lima Documentation

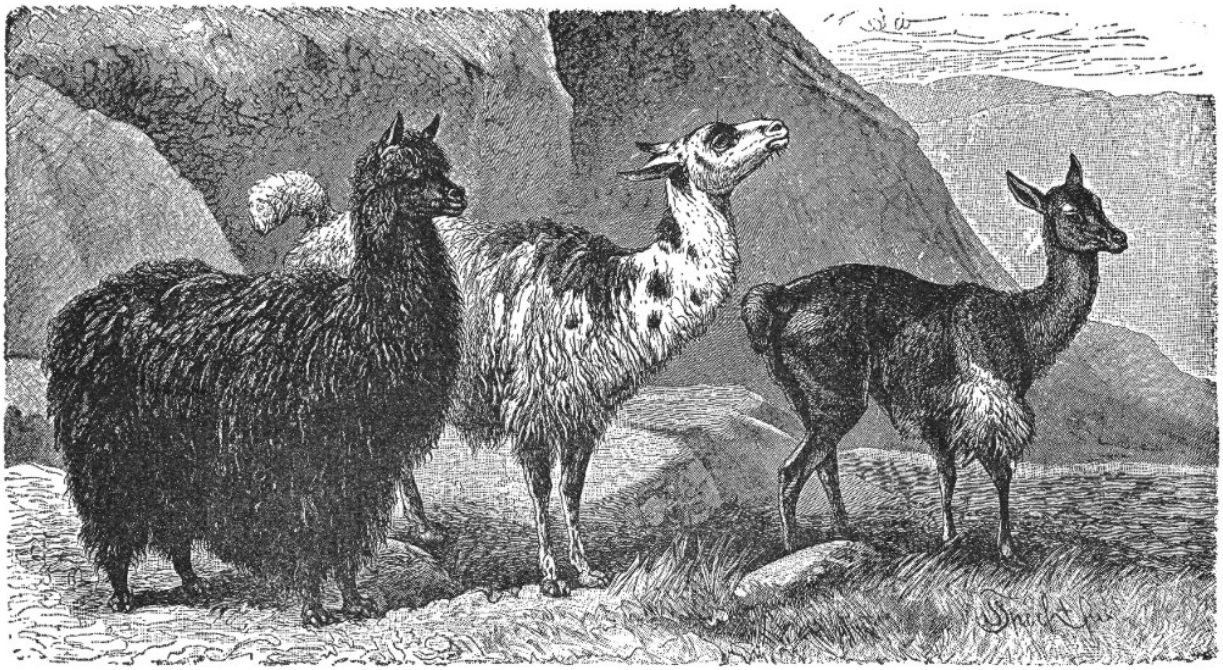
Release 0.2.2

Bernhard Weitzhofer

October 27, 2014

Contents

lima takes arbitrary Python objects and converts them into data structures native to Python. The result can easily be serialized into JSON, XML, and all sorts of other things. lima is Free Software, lightweight and fast.



ALPACA LLAMA VICUNA

CHAPTER 1

Key Features

Lightweight lima has only a few hundred SLOC. lima has no external dependencies.

Fast lima tries to be as fast as possible while still remaining pure Python 3.

Focused lima doesn't try to do an ORM's job and it doesn't try to do a JSON library's job.

Extensible Build complex schemas out of simpler ones. Extend existing fields types or define your own.

Easy to learn lima is not only fast but also fast (and easy) to learn.

Well documented lima has a comprehensive tutorial and more than one line of docstring per line of Python code

Free lima is Free Software, *licensed* under the terms of the MIT license.

CHAPTER 2

lima at a Glance

```
import datetime
import lima

# a model
class Book:
    def __init__(self, title, date_published):
        self.title = title
        self.date_published = date_published

# a marshalling schema
class BookSchema(lima.Schema):
    title = lima.fields.String()
    published = lima.fields.Date(attr='date_published')

book = Book('The Old Man and the Sea', datetime.date(1952, 9, 1))
schema = BookSchema()
schema.dump(book)
# {'published': '1952-09-01', 'title': 'The Old Man and the Sea'}
```

CHAPTER 3

Documentation

3.1 Installation

The recommended way to install lima is via pip.

Just make sure you have at least Python 3.3 and a matching version of pip available and installing lima becomes a one-liner:

```
$ pip install lima
```

Most of the time it's also a good idea to do this in an isolated virtual environment.

Starting with version 3.4, Python handles all of this (creation of virtual environments, ensuring the availability of pip) out of the box:

```
$ python3 -m venv /path/to/my_venv
$ source /path/to/my_venv/bin/activate
(my_venv) $ pip install lima
```

If you should run into trouble, the Tutorial on Installing Distributions from the Python Packaging User Guide might be helpful.

3.2 First Steps

lima tries to be lean, consistent, and easy to learn. Assuming you already have *installed* lima, this section should help you getting started.

