Kylie Documentation

Release 0.3.1

Mark Smith

June 05, 2015

Contents

Kylie provides mappings between JSON data structures and Python objects. It provides a reasonable amount of power with only a tiny bit of magic, and it has 100% code coverage.

Contents:

CHAPTER 1

Installation

At the command line:

```
$ pip kylie
```

Kylie is 100% Python and has no dependencies, so it should be nice and easy.

CHAPTER 2

Usage

Kylie's design is based on Django's ORM, so it may look pretty familiar. To use Kylie Models in a project, first import it. The API is pretty small:

```
from kylie import Model, Attribute, Relation
```

The simplest use is to extend `Model`, and attach some `Attribute` instances:

```
class Animal(Model):
    id = Attribute()
    name = Attribute()
```

If you have a dictionary of JSON types, you can deserialize it into an *Animal* instance as follows:

```
daisy_pig = Animal.deserialize({
    'id': 1234,
    'name': 'Daisy',
})
```

This will give you an object with a bunch of attributes you can query, such as `daisy_pig.id` and `daisy_pig.name`.

We can do the opposite, by calling serialize on an instantiated `Animal` instance:

```
>>> daisy_data = daisy_pig.serialize()
>>> daisy_data
{
    'id': 1234,
    'name': 'Daisy',
}
```

Instantiating a `Model` with a bunch of data can be done with keyword params:

```
daisy_pig = Animal(animal_id=1234, name='Daisy')
```

You can then call `serialize` on this to return a dict containing the object's data.

But that's not very interesting, so let's see what else we can do.

2.1 Name Mapping

`id` is not a very good attribute name in Python. So we probably want to map the JSON attribute's key to something like `animal_id`, so let's try that:

```
class Animal(Model):
    animal_id = Attribute('id')
    name = Attribute()
```

Now if we run the `deserialize` call above, then `daisy_pig` will have an attribute called `animal_id` instead of `id`. Result!

This is particularly nice if you're mapping a bunch of keys that use *javaNamingConvention* to *python_naming_convention*.

2.2 Type Mapping

So we can do name-mapping, but what about type-mapping? For example, JSON doesn't support timestamps unless they're stored as numbers or formatted strings, but that's not very nice in Python, where we have (slightly) nicer `datetime` objects.

Can Kylie do the mapping for you? You bet! You'll need a function that converts from the serialized form to the Python type, and another that does the reverse mapping though. Let's define those:

```
from datetime import datetime, timedelta

def dt_to_milliseconds(dt):
    epoch = datetime.utcfromtimestamp(0)
    delta = dt - epoch
    return int(delta.total_seconds() * 1000.0)

def milliseconds_to_dt(millis):
    epoch = datetime.utcfromtimestamp(0)
    return epoch + timedelta(seconds=millis / 1000.0)
```

And now we create an `Attribute` using the `python_type` and `serialized_type` parameters:

```
class Animal(Model):
    animal_id = Attribute('id')
    name = Attribute()
    birth_date = Attribute(python_type=milliseconds_to_dt,
                           serialized_type=dt_to_milliseconds)
```

Now you can do the following:

```
>>> daisy_pig = Animal.deserialize({
...     'id': 1234, 'name': 'Daisy', 'birth_date': 1428870071656
... })
>>> daisy_pig.birth_date
datetime.datetime(2015, 4, 12, 20, 21, 11, 656000)
```

2.3 Nested Models

If a person drives a car, you can define the following:

```
class Car(Model):
    color = Attribute()

class Person(Model):
```

```
    name = Attribute()
    car = Relation(Car)
```

The following will now work:

```
>>> maggie = Person.deserialize({
...     'name': 'Margaret',
...     'car': {
...         'color': 'red'
...     }
... })

>>> maggie.car
<__main__.Car instance as #123455>

>>> maggie.car.color
'red'
```

2.4 Nested Sequences

If a car has multiple wheels, you can store them in an embedded sequence:

```
class Wheel(Model):
    front = Attribute()
    side = Attribute()

class Car(Model):
    wheels = Relation(Wheel, sequence=True)
```

Now you can store and lists of Wheels with your car:

```
>>> reliant_robin = Car.deserialize({
...     'wheels': [
...         dict(front=True, side='Middle'),
...         dict(front=False, side='Left'),
...         dict(front=False, side='Right'),
...     ]
... })

>>> reliant_robin.wheels
[<__main__.Wheel at 0x10306bdd0>,
 <__main__.Wheel at 0x10306ba50>,
 <__main__.Wheel at 0x10306bb90>]
```

- The Reliant Robin was a 3-wheeled car.

2.5 Type Choices

Sometimes you need to determine the type of an input dictionary at runtime. Often the dictionary will contain a special attribute, called *type*, *__type__* or *class* (or something else) that tells the deserializer how to deserialize the dictionary into an object.

2.5.1 Contrived Example

A *PetOwner* class contains a *pet* attribute that can either be an instance of *Cow* or an instance of *Dog*. The type is indicated by a '__type__' attribute on the serialized dictionary.

```
class TypedModel(Model):
    """
    Models must be stored with an extra attribute for MappedModelChoice
    to work.
    """
    model_type = None

    def post_serialize(self, d):
        d['__type__'] = self.model_type


class Cow(TypedModel):
    model_type = 'cow'


class Dog(TypedModel):
    model_type = 'dog'
    wagging = Attribute()


class PetOwner(Model):
    """ A class that either has a cow or a dog as a pet. """

    # MappedModelChoice defaults to using the '__type__' attribute, and
    # takes a map of __type__ value -> deserialization class.
    pet = Relation(MappedModelChoice({
        'cow': Cow,
        'dog': Dog
    }))
```

Now you can deserialize the following:

```
data = {
    'pet': {'__type__': 'dog', 'wagging': True}
}
pet_owner = PetOwner.deserialize(data)
```

If you have more complex logic for choosing a class for deserialization, you can extend *BaseModelChoice* and implement the *choose_model* method.

