validation error. Can be interpolated with `data`.

class lollipop.validators.**Range**(*min=None*, *max=None*, ***kwargs*)
Validator that checks value is in given range.

Parameters

- **min** (*int*) – Minimum length. If not provided, minimum won't be checked.
- **max** (*int*) – Maximum length. If not provided, maximum won't be checked.
- **error** (*str*) – Error message in case of validation error. Can be interpolated with `data`, `min` or `max`.

class lollipop.validators.**Length**(*exact=None*, *min=None*, *max=None*, ***kwargs*)
Validator that checks value length (using `len()`) to be in given range.

Parameters

- **exact** (*int*) – Exact length. If provided, `min` and `max` are not checked. If not provided, `min` and `max` checks are performed.
- **min** (*int*) – Minimum length. If not provided, minimum length won't be checked.
- **max** (*int*) – Maximum length. If not provided, maximum length won't be checked.
- **error** (*str*) – Error message in case of validation error. Can be interpolated with `data`, `length`, `exact`, `min` or `max`.

class lollipop.validators.**NoneOf**(*values*, *error=None*, ***kwargs*)
Validator that succeeds if `value` is not a member of given `values`.

Parameters

- **values** (*iterable*) – A sequence of invalid values.
- **error** (*str*) – Error message in case of validation error. Can be interpolated with `data` and `values`.

class lollipop.validators.**AnyOf**(*choices*, *error=None*, ***kwargs*)
Validator that succeeds if `value` is a member of given `choices`.

Parameters

- **choices** (*iterable*) – A sequence of allowed values.
- **error** (*str*) – Error message in case of validation error. Can be interpolated with `data` and `choices`.

class lollipop.validators.**Regexp**(*regexp*, *flags=0*, *error=None*, ***kwargs*)
Validator that succeeds if `value` matches given `regex`.

Parameters

- **regexp** (*str*) – Regular expression string.
- **flags** (*int*) – Regular expression flags, e.g. re.IGNORECASE. Not used if regexp is not a string.

- **error** (*str*) – Error message in case of validation error. Can be interpolated with `data` and `regexp`.

2.1.4 Errors

`lollipop.errors.`**SCHEMA = '_schema'**
Name of an error key for cases when you have both errors for the object and for it's fields:

```
{'field1': 'Field error', '_schema': 'Whole object error'}
```

exception `lollipop.errors.`**ValidationError**(*messages*)
Exception to report validation errors.

Examples of valid error messages:

```
raise ValidationError('Error')
raise ValidationError(['Error 1', 'Error 2'])
raise ValidationError({
    'field1': 'Error 1',
    'field2': {'subfield1': ['Error 2', 'Error 3']}
})
```

Parameters **messages** – Validation error messages. String, list of strings or dict where keys are nested fields and values are error messages.

class `lollipop.errors.`**ValidationErrorBuilder**
Helper class to report multiple errors.

Example:

```
def validate_all(data):
    builder = ValidationErrorBuilder()
    if data['foo']['bar'] >= data['baz']['bam']:
        builder.add_error('foo.bar', 'Should be less than bam')
    if data['foo']['quux'] >= data['baz']['bam']:
        builder.add_fields('foo.quux', 'Should be less than bam')
    ...
    builder.raise_errors()
```

add_error(*path*, *error*)
Add error message for given field path.

Example:

```
builder = ValidationErrorBuilder()
builder.add_error('foo.bar.baz', 'Some error')
print builder.errors
# => {'foo': {'bar': {'baz': 'Some error'}}}
```

Parameters

- **path** (*str*) – '.'-separated list of field names
- **error** (*str*) – Error message

add_errors(*errors*)
Add errors in dict format.

Example:

```
        builder = ValidationErrorBuilder()
        builder.add_errors({'foo': {'bar': 'Error 1'}})
        builder.add_errors({'foo': {'baz': 'Error 2'}, 'bam': 'Error 3'})
        print builder.errors
        # => {'foo': {'bar': 'Error 1', 'baz': 'Error 2'}, 'bam': 'Error 3'}
```

Parameters **list or dict errors** (*str,*) – Errors to merge

raise_errors()
Raise *ValidationError* if errors are not empty; do nothing otherwise.

lollipop.errors.**merge_errors**(*errors1*, *errors2*)
Deeply merges two error messages. Error messages can be string, list of strings or dict of error messages (recursively). Format is the same as accepted by *ValidationError*. Returns new error messages.

l