Note: Throughout this documentation, the terms *marshalling* and *serialization* will be used synonymously.

3.2.1 A simple Example

Let's say we want to expose our data to the world via a web API and we've chosen JSON as our preferred serialization format. We have defined a data model in the ORM of our choice. It might behave something like this:

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

Our person objects look like this:

```
import datetime
person = Person('Ernest', 'Hemingway', datetime.date(1899, 7, 21))
```

If we want to serialize such person objects, we can't just feed them to Python's `json.dumps()` function: per default it only knows how to deal with a very basic set of data types.

Here's where lima comes in: Defining an appropriate `Schema`, we can convert person objects into data structures accepted by `json.dumps()`.

```
from lima import fields, Schema

class PersonSchema(Schema):
    first_name = fields.String()
    last_name = fields.String()
    date_of_birth = fields.Date()

schema = PersonSchema()
serialized = schema.dump(person)
# {'date_of_birth': '1899-07-21',
#  'first_name': 'Ernest',
#  'last_name': 'Hemingway'}
```

... and to conclude our example:

```
import json
json.dumps(serialized)
# '{"last_name": "Hemingway", "date_of_birth": "1899-07-21", ...
```

3.2.2 First Steps Recap

- You now know how to do basic marshalling (Create a schema class with appropriate fields. Create a schema object. Pass the object(s) to marshal to the schema object's `dump()` method.
- You now know how to get JSON for arbitrary objects (pass the result of a schema object's `dump()` method to `json.dumps()`).

3.3 A closer Look at Fields

Fields are the basic building blocks of a Schema. Even though lima fields follow only the most basic protocol, they are rather powerful.

3.3.1 What Data a Field presents

The `PersonSchema` from the last chapter contains three field objects named *first_name, last_name* and *date_of_birth*. These correspond to a person object's attributes of the same name. What if our model didn't have an attribute `date_of_birth` but an attribute `birthday` instead?

To get the data for our `date_of_birth` field from the model's `birthday` attribute, we have to tell the field by supplying the attribute name via the `attr` argument:

```
import datetime
from lima import Schema, fields
```

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

person = Person('Ernest', 'Hemingway', datetime.date(1899, 7, 21))

class PersonSchema(Schema):
    first_name = fields.String()
    last_name = fields.String()
    date_of_birth = fields.Date(attr='birthday')

schema = PersonSchema()
schema.dump(person)
# {'date_of_birth': '1899-07-21',
#  'first_name': 'Ernest',
#  'last_name': 'Hemingway'}
```

Providing `attr` is the preferred way to deal with attribute names differing from field names, but `attr` is not enough for some other cases. What if we can't get the information we need from a single attribute? Here *getters* come in handy.

A getter in this context is a callable that takes an object (in our case: a person object) and returns the value we're interested in. We tell a field about the getter via the `get` parameter:

```
def sort_name_getter(obj):
    return '{}, {}'.format(obj.last_name, obj.first_name)

class PersonSchema(Schema):
    first_name = fields.String()
    last_name = fields.String()
    sort_name = fields.String(get=sort_name_getter)
    date_of_birth = fields.Date(attr='birthday')

schema = PersonSchema()
schema.dump(person)
# {'date_of_birth': '1899-07-21',
#  'first_name': 'Ernest',
#  'last_name': 'Hemingway'
#  'sort_name': 'Hemingway, Ernest'}
```

Note: For getters, lambda expressions come in handy. `sort_name` could just as well have been defined like this:

```
sort_name = fields.String(
    get=lambda obj: '{}, {}'.format(obj.last_name, obj.first_name)
)
```

`attr` and `get` are *keyword-only arguments* - a relatively uncommon feature of Python 3 that the lima API makes heavy use of.

Keyword-only arguments

Keyword-only arguments can be recognized by their position in a method/function signature: Every argument coming after the varargs argument like `*args` (or after a single `*`) is a keyword-only argument.

A function that is defined as `def foo(*, x, y): pass` *must* be called like this: `foo(x=1, y=2)`; calling `foo(1, 2)` will raise a `TypeError`.

It is the author's opinion that enforcing keyword arguments in the right places makes the resulting code more readable.

For more information about keyword-only arguments, see PEP 3102

3.3.2 How a Field presents its Data

If a field has a static method (or instance method) `pack()`, this method is used to present a field's data. (Otherwise the field's data is just passed through on marshalling. Some of the more basic built-in fields behave that way.)

So by implementing a `pack()` static method (or instance method), we can support marshalling of any data type we want:

```
from collections import namedtuple
from lima import fields, Schema

# a new data type
GeoPoint = namedtuple('GeoPoint', ['lat', 'long'])

# a field class for the new date type
class GeoPointField(fields.Field):
    @staticmethod
    def pack(val):
        ns = 'N' if val.lat > 0 else 'S'
        ew = 'E' if val.long > 0 else 'W'
        return '{}° {}, {}° {}'.format(val.lat, ns, val.long, ew)

