redis-limpyd Documentation

Release 0.1.0

Yohan Boniface

December 16, 2015

Contents

1 Show me some code! 3

2 Contents 5

2.1 About . . . 5
2.2 Database . . . 5
2.3 Models . . . 6
2.4 Fields . . . 8
2.5 Collections . . . 19
2.6 Contrib . . . 21

3 Indices and tables 35

Idea is to provide an **easy** way to store objects in Redis, **without losing the power and the control of the Redis API**, in a *limpid* way. So, scope is to provide just as abstraction as needed.

Featuring:

- Don't care about keys, *limpyd* do it for you
- Retrieve objects from some of their attributes
- Retrieve objects collection
- CRUD abstraction
- Keep the power of all the Redis data types in your own code

Source code: https://github.com/yohanboniface/redis-limpyd

CHAPTER 1

Show me some code!

Example of configuration:

```
from limpyd import model

main_database = model.RedisDatabase(
    host="localhost",
    port=6379,
    db=0
)

class Bike(model.RedisModel):

    database = main_database

    name = model.InstanceHashField(indexable=True, unique=True)
    color = model.InstanceHashField()
    wheels = model.StringField(default=2)
```

So you can use it like this:

```
>>> mountainbike = Bike(name="mountainbike")
>>> mountainbike.wheels.get()
'2'
>>> mountainbike.wheels.incr()
>>> mountainbike.wheels.get()
'3'
>>> mountainbike.name.set("tricycle")
>>> tricycle = Bike.collection(name="tricycle")[0]
>>> tricycle.wheels.get()
'3'
>>> tricycle.hmset(color="blue")
True
>>> tricycle.hmget('color')
['blue']
>>> tricycle.hmget('color', 'name')
['blue', 'tricycle']
>>> tricycle.color.hget()
'blue'
>>> tricycle.color.hset('yellow')
True
>>> tricycle.hmget('color')
['yellow']
```

CHAPTER 2

Contents

2.1 About

redis-limpyd is a project initiated by Yohan Boniface, using python to store "models" in Redis.

The project can be found here: https://github.com/yohanboniface/redis-limyd

Yohan is helped in the developement by Stéphane "Twidi" Angel, with lot of work on his branches, aimed to be integrated upstream via pull requests when we have time to review code. You can found these branches here: https://github.com/twidi/redis-limpyd/branches (the *develop* branch is generally up to date with the work on all of them)

If you want to help, please fork (*master* or a feature branch, not *develop*) and work on a branch with a comprehensive name, write tests (seriously, everything is severely tested in *limpyd*) and make a pull request.

2.1.1 Install

Python version 2.6, 2.7, 3.3 and 3.4 are supported. Redis-py versions >= 2.9.1, < 2.11 are supported.

```
pip install redis-limpyd
```

2.2 Database

The first element to define when using *limpyd* is the database. The main goal of the database is to handle the connection to Redis and to host the models.

It's easy to define a database, as its arguments are the same as for a standard connection to Redis via redis-py:

```
from limpyd.database import RedisDatabase

main_database = RedisDatabase(host='localhost', port=6379, db=0)
```

Then it's also easy to define the database (which is mandatory) on which a model is defined:

```
class Example(model.RedisModel):
    database = main_database
    some_field = fields.StringField()
```

If you have more than one model to host on a database, it's a good idea to create an abstract model:

```
class BaseModel(model.RedisModel):
    database = main_database
    abstract = True

class Foo(BaseModel):
    foo_field = fields.StringField()

class Bar(BaseModel):
    bar_field = fields.StringField()
```

Note that you cannot have two models with the same name (the name of the class) in the same database (for obvious collusion problems), but we provide a namespace attribute on models to resolve this problem (so you can use an external module with models named as yours). See `models` to know how to use them.

It's not a good idea to declare many RedisDatabase objects on the same Redis database (defined with host+port+db), because of obvious colusion problems if models have the same name in each. So do it only if you really know what you're doing, and with different models only.

