RedisORM

Release 0.1.0

January 08, 2016

Contents

RedisORM is just a quick and simple little Key-Value group to Python Object mapper that makes it easier to have somewhat more complex structures in Redis. While a similar structure could be achieved by using a Redis hash, this module also allows for lists and future support for Redis data structures is hopefully planed, making this more helpful than basic hashs.

There is a small test suite provided. It requires an actual Redis install that is up and running. If you want too change the address then please take a look in the test directory. The tests are automatically ran each commit, thanks to travis-ci.org and coverage is provided by Coveralls.io and this documentation is kindly hosted and automatically rebuilt by readthedocs.org.

If you find this project helpful and would like to make a small donation, I'm available on Gittip:

CHAPTER 1

A Few Minor Warnings

1. This is a very early release, and although I've been using a large part of this code for about a year now, things are still going to break and not function well. Don't be afraid to submit a bug report or a patch on Github to fix something.
2. I'm only a second year university student, and software isn't even my major; I'm working towards an Electrical and Computer Engineering degree, so not only do I have limited time to keep this maintained, but I also probably won't write the best code ever.
3. This project follows the semantic versioning specs. All Minor and patch versions will not break the major versions API, however a bump of the major version signifies that backwards compatibility will most likely be broken in some way.

CHAPTER 2

Quick start

This module provides a basic object mapper for groups of Redis keys.

Note: The model stores all its data with a Redis key structure like so: namespace:key:part

Where:

1. namespace - the key prefix
2. key - the actual name or id of this object
3. part - the specific element of the model

Basic use is like so:

```
>>> import redis
>>> from redisORM import RedisModel
>>> redis_instance = redis.StrictRedis("localhost", db=0)
```

Lets create a new model:

```
>>> sample1 = RedisModel(namespace="test", key="sample1", conn=redis_instance)
>>> sample1
<RedisModel.RedisModel ...>
```

And now lets add some data to it. Two strings and a list:

```
>>> sample1.name = "Ludwig Van Beethoven"
>>> sample1["era"] = "Classical" # you can also treat it like a dictionary (with some missing feature
>>> sample1.famous_works = ["Symphony No.5", "Symphony No.7", "Symphony No.9"] # Lists have limited s
```

Along with setting data you can also access data, both like an object property or by using the dictionary index style:

```
>>> sample1["name"]
'Ludwig Van Beethoven'
>>> sample1.era
'Classical'
>>> sample1.famous_works
['Symphony No.5', 'Symphony No.7', 'Symphony No.9']
```

You can also check for a property in the model:

```
>>> "name" in sample1
True
>>> "age" in sample1
False
```

```
>>> "Symphony No.9" in sample1.famous_works
True
```

CHAPTER 3

Contributing

All code for this can be found online at github. If something is broken, or a feature is missing, please submit a pull request or open an issue. Most things I probably won't have time to get around to looking at too deeply, so if you want it fixed, a pull request is the way to go. In your pull request please provide an explanation as to what your request is for, and what benefit it provides. Also, please try to match the style of the code, or make sure your code is nearly all PEP8 compliant just to maintain code consistency.

Besides that, this project is licensed under the MIT License as found in the `LICENSE.txt` file. Enjoy!

CHAPTER 4

Documentation

4.1 Exceptions

This library will only raise a subclass of `RedisORMException` if it encounters a problem.

class `redisORM.redis_model.`**`RedisORMException`**

The general exception class which is raised by this module. Nothing special.

4.2 Model Class

Besides `RedisORMException`, this Model class should be the only other class you need to use in this library. It acts as a simple *dict* style object which will back all its data in Redis.

class `redisORM.redis_model.`**`RedisModel`**(*namespace=None*, *key=None*, *conn=None*, ***kwargs*)

Emulates a python *object* for the data stored in the collection of keys which match this models, in Redis. Raw data from the redis is stored in *_data* which is a `RedisKeys` instance. This allows for the black magic which makes this class store changes in realtime to redis.

This object has a *__repr__* method which can be used with print or logging statements. It will give the id and a representation of the internal *_data* `RedisKeys` for debugging purposes.

`namespace` **= None**

`key` **= None**

`conn` **= None**

`finish_init`()

A hook called at the end of the main *__init__* to allow for custom inherited classes to customize their init process without having to redo all of the existing int. This should accept nothing besides *self* and nothing should be returned.

`get`(*attr*, *default=None*)

Acts like a *dict.get()* where it will return a default if no matching value was found for the given key.

Parameters

- **`attr`** – The key to look for. If this is found then its value is returned, otherwise *default* is returned.
- **`default`** – The default to return if no match was found.

classmethod **`new`**(*id=None*, ***kwargs*)
: Creates a new instance, filling out the models data with the keyword arguments passed, so long as those keywords are not in the protected items array.

`delete`()
: Deletes the current instance, if its in the database (or try).

`protected_items`
: Provides a cleaner interface to dynamically add items to the models list of protected functions to not store in the database

4.3 Helper Classes

These classes help make `RedisModel` function smoothly and allow for the easy addition of new Redis data structures. Most of the time you should have a need for these classes, although admittedly, the `RedisList` does come in handy once in a while for a *list* like object that backs its data in an actual Redis list.

class `redisORM.redis_model.`**`RedisList`**(*key*, *conn*, *start=[]*, *reset=False*)
: Attempts to emulate a python *list*, while backing the list in redis. This supports most of the common *list* functions, except as noted.

Generally speaking, you won't have to create an instance of this class, however if you are working with a *list* then this is the class you'll get back, not a *list* class.

Note: Most notably, this is currently missing the sort and reverse functions.

`sync`()

`listToInt`()

`append`(*other*)

`prepend`(*other*)

`extend`(*other*)

`insert`(*index*, *elem*)

`remove`(*elem*)

`pop`()

`lpop`()

`index`(*elem*)

`count`()

`reset`()

class `redisORM.redis_model.`**`RedisKeys`**(*key*, *namespace='‘*, *conn=None*)
: Where the realtime syncing and updating takes place.

A *dict* like object which is used as the backing data store for `RedisModel`.

Aka: The Source of Magic

`delete`()
: Deletes all the keys from redis along with emptying the objects internal *_data* dict, then deleting itself at the end of it all.

`get` (*part*)

Retrieves a part of the model from redis and stores it.

Parameters **`part`** – The part of the model to retrieve.

Raises RedisORMException If the redis type is different from string or list (the only two supported types at this time.)

`get_default` (*part*, *default=None*)

Works just like a *dict*'s *get()* method, returning the default if no matching key was found.

Parameters

- **`part`** – The key which to look for
- **`default`** – The default to return if no match was found