# a model using the new data type
class Treasure:
    def __init__(self, name, location):
        self.name = name
        self.location = location

# a schema for that model
class TreasureSchema(Schema):
    name = fields.String()
    location = GeoPointField()

treasure = Treasure('The Amber Room', GeoPoint(lat=59.7161, long=30.3956))
schema = TreasureSchema()
schema.dump(treasure)
# {'location': '59.7161° N, 30.3956° E', 'name': 'The Amber Room'}
```

Or we can change how already supported data types are marshalled:

```
class FancyDate(fields.Date):
    @staticmethod
    def pack(val):
        return val.strftime('%A, the %d. of %B %Y')

class FancyPersonSchema(Schema):
    first_name = fields.String()
    last_name = fields.String()
    date_of_birth = FancyDate(attr='birthday')

schema = FancyPersonSchema()
schema.dump(person)
# {'date_of_birth': 'Friday, the 21. of July 1899',
```

```
#  'first_name': 'Ernest',
#  'last_name': 'Hemingway'}
```

Warning: Make sure the result of your `pack()` methods is JSON serializable (or at least in a format accepted by the serializer of your target format).
Also, don't try to override an existing instance method with a static method. Have a look at the source if in doubt (currently only `lima.fields.Nested` implements `pack()` as an instance method.

3.3.3 Data Validation

In short: *There is none.*

lima is opinionated in this regard. It assumes you have control over the data you want to serialize and have already validated it *before* putting it in your database.

But this doesn't mean it can't be done. You'll just have to do it yourself. The `pack()` method would be the place for this:

```
import re

class ValidEmailField(fields.String):
    @staticmethod
    def pack(val):
        if not re.match(r'[^@]+@[^@]+\.[^@]+', val):
            raise ValueError('Not an email address: {!r}'.format(val))
        return val
```

Note: If - depending on demand and developer time - lima ever gets deserialization support, sensible validation of incoming data would be a key component of this feature.

If you need full-featured validation of your existing data at marshalling time, have a look at marshmallow.

3.3.4 Fields Recap

- You now know how a field gets its data (in order of precedence: getter > `attr` parameter > field name).
- You know how a field presents its data (`pack()` method).
- You know how to support your own data types (subclass `lima.fields.Field`) and implement `pack()`
- And you know how to change the marshalling of already supported data types (subclass the appropriate field class and override `pack()`)
- Also, you're able to implement data validation should the need arise (implement/override `pack()`).

3.4 Working with Schemas

Now that we know about fields, let's focus on schemas:

3.4.1 Defining Schemas

We already know how to define schemas: subclass `lima.Schema` (the shortcut for `lima.schema.Schema`) and add fields as class attributes.

But there's more to schemas than this. First of all – schemas are composible:

```
from lima import Schema, fields

class PersonSchema(Schema):
    first_name = fields.String()
    last_name = fields.String()

class AccountSchema(Schema):
    login = fields.String()
    password_hash = fields.String()

class UserSchema(PersonSchema, AccountSchema):
    pass

list(UserSchema.__fields__)
# ['password_hash', 'login', 'last_name', 'first_name']
```

Secondly, it's possible to *remove* fields from subclasses that are present in superclasses. This is done by setting a special class attribute `__lima_args__` like so:

```
class UserProfileSchema(UserSchema):
    __lima_args__ = {'exclude': ['last_name', 'password_hash']}

list(UserProfileSchema.__fields__)
# ['login', 'first_name']
```

If there's only one field to exclude, you don't have to put its name inside a list - lima does that for you:

```
class NoLastNameSchema(UserSchema):
    __lima_args__ = {'exclude': 'last_name'}  # string instead of list

list(NoLastNameSchema.__fields__)
# ['password_hash', 'login', 'first_name']
```

And finally, we can't just *exclude* fields, we can *include* them too. So here is a user schema with fields provided via `__lima_args__`:

```
class UserSchema(Schema):
    __lima_args__ = {
        'include': {
            'first_name': fields.String(),
            'last_name': fields.String(),
            'login': fields.String(),
            'password_hash': fields.String()
        }
    }

list(UserSchema.__fields__)
# ['password_hash', 'last_name', 'first_name', 'login']
```

Note: It's possible to mix and match all those features to your heart's content. lima tries to fail early if something doesn't add up.

Note: The inheritance and precedence rules for fields are intuitive, but should there ever arise the need for clarification, you can read about how a schema's fields are determined in the documentation of `lima.schema.SchemaMeta`.

3.4.2 Automated Schema Definition

Validating ORM agnosticism for a moment, let's see how we could utilize `__lima_args__['include']` to create our Schema automatically.

We start with this SQLAlchemy model (skip this section if you don't want to install SQLAlchemy):