You want to change the database used after the models being created. It can be useful if you want to use models defined in an external module. To manage this, simply use the *use_database* method of a model class.

Say you use an external module defined like this:

```
class BaseModel(RedisModel):
    database = RedisDatabase()
    abstract = True

class Foo(BaseModel):
    # ... fields ...

class Bar(BaseModel):
    # ... fields ...
```

In your code, to add these models to your database (which also allow to use them in *Related model*), simply do:

```
database = RedisDatabase(**connection_settings)
BaseModel.use_database(database)
```

You can notice that you don't have to call this method on *Foo* and *Bar*. It's because they are subclasses of *BaseModel* and they don't have another database defined.

If you simply want to change the settings of the redis connection to use (different server or db), you can use the *connect* method of your database, which accepts the same parameters as the constructor:

```
main_database = RedisDatabase(host='localhost', port=6379, db=0)

# ... later ...

main_database.connect(host='localhost', port=6370, db=3)
```

2.3 Models

Models are the core of limpyd, it's why we're here. A *RedisModel* is a class, in a database, with some fields. Each instance of this model is a new object stored in Redis by *limpyd*.

Here a simple example:

```
class Example(model.RedisModel):
    database = main_database

    foo = field.StringField()
    bar = field.StringField()
```

To create an instance, it's as easy as:

```
>>> example = Example(foo='FOO', bar='BAR')
```

By just doing this, the fields are created, and a *PKField* is set with a value that you can use:

```
>>> print "New example object with pk #%s" % example.pk.get()
New example object with pk #1
```

Then later to get an instance from Redis with it's pk, it's as simple as:

```
>>> example = Example(1)
```

So, to create an object, pass fields and their values as named arguments, and to retrieve it, pass its pk as the only argument. To retrieave instances via other fields than the pk, check the Collections section later in this document.

If you don't pass any argument to the *RedisModel*, default one from fields are taken and are saved. But if no arguments and no default values, you get an empty instance, with no filled fields and no pk set.

The pk will be created with the first field. It's important to know that we do not store any concept of "model", each field is totally independent, thought the keys to save them in Redis are based on the object's pk. So you can have 50 fields in a model and save only one of them.

Another really important thing to know is that when you create/retrieve an object, there is absolutely no data stored in it. Each time you access data via a field, the data is fetched from Redis.

2.3.1 Model attributes

When defining a model, you will add fields, but there is also some other attributes that are mandatory or may be useful.

database

The *database* attribute is mandatory and must be a RedisDatabase instance. See Database

namespace

You can't have two models with the same name on the same database. Except if you use namespacing.

Each model has a *namespace*, default to an empty string.

The *namespace* can be used to regroup models. All models about registration could have the *namespace* "registration", ones about the payment could have "payment", and so on.

With this you can have models with the same name in different *namespaces*, because the Redis keys created to store your data is computed with the *namespace*, the model name, and the pk of objects.

abstract

If you have many models sharing some field names, and/or within the same database and/or the same namespace, it could be useful to regroup all common stuff into a "base model", without using it to really store data in Redis.

For this you have the *abstract* attribute, *False* by default:

```
class Content(model.RedisModel):
    database = main_database
    namespace = "content"
```

```
    abstract = True

    title = fields.InstanceHashField()
    pub_date = field.InstanceHashField()

class Article(Content):
    content = fields.StringField()

class Image(Content):
    path = fields.InstanceHashField()
```

In this example, only *Article* and *Image* are real models, both using the *main_database* database, the namespace "content", and having *title* and *pub_date* fields, in addition to their own.

lockable

By default, when updating a *indexable* field, update of the same field for all other instances of the model are locked while the update is not finished, to ensure consistency.

If you prefere speed, or are sure that you don't have more than one thread/process/server that write to the same database, you can set this *lockable* attribute to False to disable it for all the model's fields.

Note that you can also disable it at the field's level.

