json2py Documentation

Release 0.1a

Víctor Cabezas

April 07, 2016

Contents

The goal of this module is to map plain `json` strings and `json.loads` structures into Python objects.

CHAPTER 1

Contents:

1.1 Usage

Note: For extended and more in depth examples, refer to *Examples*

The following example illustrates how this module works.

```
from json2py.models import *
class Example(NestedField):
    hello = TextField(name = 'hi')
    integer = IntegerField()
    floating = FloatField()

class ExampleList(ListField):
    __model__ = Example

dict_var = {'hi': 'world', 'integer': 1000, 'floating': 10.5, 'ignored': "you won't see me"}
list_var = [dict_var] * 3

myMappedList = ExampleList(list_var)

myMappedList[1].integer.value = 1234

print myMappedList.json_encode(indent = 4)
```

Should return something like:

```
[
    {
        "integer": 1000,
        "floating": 10.5,
        "hi": "world"
    },
    {
        "integer": 1234,
        "floating": 10.5,
        "hi": "world"
    },
    {
        "integer": 1000,
        "floating": 10.5,
```

```
        "hi": "world"
    }
]
```

1.1.1 Models

Note: The classes of this modules are intended to be reimplemented in order to make use of this module.

Models represent basic JSON data types. The usage of this models is intended to be subclassed in order to fully map the original JSON structure.

BaseField

class json2py.models.**BaseField**(*value=None*, *name=None*, *required=True*)
Base Class holding and defining common features for all the other subclasses.

Parameters

- **value** – Value to be stored
- **name** – Name of the field in source data.
- **required** – Whether raise LookupError when key is missing or not.

Note This class must be treated as abstract class and should not be reimplemented.

json_decode(*data*, ***kwargs*)
Parses a JSON-string into this object. This method is intended to build the JSON to Object map, so it doesn't return any value, instead, the object is built into itself.

Parameters

- **data** – JSON-string passed to `json.loads()`
- **kwargs** – Parameters passed to `json.loads()`

json_encode(***kwargs*)
Converts an object of class *BaseField* into JSON representation (string) using `BaseEncoder` JSONEncoder.

Parameters **kwargs** – Parameters passed to `json.dumps()`

Returns JSON-string representation of this object.

TextField

class json2py.models.**TextField**(*value=None*, *name=None*, *required=True*)
Class representing a string field in JSON.

Parameters

- **value** – It is the raw data that is this object will represent once parsed.
- **name** – It has the same meaning as in *BaseField*
- **required** – It has the same meaning as in *BaseField*

Raises **ParseException** – If `value` is not a string nor None

IntegerField

class json2py.models.**IntegerField**(*value=None*, *name=None*, *required=True*)
Class representing an integer field in JSON.

Parameters

- **value** – It is the raw data that is this object will represent once parsed.
- **name** – It has the same meaning as in *BaseField*
- **required** – It has the same meaning as in *BaseField*

Raises **ParseException** – If value is not a integer nor None

FloatField

class json2py.models.**FloatField**(*value=None*, *name=None*, *required=True*)
Class representing a float field in JSON.

Parameters

- **value** – It is the raw data that is this object will represent once parsed.
- **name** – It has the same meaning as in *BaseField*
- **required** – It has the same meaning as in *BaseField*

Raises **ParseException** – If value is not a float nor None

BooleanField

class json2py.models.**BooleanField**(*value=None*, *name=None*, *required=True*)
Class representing boolean field in JSON.

Parameters

- **value** – It is the raw data that is this object will represent once parsed.
- **name** – It has the same meaning as in *BaseField*
- **required** – It has the same meaning as in *BaseField*

Raises **ParseException** – If value is not boolean nor None

NestedField

class json2py.models.**NestedField**(*value=None*, *name=None*, *required=True*)
Class representing a document field in JSON.

Parameters

- **value** – It is the raw data that is this object will represent once parsed.
- **name** – It has the same meaning as in *BaseField*
- **required** – It has the same meaning as in *BaseField*

Raises

- **ParseException** – If value is not a dict nor None
- **InvalidAttribute** – If a reserved keyword is used as attribute

Note Reserved keywords are: `name`, `value` and `required`

Note For use cases and examples refer to Examples

ListField

class `json2py.models.`**`ListField`**(*value=None*, *name=None*, *required=True*)

Class representing a list field in JSON. This class implements `list` interface so you can slicing, appending, popping, etc.