```
import sqlalchemy as sa
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Account(Base):
    __tablename__ = 'accounts'
    id = sa.Column(sa.Integer, primary_key=True)
    login = sa.Column(sa.String)
    password_hash = sa.Column(sa.String)
```

`lima.fields` defines a mapping `lima.fields.type_mapping` of some Python types to field classes. We can utilize this as follows:

```
from lima import fields

def fields_for_model(model):
    result = {}
    for name, col in model.__mapper__.columns.items():
        field_class = fields.type_mapping[col.type.python_type]
        result[name] = field_class()
    return result
```

Defining lima schemas becomes a piece of cake now:

```
from lima import Schema

class AccountSchema(Schema):
    __lima_args__ = {'include': fields_for_model(Account)}

AccountSchema.__fields__
# {'id': <lima.fields.Integer at 0x...>,
#  'login': <lima.fields.String at 0x...>,
#  'password_hash': <lima.fields.String at 0x...>}
```

... and of course you still can manually add, exclude or inherit anything you like.

Warning: Neither `lima.fields.type_mapping` nor the available field classes are as exhaustive as they should be. Expect above code to fail on slightly exotic column types. There is still work to be done.

3.4.3 Schema Objects

Up until now we only ever needed a single instance of a schema class to marshal the fields defined in this class. But schema objects can do more.

Providing the keyword-only argument `exclude`, we may exclude certain fields from being serialized. This saves the need to define lots of almost similar schema classes:

```
import datetime
from lima import Schema, fields

# again, our model
class Person:
    def __init__(self, first_name, last_name, birthday):
        self.first_name = first_name
        self.last_name = last_name
        self.birthday = birthday

# again, our schema
class PersonSchema(Schema):
    first_name = fields.String()
    last_name = fields.String()
    date_of_birth = fields.Date(attr='birthday')

# again, our person
person = Person('Ernest', 'Hemingway', datetime.date(1899, 7, 21))

# as before, for reference
person_schema = PersonSchema()
person_schema.dump(person)
# {'date_of_birth': '1899-07-21',
#  'first_name': 'Ernest',
#  'last_name': 'Hemingway'}

birthday_schema = PersonSchema(exclude=['first_name', 'last_name'])
birthday_schema.dump(person)
# {'date_of_birth': '1899-07-21'}
```

The same thing can be achieved via the `only` keyword-only argument:

```
birthday_schema = PersonSchema(only='date_of_birth')
birthday_schema.dump(person)
# {'date_of_birth': '1899-07-21'}
```

You may have already guessed: both `exclude` and `only` take lists of field names as well as simple strings for a single field name – just like `__lima_args__['exclude']`.

3.4.4 Marshalling Collections

Consider this:

```
persons = [
    Person('Ernest', 'Hemingway', datetime.date(1899, 7, 21)),
    Person('Virginia', 'Woolf', datetime.date(1882, 1, 25)),
    Person('Stefan', 'Zweig', datetime.date(1881, 11, 28)),
]
```

Instead of looping over this collection ourselves, we can ask the schema object to do this for us - either for a single call (by specifying `many=True` to the `dump()` method), or for every call of `dump()` (by specifying `many=True` to the schema's constructor):

```
person_schema = PersonSchema(only='last_name')
person_schema.dump(persons, many=True)
```

```
# [{'last_name': 'Hemingway'},
#  {'last_name': 'Woolf'},
#  {'last_name': 'Zweig'}]

many_persons_schema =  PersonSchema(only='last_name', many=True)
many_persons_schema.dump(persons)
# [{'last_name': 'Hemingway'},
#  {'last_name': 'Woolf'},
#  {'last_name': 'Zweig'}]
```

3.4.5 Schema Recap

- You now know how to compose bigger schemas from smaller ones (inheritance of schema classes).
- You know how to exclude certain fields from schemas (`__lima_args__['exclude']`).
- You know three different ways to add fields to schemas (class attributes, `__lima_args__['include']` and inheriting from other schemas)
- You are now able to create schemas automatically (`__lima_args__['include']` with some model-specific code)
- You can fine-tune what gets dumped by a schema object (`only` and `exclude` keyword-only arguments) and you can serialize collections of objects (`many=True`)

3.5 Nested Data

Most ORMs represent linked objects nested under an attribute of the linking object. As an example, lets model the relationship between a book and its author:

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

# A book links to its author via a nested Person object
class Book:
    def __init__(self, title, author=None):
        self.title = title
        self.author = author

person = Person('Ernest', 'Hemingway')
book = Book('The Old Man and the Sea')
book.author = person
```

3.5.1 One-way Relationships

Currently, this relationship is one way only: *From* a book *to* its author. The author doesn't know anything about books yet (well, in our model at least).