2.4 Fields

The core module of *limpyd* provides 6 fields types, matching the ones in Redis:

- *StringField*, for the main data type in *Redis*, strings
- *HashField*, for dicts
- *InstanceHashField*, for hashes
- *SetField*, for sets
- *ListField*, for lists
- *SortedSetField*, for sorted sets

You can also manage primary keys with these too fields:

- *PKField*, based on *StringField*
- *AutoPKField*, same as *PKField* but auto-incremented.

All these fields can be indexed, and they manage the keys for you (they take the same arguments as the real Redis ones, as defined in the *StrictRedis* class of redis-py, but without the *key* parameter).

Another thing all fields have in common, is the way to delete them: use the *delete* method on a field, and both the field and its value will be removed from Redis.

2.4.1 Field attributes

When adding fields to a model, you can configure it with some attributes:

default

It's possible to set default values for fields of type *StringField* and *InstanceHashField*:

```
class Example(model.RedisModel):
    database = main_database
    foo = fields.StringField(default='FOO')
    bar = fields.StringField()

>>> example = Example(bar='BAR')
>>> example.foo.get()
'FOO'
```

When setting a default value, the field will be saved when creating the instance. If you defined a *PKField* (not *AutoPKField*), don't forget to pass a value for it when creating the instance, it's needed to store other fields.

indexable

Sometimes getting objects from Redis by its primary key is not what you want. You may want to search for objects with a specific value for a specific field.

By setting the *indexable* argument to True when defining the field, this functionnality is automatically activated, and you'll be able to retrieve objects by filtering on this field using Collections.

To activate it, just set the *indexable* argument to True:

```
class Example(model.RedisModel):
    database = main_database
    foo = fields.StringField(indexable=True)
    bar = fields.StringField()
```

In this example you will be able to filter on the field *foo* but not on *bar*.

See Collections to know how to filter objects.

When updating an indexable field, a lock is acquired on Redis on this field, for all instances of the model. It wasn't possible to use pipeline or redis scripting, because both need to know in advance keys to update, but we don't always know since keys for indexes are based on values. So a all *writing* operations on an indexable field are protected, to ensure consistensy if many threads, process, servers are working on the same redis database.

If you are sure you have only one thread, or you don't want to ensure consistency, you can disable locking by setting to False the *lockable* argument when creating a field, or the *lockable* attribute of a model to inactive the lock for all of its fields.

unique

The *unique* argument is the same as the *indexable* one, except it will ensure that you can't have multiple objects with the same value for some fields. *unique* fields are also indexed, and can be filtered, as for the *indexable* argument.

Example:

```
class Example(model.RedisModel):
    database = main_database
    foo = fields.StringField(indexable=True)
    bar = fields.StringField(unique=True)

>>> example1 = Example(foo='FOO', bar='BAR')
True
>>> example2 = Example(foo='FOO', bar='BAR')
UniquenessError: Key :example:bar:BAR already exists (for instance 1)
```

See Collections to know how to filter objects, as for *indexable*.

lockable

You can set this argument to False if you don't want a lock to be acquired on this field for all instances of the model. See *indexable* for more informations about locking.

If not specified, it's default to True, except if the *lockable* attribute of the model is False, in which case it's forced to False for all fields.

2.4.2 Field types

StringField

StringField based fields allow the storage of strings, but some Redis string commands allow to treat them as integer, float [1] or bits.

Example:

```
from limpyd import model, fields

class Example(model.RedisModel):
    database = main_database

    name = fields.StringField()
```

You can use this model like this:

```
>>> example = Example(name='foo')
>>> example.name.get()
'foo'
>>> example.name.set('bar')
>>> example.name.get()
'bar'
>> example.delete()
```

The *StringField* type support these Redis string commands:

Getters:

- *bitcount*
- *get*
- *getbit*
- *getrange*
- *getset*
- *strlen*

Modifiers:

- *append*
- *decr*
- *getset*
- *incr*
- *incrbyfloat* [1]
- *set*
- *setbit*
- *setnx*

[1] When working with floats, pass them as strings to avoid precision problems.