Parameters

- **`name`** – It has the same meaning as in *`BaseField`*
- **`value`** – It is the raw data that is this object will represent once parsed.
- **`required`** – It has the same meaning as in *`BaseField`*

Raises **`ParseException`** – If `value` is not a list nor None

Note Hinting the structure of values of the list should be done using the meta variable `__model__` inside class reimplementation.

Note JSON lists' values can be of any type even in the same list, but in real world apps, every JSON lists' values should be of the same type, this behaviour also simplifies this module, so this class expects that all values in lists must have the same structure.

DateField

class `json2py.models.`**`DateField`**(*value=None*, *name=None*, *required=True*, *formatting='%Y-%m-%dT%H:%M:%SZ'*)

Class used to parse and represent dates. It makes use of `datetime` and `dateutil`.

Parameters

- **`name`** – It has the same meaning as in *`BaseField`*
- **`value`** – It is the raw data that is this object will represent once parsed.
- **`required`** – It has the same meaning as in *`BaseField`*
- **`formatting`** – Format used to represent date.

Raises **`ParseException`** – If `value` is not valid nor None

Note Several format's can be in `formatting`: **auto**: use `dateutil.parser.parse()`, **timestamp**: provide UNIX timestamp and use `datetime.datetime.utcfromtimestamp()` and **custom string**: use any format in compliance with `datetime.datetime.strptime()` valid formats.

1.2 Exceptions

1.2.1 ParseException

class `json2py.models.`**`ParseException`**

Exception raised when an error occur trying to parse data on any of *`BaseField`* subclasses.

1.3 Examples

In the examples below, we will try to learn how to model JSON with `json2py`‘. We will cover how to re-utilize models into bigger ones, like JSON support sub-documents. We will also learn how to model a list of JSON documents.

1.3.1 Modeling Github API

For the examples we will try to model Github’s public API (or at least a part of it). We will be model the *user* response from https://api.github.com/users/{user} using my user account, we will model this *repo* information on https://api.github.com/users/{user}/{repo_name} And with a bit more effort we will model the *repo* listing on https://api.github.com/users/{user}/repos. In this example, `requests` module will be used for simplicity, but the way of requesting remote resources is up to you.

Let’s begin!

User modelling

The user data used on this example will be extracted from https://api.github.com/users/wiston999

Let’s suppose we want to grab the user’s id, login, url, type and if user is admin. This task can be done with the following code.

We will map the *type* key into a field named *user_type* into our model.

Listing 1.1: models1.py

```
from json2py.models import *

class User(NestedField):
    login = TextField()
    id = IntegerField()
    url = TextField()
    user_type = TextField(name = 'type')
    site_admin = BooleanField()
```

And we are all done! Now let’s request the Github’s user info endpoint.

Listing 1.2: example1.py

```
import requests
from models import User

response = requests.get('https://api.github.com/users/wiston999')
my_user = User(response.json())
print (my_user.login.value, "'s stats:")
print ("id:", my_user.id.value)
print ("login:", my_user.login.value)
print ("url:", my_user.url.value)
print ("type:", my_user.user_type.value)
print ("site_admin:", my_user.site_admin.value)
```

Output after executing this code is

```
Wiston999 's stats:
id: 1099504
login: Wiston999
```

```
url: https://api.github.com/users/Wiston999
type: User
site_admin: False
```

This is how modeling works, all you have to do is define class variables into the class inheriting from *json2py.models.NestedField*.

Repository modeling

The next step will be modeling a repository information from Github. We will use the information from this repository, https://api.github.com/repos/wiston999/json2py. We want to get the id, name, full_name, is_private, description, size, language, default_branch and the owner fields. One can notice that owner nested document looks familiar, as it shares several fields with the data on https://api.github.com/users/wiston999. We notice too that shared data is already modeled into the previous example, so, let's use a bit of code re-utilization.