To serialize this construct, we have to tell lima that a `Book` object has a `Person` object nested inside, designated via the `author` attribute.

For this we use a field of type `lima.fields.Nested` and tell lima what data to expect by providing the `schema` parameter:

```
from lima import fields, Schema

class PersonSchema(Schema):
    first_name = fields.String()
    last_name = fields.String()

class BookSchema(Schema):
    title = fields.String()
    author = fields.Nested(schema=PersonSchema)

schema = BookSchema()
schema.dump(book)
# {'author': {'first_name': 'Ernest', 'last_name': 'Hemingway'},
#  'title': The Old Man and the Sea'}
```

Along with the mandatory keyword-only argument `schema`, `lima.fields.Nested` accepts the optional keyword-only-arguments we already know (`attr` or `get`). All other keyword arguments provided to `lima.fields.Nested` get passed through to the constructor of the nested schema. This allows us to do stuff like the following:

```
class BookSchema(Schema):
    title = fields.String()
    author = fields.Nested(schema=PersonSchema, only='last_name')

schema = BookSchema()
schema.dump(book)
# {'author': {'last_name': 'Hemingway'},
#  'title': The Old Man and the Sea'}
```

3.5.2 Two-way Relationships

If not only a book should link to its author, but an author should also link to his/her bestselling book, we can adapt our model like this:

```
# authors link to their bestselling book
class Author(Person):
    def __init__(self, first_name, last_name, bestseller=None):
        super().__init__(first_name, last_name)
        self.bestseller = bestseller

# books link to their authors
class Book:
    def __init__(self, title, author=None):
        self.title = title
        self.author = author

author = Author('Ernest', 'Hemingway')
book = Book('The Old Man and the Sea')
book.author = author
author.bestseller = book
```

If we want to construct schemas for models like this, we will have to adress two problems:

1. **Definition order:** If we define our `AuthorSchema` first, its `bestseller` attribute will have to reference a `BookSchema` - but this doesn't exist yet, since we decided to define `AuthorSchema` first. If we decide to define `BookSchema` first instead, we run into the same problem with its `author` attribute.

2. **Recursion:** An author links to a book that links to an author that links to a book that links to an author that links to a book that links to an author that links to a book that links to an author that links to a book that links to an author `RuntimeError: maximum recursion depth exceeded`

lima makes it easy to deal with those problems:

To overcome the problem of recursion, just exclude the attribute on the other side that links back.

To overcome the problem of definition order, lima supports lazy evaluation of schemas. Just pass the *qualified name* (or the *fully module-qualified name*) of a schema class to `lima.fields.Nested` instead of the class itself:

```
class AuthorSchema(PersonSchema):
    bestseller = fields.Nested(schema='BookSchema', exclude='author')

class BookSchema(Schema):
    title = fields.String()
    author = fields.Nested(schema=AuthorSchema, exclude='bestseller')

author_schema = AuthorSchema()
author_schema.dump(author)
# {'first_name': 'Ernest',
#  'last_name': 'Hemingway',
#  'bestseller': {'title': The Old Man and the Sea'}

book_schema = BookSchema()
book_schema.dump(book)
# {'author': {'first_name': 'Ernest', 'last_name': 'Hemingway'},
#  'title': The Old Man and the Sea'}
```

On class names

For referring to classes via their name, the lima documentation only ever talks about two different kinds of class names: the *qualified name* (*qualname* for short) and the *fully module-qualified name*:

The qualified name This is the value of the class's `__qualname__` attribute. Most of the time, it's the same as the class's `__name__` attribute (except if you define classes within classes or functions ...). If you define `class Foo: pass` at the top level of your module, the class's qualified name is simply *Foo*. Qualified names were introduced with Python 3.3 via PEP 3155

The fully module-qualified name This is the qualified name of the class prefixed with the full name of the module the class is defined in. If you define `class Qux: pass` within a class `Baz` (resulting in the qualified name *Baz.Qux*) at the top level of your `foo.bar` module, the class's fully module-qualified name is *foo.bar.Baz.Qux*.

Warning: If you define schemas in local namespaces (at function execution time), their names become meaningless outside of their local context. For example:

```
def make_schema():
    class FooSchema(Schema):
        foo = fields.String()
    return FooSchema

schemas = [make_schema() for i in range(1000)]
```

Which of those one thousend schemas would we refer to, would we try to link to a `FooSchema` by name? To avoid ambiguity, lima will refuse to link to schemas defined in local namespaces.