- *setrange*

HashField

HashField allows storage of a dict in Redis.

Example:

```
class Email(model.RedisModel):
    database = main_database
    headers = fields.HashField()

>>> email = Email()
>>> headers = {'from': 'foo@bar.com', 'to': 'me@world.org'}
>>> email.headers.hmset(**headers)
>>> email.headers.hget('from')
'foo@bar.com'
```

Supported commands:

Getters:

- *hget*
- *hgetall*
- *hmget*
- *hkeys*
- *hvals*
- *hexists*
- *hlen*

Modifiers:

- *hdel*
- *hmset*
- *hsetnx*
- *hset*
- *hincrby*
- *hincrbyfloat* [1]

InstanceHashField

As for *StringField*, *InstanceHashField* based fields allow the storage of strings. But all the *InstanceHashField* fields of an instance are stored in the same Redis hash, the name of the field being the key in the hash.

To fully use the power of Redis hashes, we also provide two methods to get and set multiples field in one operation (see *hmget* and *hmset*). It's usually cheaper to store fields in hash that in strings. And it's faster to set/retrieve them using these two commands.

Example with simple commands:

```
class Example(model.RedisModel):
    database = main_database

    foo = fields.InstanceHashField()
    bar = fields.InstanceHashField()

>>> example.foo.hset('FOO')
1  # 1 because the hash field was created
>>> example.foo.hget()
'FOO'
```

The *InstanceHashField* type support these Redis hash commands:

Getters:

- hget

Modifiers:

- *hincrby*
- *hincrbyfloat* [1]
- *hset*
- *hsetnx*

Deleter:

- Note that to delete the value of a *InstanceHashField*, you can use the *hdel* command, which do the same as the main *delete* one.
- See also *hdel* on the model to delete many *InstanceHashField* at once

Multi:

The following commands are not called on the fields themselves, but on an instance:

- *hmget*
- *hmset*
- *hgetall*
- *hkeys*
- *hvals*
- *hlen*
- *hdel*

hmget

hmget is called directly on an instance, and expects a list of field names to retrieve.

The result will be, as in Redis, a list of all values, in the same order.

If no names are provided, nothing will be fetched. Use *hvals*, or better, *hgetall* to get values for all InstanceHashFields

It's up to you to associate names and values, but you can find an example below:

```
class Example(model.RedisModel):
    database = main_database

    foo = fields.InstanceHashField()
    bar = fields.InstanceHashField()
    baz = fields.InstanceHashField()
    qux = fields.InstanceHashField()

    def hmget_dict(self, *args):
        """
        A call to hmget but which return a dict with field names as keys, instead
        of only a list of values
        """
        values = self.hmget(*args)
        keys = args or self._hashable_fields
        return dict(zip(keys, values))


>>> example = Example(foo='FOO', bar='BAR')
>>> example.hmget('foo', 'bar')
['FOO', 'BAR']
>>> example.hmget_dict('foo', 'bar')
{'bar': 'BAR', 'foo': 'FOO'}
```

hmset

hmset is the reverse of *hmget*, and also called directly on an instance, and expects named arguments with field names as keys, and new values to set as values.

Example (with same model as for *hmget*):

```
>>> example = Example()
>>> example.hmset(foo='FOO', bar='BAR')
True
>>> example.hmget('foo', 'bar')
['FOO', 'BAR']
```

hdel

hdel is called directly on an instance, and expects a list of field names to delete.