Listing 1.3: models2.py

```
class User(NestedField):
    login = TextField()
    id = IntegerField()
    url = TextField()
    user_type = TextField(name = 'type')
    site_admin = BooleanField()

class Repo(NestedField):
    id = IntegerField()
    name = TextField()
    full_name = TextField()
    owner = User()
    is_private = BooleanField(name = 'private')
    description = TextField()
    size = IntegerField()
    language = TextField()
    default_branch = TextField()
```

Notice how the **owner** field is an instance of **User** class defined above.

Let's try these models

Listing 1.4: example2.py

```
import requests
from models import User, Repo

response = requests.get('https://api.github.com/repos/wiston999/json2py')
this_repo = Repo(response.json())
print (this_repo.name.value, "'s stats:")
print ("id:", this_repo.id.value)
print ("full_name:", this_repo.full_name.value)
print ("owner:", this_repo.owner.login.value)
print ("private:", this_repo.is_private.value)
print ("description:", this_repo.description.value)
print ("language:", this_repo.language.value)
print ("default_branch:", this_repo.default_branch.value)
```

Will output

```
json2py 's stats:
id: 54333024
full_name: Wiston999/json2py
owner: Wiston999
private: False
description: Convert JSON/dict to python object and viceversa
language: Python
default_branch: master
```

Repository list modeling

As a last example, lest loop the loop, we are going to model the data returned by https://api.github.com/users/Wiston999/repos request. We see that this is a list of repositories, which we have already modeled, so, this should be as simple as

Listing 1.5: models3.py

```
class User(NestedField):
    login = TextField()
    id = IntegerField()
    url = TextField()
    user_type = TextField(name = 'type')
    site_admin = BooleanField()

class Repo(NestedField):
    id = IntegerField()
    name = TextField()
    full_name = TextField()
    owner = User()
    is_private = BooleanField(name = 'private')
    description = TextField()
    size = IntegerField()
    language = TextField()
    default_branch = TextField()

class RepoList(ListField):
    __model__ = Repo
```

Everything done! Let's try it

Listing 1.6: example3.py

```
import requests
from models import RepoList

response = requests.get('https://api.github.com/users/wiston999/repos')
user_repo_list = RepoList(response.json())
print ("wiston999's repositories:")
for repo in user_repo_list:
    print ("Repository name:", repo.name.value, "with id:", repo.id.value, "written in", repo.languag
    print ("Repository Owner:", repo.owner.login.value)
    print ('-'*70)
```

And the output

```
wiston999 repositories:
Repository name: BRTMT with id: 24468609 written in JavaScript
Repository Owner: Wiston999
-----------------------------------------------------------------------
Repository name: cursoJS with id: 14053600 written in JavaScript
Repository Owner: Wiston999
-----------------------------------------------------------------------
Repository name: DDSBox with id: 36035006 written in Java
Repository Owner: Wiston999
-----------------------------------------------------------------------
Repository name: DSS with id: 20038644 written in Python
Repository Owner: Wiston999
-----------------------------------------------------------------------
Repository name: ISIII with id: 3630135 written in None
Repository Owner: Wiston999
-----------------------------------------------------------------------
Repository name: json2py with id: 54333024 written in Python
Repository Owner: Wiston999
-----------------------------------------------------------------------
Repository name: Plataforma with id: 2506501 written in Python
Repository Owner: Wiston999
-----------------------------------------------------------------------
Repository name: repos-git with id: 20038280 written in Python
Repository Owner: Wiston999
-----------------------------------------------------------------------
```

1.3.2 Using Python's reserved keywords

When the need of model JSON or dict keys that are Python's keywords too (like from, in, for, etc.), one cannot do

```
class BadKeyword(NestedField):
    from = IntegerField()
```

as it raises `SyntaxError`. A workaround to solve this is use the `name` parameter declared in *`json2py.models.BaseField`*, so the previous can be solved with the following code

```
class BetterKeyword(NestedField):
    from_ = IntegerField(name = 'from')
```

Once `name` parameter is used, the name of variable can be anything distinct to Python's keywords and previous variable names.

1.4 Changelog

- **0.1**: First version, base implementation done. Added docs and tests.
- **0.2**: Added DateField, added optional fields, fixed some bugs.
- **0.3**: Fixed Python's 3 compatibility. Added some examples.
- **0.4**: Check if reserved words are used inside NestedField definition. Added example to README.md. Some bug fixes.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

j

B

D

F

I

J

L

N

P

T