By the way, there's nothing stopping us from using the idioms we just learned for models that link to themselves - everything works as you'd expect:

```
class MarriedPerson(Person):
    def __init__(self, first_name, last_name, spouse=None):
        super().__init__(first_name, last_name)
        self.spouse = spouse

class MarriedPersonSchema(PersonSchema):
    spouse = fields.Nested(schema='MarriedPersonSchema', exclude='spouse')
```

3.5.3 One-to-many and many-to-many Relationships

Until now, we've only dealt with one-to-one relations. What about one-to-many and many-to-many relations? Those link to collections of objects.

We know the necessary building blocks already: Providing additional keyword arguments to `lima.fields.Nested` passes them through to the specified schema's constructor. And providing `many=True` to a schema's construtor will have the schema marshalling collections - so:

```
# authors link to their books now
class Author(Person):
    def __init__(self, first_name, last_name, books=None):
        super().__init__(first_name, last_name)
        self.books = books

author = Author('Virginia', 'Woolf')
author.books = [
    Book('Mrs Dalloway', author),
    Book('To the Lighthouse', author),
    Book('Orlando', author)
]

class AuthorSchema(PersonSchema):
    books = fields.Nested(schema='BookSchema', exclude='author', many=True)

class BookSchema(Schema):
    title = fields.String()
    author = fields.Nested(schema=AuthorSchema, exclude='books')

schema = AuthorSchema()
schema.dump(author)
# {'books': [{'title': 'Mrs Dalloway'},
#            {'title': 'To the Lighthouse'},
#            {'title': 'Orlando'}],
#  'last_name': 'Woolf',
#  'first_name': 'Virginia'}
```

3.5.4 Nested Data Recap

- You now know how to marshal nested objects (via a field of type `lima.fields.Nested`)
- You know about lazy evaluation of nested schemas and how to specify those via qualified and fully module-qualified names.
- You know how to implement two-way relationships between objects (pass `exclude` or `only` to the nested schema through `lima.fields.Nested`)

- You know how to marshal nested collections of objects (pass `many=True` to the nested schema through `lima.fields.Nested`)

3.6 The lima API

Please note that the lima API uses a relatively uncommon feature of Python 3: *Keyword-only arguments.*

Keyword-only arguments

Keyword-only arguments can be recognized by their position in a method/function signature: Every argument coming after the varargs argument like `*args` (or after a single `*`) is a keyword-only argument.

A function that is defined as `def foo(*, x, y): pass` *must* be called like this: `foo(x=1, y=2)`; calling `foo(1, 2)` will raise a `TypeError`.

It is the author's opinion that enforcing keyword arguments in the right places makes the resulting code more readable.

For more information about keyword-only arguments, see PEP 3102

3.6.1 lima.fields

Field classes and related code.

`lima.fields.`**`type_mapping`** **=dict(...)**
dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

class `lima.fields.`**`Boolean`**(*, *attr=None*, *get=None*)
A boolean field.

currently this class has no additional functionality compared to `Field`. Nevertheless it should be used over `Field` when referencing boolean values as an indicator for a field's type and to keep code future-proof.

class `lima.fields.`**`Date`**(*, *attr=None*, *get=None*)
A date field.

static **`pack`**(*val*)
Return a string representation of `val`.

Parameters **val** – The `datetime.date` object to convert.

Returns The ISO 8601-representation of `val` (`YYYY-MM-DD`).

class `lima.fields.`**`DateTime`**(*, *attr=None*, *get=None*)
A DateTime field.

static **`pack`**(*val*)
Return a string representation of `val`.

Parameters **val** – The `datetime.datetime` object to convert.

Returns The ISO 8601-representation of `val` (`YYYY-MM-DD%HH:MM:SS.mmmmmm+HH:MM` for `datetime.datetime` objects with Timezone information and microsecond precision).

class `lima.fields.`**`Field`**`(*, attr=None, get=None)`
Base class for fields.

Parameters

- **attr** – The optional name of the corresponding attribute.
- **get** – An optional getter function accepting an object as its only parameter and returning the field value.

When a `Field` has both `attr` *and* `get` defined (either via passing them to the constructor of `Field` or because the subclass implements `attr` or `get` on the class level, `get` *always* takes precedence.

If neither `get` nor `attr` are defined (not per instance and not at the class level), `lima.schema.Schema.dump()` tries to get the field value by looking for an attribute of the same name the Field has within the corresponding `lima.schema.Schema` instance.

class `lima.fields.`**`Float`**`(*, attr=None, get=None)`
A float field.

currently this class has no additional functionality compared to `Field`. Nevertheless it should be used over `Field` when referencing float values as an indicator for a field's type and to keep code future-proof.

class `lima.fields.`**`Integer`**`(*, attr=None, get=None)`
An integer field.

currently this class has no additional functionality compared to `Field`. Nevertheless it should be used over `Field` when referencing integer values as an indicator for a field's type and to keep code future-proof.

class `lima.fields.`**`Nested`**`(*, schema, attr=None, get=None, **kwargs)`
A Field referencing another object with it's respective schema.