The result will be, as in Redis, the number of field really deleted (non-filled ones won't be counted).

```
>>> example = Example()
>>> example.hmset(foo='FOO', bar='BAR', baz='BAZ')
True
>>> example.hmget('foo', 'bar', 'baz')
['FOO', 'BAR', 'BAZ']
>>> example.hdel('foo', 'bar', 'qux')
2
>>> example.hmget('foo', 'bar', 'baz')
[None, None, 'BAZ']
```

Note that you can also call *hdel* on an *InstanceHashField* itself, without parameters, to delete this very field.

```
>>> example.baz.hdel()
1
```

hgetall

hgetall must be called directly on an instance, and will return a dictionary containing names and values of all InstanceHashField with a stored value. If a field has no stored value, it will not appear in the result of *hgetall*.

Example (with same model as for *hmget*):

```
>>> example = Example(foo='FOO', bar='BAR')
>>> example.hgetall()
{'foo': 'FOO', 'bar': 'BAR'}
>>> example.foo.hdel()
>>> example.hgetall()
{bar': 'BAR'}
```

hkeys

hkeys must be called on an instance and will return the name of all the InstanceHashField with a stored value. If a field has no stored value, it will not appear in the result of *hkeys*. Note that the result is not ordered in any way.

Example (with same model as for *hmget*):

```
>>> example = Example(foo='FOO', bar='BAR')
>>> example.hkeys()
['foo', 'bar']
>>> example.foo.hdel()
>>> example.hkeys()
['bar']
```

hvals

hkeys must be called on an instance and will return the value of all the InstanceHashField with a stored value. If a field has no stored value, it will not appear in the result of *hvals*. Note that the result is not ordered in any way.

Example (with same model as for *hmget*):

```
>>> example = Example(foo='FOO', bar='BAR')
>>> example.hvals()
['FOO', 'BAR']
>>> example.foo.hdel()
>>> example.hvals()
['BAR']
```

hlen

hlen must be called on an instance and will return the number of InstanceHashField with a stored value. If a field has no stored value, it will not be count in the result of *hlen*.

Example (with same model as for *hmget*):

```
>>> example = Example(foo='FOO', bar='BAR')
>>> example.hlen()
2
>>> example.foo.hdel()
>>> example.hlen()
1
```

SetField

SetField based fields can store many values in one field, using the set data type of Redis, an unordered set (with unique values).

Example:

```
from limpyd import model, fields

class Example(model.RedisModel):
    database = main_database

    stuff = fields.SetField()
```

You can use this model like this:

```
>>> example = Example()
>>> example.stuff.sadd('foo', 'bar')
2  # number of values really added to the set
>>> example.stuff.smembers()
set(['foo', 'bar'])
>>> example.stuff.sismember('bar')
True
>>> example.stuff.srem('bar')
True
>>> example.stuff.smembers()
set(['foo'])
>>> example.stuff.delete()
True
```

The *SetField* type support these Redis set commands:

Getters:

- *scard*
- *sismember*
- *smembers*
- *srandmember*

Modifiers:

- *sadd*
- *spop*
- *srem*

ListField

ListField based fields can store many values in one field, using the list data type of Redis. Values are ordered, and are not unique (you can push many times the same value).

Example:

```
from limpyd import model, fields

class Example(model.RedisModel):
    database = main_database

    stuff = fields.ListField()
```

You can use this model like this:

```
>>> example = Example()
>>> example.stuff.rpush('foo', 'bar')
2  # number of values added to the list
>>> example.stuff.lrange(0, -1)
['foo', 'bar']
>>> example.stuff.lindex(1)
'bar'
>>> example.stuff.lrem(1, 'bar')
1  # number of values really removed
>>> example.stuff.lrange(0, -1)
['foo']
>>> example.stuff.delete()
True
```

The *ListField* type support these Redis list commands:

Getters:

- *lindex*
- *llen*
- *lrange*

Modifiers:

- *linsert*
- *lpop*
- *lpush*
- *lpushx*
- *lrem*
- *lset*
- *ltrim*
- *rpop*
- *rpush*
- *rpushx*

SortedSetField

SortedSetField based fields can store many values, each scored, in one field using the sorted-set data type of Redis. Values are unique (it's a set), and are ordered by their score.