2.6 What else should I know?

If a value in the input dict is `None`, it will be set to `None` in the deserialized object. There's no way to ensure a value is non-None.

If an attribute is missing from the input dict, `deserialize` will fail with an exception. There is currently no way to flag an attribute as 'possibly missing'. It's on the list.

Currently, Kylie doesn't do any validation of anything. If you get an exception that seems like a bad fit, please raise an issue on GitHub.

CHAPTER 3

kylie package

3.1 Module contents

kylie - A module for mapping between JSON and Python classes.

class kylie.**Attribute**(*struct_name=None*, *python_type=None*, *serialized_type=None*)
Bases: object

Used to define a persistent attribute on a Model subclass.

Parameters

- **struct_name** (*str, optional*) – The dict key to be used when serializing this attribute. Defaults to the name the attribute's name on its host Model.
- **python_type** (*function, optional*) – A function that takes the serialized value and converts it to the type that will be stored on the Model instance. This parameter is the usually used with, and is the opposite of serialized_type.
- **serialized_type** (*function, optional*) – A function that takes the value stored on the Model instance and returns the value that should be stored in the serialized dict. This parameter is the usually used with, and is the opposite of python_type.

pack(*instance*, *record*)
Store the attribute on the provided dictionary, *record*.

struct_name
The name of the attribute when it is persisted.

This is either calculated from the attribute's name on the Model it is assigned to, or provided by the constructor's struct_name parameter.

unpack(*instance*, *value*)
Unpack the data item and store on the instance.

class kylie.**BaseModelChoice**
Bases: object

Abstract class for providing Model class choice on deserialization.

If records contain an attribute that specify the class of the serialized object, then you should probably use MappedModelChoice instead. If your logic is more complex then subclass BaseModelChoice and and implement choose_model.

Instances of BaseModelChoice can be passed to the Attribute constructor in place of a Model class.

choose_model(*value*)
Return a Model class suitable for deserializing the given *value*.

Params: value: A data item.

Return: An object (or class, usually a Model) with a deserialize method that can deserialize the given *value*

deserialize(*value*)
Deserialize `value` into a dynamically chosen `Model`.

The chosen Model is specified by the abstract method `choose_model`.

exception `kylie.`**DeserializationError**
Bases: `exceptions.Exception`

Error indicating deserialization failed.

class `kylie.`**MappedModelChoice**(*type_map*, *attribute_name='__type__'*)
Bases: `kylie.kylie.BaseModelChoice`

choose_model(*value*)
Choose a `Model` for deserialization.

This implementation chooses a Model based on the value of the attribute specified by the `attribute_name` this `MappedModelChoice` was instantiated with. The value of this attribute is looked up in this instance's `type_map` and the resulting `Model` class is returned.

If the value of the special attribute is not found in `type_map`, a `DeserializationError` is raised.

class `kylie.`**Model**(**args*, ***kwargs*)
Bases: `object`

A parent class that can map to and from JSON-style data structures.

classmethod deserialize(*record*)
Extract the data from a dict into this Model instance.

serialize()
Extract this model's Attributes into a dict.

class `kylie.`**Relation**(*deserializable*, *struct_name=None*, *sequence=False*)
Bases: `kylie.kylie.Attribute`

An Attribute that embeds to another Model.

Params:

deserializable (Model, BaseModelChoice): The Model or BaseModelChoice subclass that will be deserialized into this attribute.

struct_name (str, optional): The name of the key that will be used when serializing this attribute into a dict. Defaults to the name of the attribute on the host Model.

sequence (bool, optional): Indicates that this attribute will store a sequence of `deserializables`, which will be serialized to a list.

pack(*instance*, *record*)
Serialize the provided *instance* into the provided dict *record*.

unpack(*instance*, *value*)
Unpack an embedded, serialized Model.

Create a new instance of the *relation_class* and deserialize the provided value into it.

CHAPTER 4

Contributing

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

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/judy2k/kylie/issues.

If you are reporting a bug, please include:

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

4.1.2 Fix Bugs

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

4.1.3 Implement Features

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

4.1.4 Write Documentation

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

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/judy2k/kylie/issues.

If you are proposing a feature:

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

4.2 Get Started!

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

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

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

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

```
$ mkvirtualenv kylie
$ cd kylie/
$ python setup.py develop
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 kylie tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

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

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

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

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

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/judy2k/kylie/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_kylie
```

CHAPTER 5

Credits

5.1 Development Lead

- Mark Smith <mark.smith@practicalpoetry.co.uk>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 0.3.0 (2015-06-05)

- MappedModelChoice & BaseModelChoice for determining Model to deserialize at runtime.
- Change to internal Attribute interface (will lead to minor version bump)

  `_apply_model` has been replaced with a direct set of `attr_name`.
- Minor code quality improvements.
- Documentation improvements.

6.2 0.2.0 (2015-04-22)

- Added list support to Relation with `sequence=True` parameter.

6.3 0.1.1 (2015-04-12)

- Removed print statement inside class constructor.

6.4 0.1.0 (2015-04-12)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

k

A

B

C

D

K

M

P

R

S

U