Parameters

- **schema** – The schema of the referenced object. This can be specified via a schema *object,* a schema *class* (that will get instantiated immediately) or the qualified *name* of a schema class (for when the named schema has not been defined at the time of the `Nested` object's creation). If two or more schema classes with the same name exist in different modules, the schema class name has to be fully module-qualified (see the *entry on class names* for clarification of these concepts). Schemas defined within a local namespace can not be referenced by name.
- **attr** – The optional name of the corresponding attribute.
- **get** – An optional getter function accepting an object as its only parameter and returning the field value.
- **kwargs** – Optional keyword arguments to pass to the `Schema`'s constructor when the time has come to instance it. Must be empty if `schema` is a `lima.schema.Schema` object.

Raises `ValueError` – If `kwargs` are specified even if `schema` is a `lima.schema.Schema` *object.*

Examples:

```
# refer to PersonSchema class
author = Nested(schema=PersonSchema)

# refer to PersonSchema class with additional params
```

```
artists = Nested(schema=PersonSchema, exclude='email', many=True)

# refer to PersonSchema object
author = Nested(schema=PersonSchema())

# refer to PersonSchema object with additional params
# (note that Nested() gets no kwargs)
artists = Nested(schema=PersonSchema(exclude='email', many=true))

# refer to PersonSchema per name
author = Nested(schema='PersonSchema')

# refer to PersonSchema per name with additional params
author = Nested(schema='PersonSchema', exclude='email', many=True)

# refer to PersonSchema per module-qualified name
# (in case of ambiguity)
author = Nested(schema='project.persons.PersonSchema')

# specify attr name as well
user = Nested(attr='login_user', schema=PersonSchema)
```

pack(*val*)

Return the output of the referenced object's schema's dump method.

If the referenced object's schema was specified by name at the `Nested` field's creation, this is the time when this schema is instantiated (this is done only once).

Parameters **val** – The nested object to convert.

Returns The output of the referenced `lima.schema.Schema`'s `lima.schema.Schema.dump()` method.

class `lima.fields.`**`String`**(*, *attr=None*, *get=None*)

A string field.

currently this class has no additional functionality compared to `Field`. Nevertheless it should be used over `Field` when referencing string values as an indicator for a field's type and to keep code future-proof.

3.6.2 lima.schema

Schema class and related code.

class `lima.schema.`**`Schema`**(*, *exclude=None*, *only=None*, *many=False*)

Base class for Schemas.

Parameters

- **exclude** – A sequence of field names to be removed from the fields of the new `Schema` instance. If only one field is to be removed, it's ok to supply a simple string instead of a list containing only one string for `exclude`. `exclude` may not be specified together with `only`.
- **only** – A sequence of the names of the only fields that shall remain for the new `Schema` instance. If just one field is to remain, it's ok to supply a simple string instead of a list containing only one string for `only`. `only` may not be specified together with `exclude`.
- **many** – A boolean indicating if the new Schema will be serializing single objects (`many=False`) or collections of objects (`many=True`) per default. This can later be overridden in the `dump()` Method.

Upon creation, each Schema object gets an internal mapping of field names to fields.

This mapping starts out as a copy of the classes `__fields__` attribute. (Note that the fields themselves are not copied - changing a field for a Schema instance changes this field for the class and all base classes as well. This behaviour might change in the future. In general, it's good practice *not* to change fields once created.)

The internal mapping and is then modified depending the arguments supplied to the `Schema`'s constructor:

For an explanation on how the class's `__fields__` attribute is determined, see `SchemaMeta`.

Also upon creation, each Schema object gets an individually created dump function that aims to unroll most of the loops and to minimize the number of attribute lookups, resulting in a little speed gain on serialization.

`Schema` classes defined outside of local namespaces can be referenced by name (used by `lima.fields.Nested`).

dump(*obj*, *, *many=None*)

Return a marshalled representation of obj.

Parameters

- **obj** – The object (or collection of objects) to marshall.
- **many** – Wether obj is a single object or a collection of objects. If `many` is `None`, the value of the instance's `many` attribute is used.

Returns A representation of `obj` in the form of a JSON-serializable dict, with each entry corresponding to one of the `Schema`'s fields. (Or a list of such dicts in case a collection of objects was marshalled)

class `lima.schema.`**SchemaMeta**

Metaclass of Schema.

Note: The metaclass `SchemaMeta` is used internally to simplify the configuration of new `Schema` classes. For users of the library there should be no need to use `SchemaMeta` directly.

When defining a new `Schema` (sub)class, `SchemaMeta` makes sure that the new class has a class attribute `__fields__` of type `dict` containing the fields for the new Schema.