Example:

```
from limpyd import model, fields

class Example(model.RedisModel):
    database = main_database

    stuff = fields.SortedSetField()
```

You can use this model like this:

```
>>> example = Example()
>>> example.stuff.zadd(foo=2.5, bar=1.1)
2  # number of values added to the sorted set
>>> example.stuff.zrange(0, -1)
['bar', 'foo']
>>> example.stuff.zrangebyscore(1, 2, withscores=True)
[('bar', 1.1)]
>>> example.stuff.zrem('bar')
1  # number of values really removed
>>> example.stuff.zrangebyscore(1, 2, withscores=True)
[]
>>> example.stuff.delete()
True
```

The *SortedSetField* type support these Redis sorted set commands:

Getters:

- *zcard*
- *zcount*
- *zrange*
- *zrangebyscore*
- *zrank*
- *zrevrange*
- *zrevrangebyscore*
- *zrevrank*
- *zscore*

Modifiers:

- *zadd*
- *zincrby*
- *zrem*
- *zremrangebyrank*
- *zremrangebyscore*

PKField

PKField is a special subclass of *StringField* that manage primary keys of models. The PK of an object cannot be updated, as it serves to create keys of all its stored fields. It's this PK that is returned, with others, in Collections.

A PK can contain any sort of string you want: simple integers, float [1], long uuid, names...

If you want a PKField which will be automatically filled, and auto-incremented, see *AutoPKField*. Otherwise, with standard *PKField*, you must assign a value to it when creating an instance.

By default, a model has a *AutoPKField* attached to it, named *pk*. But you can redefine the nameand type of PKField you want.

Examples:

```
class Foo(model.RedisModel):
    """
    The PK field is `pk`, and will be auto-incremented.
    """
    database = main_database

class Bar(model.RedisModel):
    """
    The PK field is `id`, and will be auto-incremented.
    """
    database = main_database
    id = fields.AutoPKField()

class Baz(model.RedisModel):
    """
    The PK field is `name`, and won't be auto-incremented, so you must assign it a value when creati
    """
    database = main_database
    name = fields.PKField()
```

Note that wathever name you use for the *PKField* (or *AutoPKField*), you can always access it via the name *pk* (but also we its real name). It's easier for abstraction:

```
class Example(model.RedisModel):
    database = main_database
    id = fields.AutoPKField()
    name = fields.StringField()

>>> example = Example(name='foobar')
>>> example.pk.get()
1
>>> example.id.get()
1
```

AutoPKField

A *AutoPKField* field is a *PKField* filled with auto-incremented integers, starting to 1. Assigning a value to of *AutoPKField* is forbidden.

It's a *AutoPKField* that is attached by default to every model, if no other one is defined.

See *PKField* for more details.

2.5 Collections

The main and obvious way to get data from Redis via *limpyd* is to know the primary key of objects and instantiate them one by one.

But some fields can be indexed, passing them the *indexable* or *unique* attribute.

If fields are indexed, it's possible to make query to retrieve many of them, using the *collection* method on the models.

The filtering has some limitations:

- you can only filter on fields with *indexable* and/or *unique* attributes set to True
- you can only filter on full values (*limpyd* doesn't provide filters like "startswith", "contains"...)
- all filters are "and"ed
- no "not" (only able to find mathing fields, not to exlude some)
- no "join" (filter on one model only)

The result of a call to the *collection* is lazy. The query is only sent to Redis when data is really needed, to display or do computation with them.

By default, a collection returns a list of primary keys for all the matching objects, but you can sort them, retrieve only a part, and/or directly get full instances instead of primary keys.