`__fields__` is determined like this:

- The `__fields__` dicts of all base classes are copied (with base classes specified first having precedence). (Note that the fields themselves are not copied - changing an inherited field changes this field for all the base classes as well. This behaviour might change in the future. In general, it's good practice *not* to change fields once created.)
- Fields (Class variables of type `lima.abc.FieldABC`) are moved out of the class dict and into the `__fields__` dict.
- If present, the class attribute `__lima_args__` is removed from the class dict and evaluated as follows:
  - Fields specified via an optional dict `__lima_args__['include']` are added (overriding any fields of the same name present therein).
  - Fields named in an optional sequence `__lima_args__['exclude']` are removed. If only one field is to be removed, it's ok to supply a simple string instead of a list containing only one string.

`SchemaMeta` also makes sure the new Schema class is registered with the lima class registry `lima.registry` (at least if the Schema isn't defined inside a local namespace, where we wouldn't find it later on).

3.6.3 lima.exc

The lima exception hierarchy.

Currently this module only holds Exceptions related to `lima.registry`, but this might change in the future.

exception `lima.exc.`**`AmbiguousClassNameError`**
Raised when asking for a class with an ambiguous name.

Usually this is the case if two or more classes of the same name were registered from within different modules, and afterwards a registry is asked for one of those classes without specifying the module in the class name.

exception `lima.exc.`**`ClassNotFoundError`**
Raised when a class was not found by a registry.

exception `lima.exc.`**`RegisterLocalClassError`**
Raised when trying to register a class defined in a local namespace.

exception `lima.exc.`**`RegistryError`**
The base class for all registry-related exceptions.

3.6.4 lima.abc

Abstract base classes for fields and schemas.

Note: `lima.abc` is needed to avoid circular imports of fields needing to know about schemas and vice versa. The base classes are used for internal type checks. For users of the library there should be no need to use `lima.abc` directly.

---

class `lima.abc.`**`FieldABC`**
Abstract base class for fields.

Inheriting from `FieldABC` marks a class as a field for internal type checks.

(Usually, it's a *way* better Idea to subclass `lima.fields.Field` directly)

class `lima.abc.`**`SchemaABC`**
Abstract base class for schemas.

Inheriting from `SchemaABC` marks a class as a schema for internal type checks.

(Usually, it's a *way* better Idea to subclass `lima.schema.Schema` directly)

3.6.5 lima.registry

Module for the Registry class.

Note: A global `lima.registry.Registry` object is used internally to allow instances of `lima.fields.Nested` to reference by name `lima.schema.Schema` classes that have not been defined at the time of referencing. For users of the library there should be no need to use anything within `lima.registry` directly.

`lima.registry.`**`global_registry`** **=lima.registry.Registry()**
A class registry.

class `lima.registry.`**`Registry`**
A class registry.

get(*name*)
Get a registered class by its name and return it.

Parameters **name** – The name of the class to look up. Has to be either the class's qualified name or the class's fully module-qualified name in case two classes with the same qualified name from different modules were registered (see the *entry on class names* for clarification of these concepts). Schemas defined within a local namespace can not be referenced by name.

Returns The specified class.

Raises

- `ClassNotFoundError` – If the specified class could not be found (see `lima.exc.ClassNotFoundError`).
- `AmbiguousClassNameError` – If more than one class was found. Usually this can be fixed by using a fully module-qualified class name (see `lima.exc.AmbiguousClassNameError`).

register(*cls*)
Register a class.

Parameters **cls** – The class to register. Must not have been defined in a local namespace.

Raises `RegisterLocalClassError` – In case `cls` is a class defined in a local namespace (see `exc.RegisterLocalClassError`).

3.7 Project Info

lima was started in 2014 by Bernhard Weitzhofer.

3.7.1 Acknowledgements

lima is heavily inspired by marshmallow, from which it lifts most of its concepts from.

Other than that, the author believes to have benefited a lot from looking at the documentation and source code of other awesome projects, among them (in alphabetical order):

- django
- morepath
- SQLAlchemy

as well as the Python standard library itself. (Seriously, look in there!)

3.7.2 About the Image

The Vicuña is the smallest and lightest camelid in the world. In this 1914 illustration [1], it is depicted next to its bigger and heavier relatives, the Llama and the Alpaca.

Despite its delicate frame, the Vicuña is perfectly adapted to the harsh conditions in the high alpine regions of the Andes. It is a mainly wild animal long time believed to never have been domesticated. Reports of Vicuñas breathing fire are greatly exaggerated.

[1] Beach, C. (Ed.). (1914). The New Student's Reference Work. Chicago: F. E. Compton and Company (via Wikisource).

3.8 Changelog

3.8.1 0.2.2 (2014-10-27)

- Fix issue with package not uploading to PYPI
- Fix tiny issues with illustration

3.8.2 0.2.1 (2014-10-27)

- Fix issues with docs not building on readthedocs.org

3.8.3 0.2 (2014-10-27)

- Initial release

3.9 License

l