We will explain *Filtering*, *Sorting*, *Slicing*, *Instanciating*, and *Lazyness* below, based on this example:

```
class Person(model.RedisModel):
    database = main_database
    firstname = fields.InstanceHashField(indexable=True)
    lastname = fields.InstanceHashField(indexable=True)
    birth_year = fields.InstanceHashField(indexable=True)

    def __repr__(self):
        return "<[%s] %s %s (%s)>" % tuple([self.pk.get()] + self.hmget('firstname', 'lastname', 'bi

>>> Person(firstname='John', lastname='Smith', birth_year=1960)
<[1] John Smith (1960)>
>>> Person(firstname='John', lastname='Doe', birth_year=1965)
<[2] John Doe (1965)>
>>> Person(firstname='Emily', lastname='Smith', birth_year=1950)
<[3] Emily Smith (1950)>
>>> Person(firstname='Susan', lastname='Doe', birth_year=1960)
<[4] Susan Doe (1960)>
```

Note that for each primary key got from redis, a real instance is created, with a check for pk existence. As it can lead to a lot of redis calls (one for each instance), if you are sure that all primary keys really exists (it must be the case if nothing special was done), you can skip these tests by passing the *skip_exist_test* named argument to True when calling *instances*:

```
>>> Person.collection().instances(skip_exist_test=True)
```

Note that when you'll update an instance got with *skip_exist_test* set to True, the existence of the primary key will be done before the update, raising an exception if not found.

To cancel retrieving instances and get the default return format, call the *primary_keys* method:

```
>>> Person.collection(firstname='John').instances().primary_keys()
>>> ['1', '2']
```

2.5.1 Filtering

To filter, simply call the *collection* (class)method with fields you want to filter as keys, and wanted values as values:

```
>>> Person.collection(firstname='John')
['1', '2']
>>> Person.collection(firstname='john', lastname='Smith')
['1']
>>> Person.collection(birth_year=1965)
['2']
>>> Person.collection(birth_year=1965, lastname='Smith')
[]
```

You cannot pass two filters with the same name. All filters are "and"ed.

2.5.2 Slicing

To slice the result, simply act as it's the result of a collection is a list:

```
>>> Person.collection(firstname='John')
['1', '2']
>>> Person.collection(firstname='John')[1:2]
['2']
```

2.5.3 Sorting

With the help of the *sort* command of Redis, *limpyd* is able to sort the result of collections.

It's as simple as calling the *sort* method of the collection. Use the *by* argument to specify on which field to sort.

Redis default sorting is numeric. If you want to sort values lexicographically, set the *alpha* parameter to True.

Example:

```
>>> Person.collection(firstname='John')
['1', '2']
>>> Person.collection(firstname='John').sort(by='lastname', alpha=True)
['2', '1']
>>> Person.collection(firstname='John').sort(by='lastname', alpha=True)[1:2]
[1']
>>> Person.collection().sort(by='birth_year')
['3', '1', '4', '2']
```

2.5.4 Instanciating

If you want to retrieve already instanciated objects, instead of only primary keys and having to do instanciation yourself, you simply have to call *instances()* on the result of the collection. The result of the collection and its methods (*sort* and *instances*) return a collection, so you can do chaining:

```
>>> Person.collection(firstname='John')
['1', '2']
>>> Person.collection(firstname='John').instances()
[<[1] John Smith (1960)>, <[2] John Doe (1965)>]
>>> Person.collection(firstname='John').instances().sort(by='lastname', alpha=True)
[<[2] John Doe (1965)>, <[1] John Smith (1960)>]
>>> Person.collection(firstname='John').sort(by='lastname', alpha=True).instances()
```

```
[<[2] John Doe (1965)>, <[1] John Smith (1960)>]
>>> Person.collection(firstname='John').sort(by='lastname', alpha=True).instances()[0]
[<[2] John Doe (1965)>
```

2.5.5 Lazyness

The result of a collection is lazy. In fact it's the collection itself, it's why we can chain calls to *sort* and *instances*.

The query is sent to Redis only when the data are needed. In the previous examples, data was needed to display them.

But if you do something like:

```
>>> results = Person.collection(firstname='John').instances())
```

nothing will be done while results is not printed, iterated...

2.5.6 Subclassing

The collection stuff is managed by a class named *CollectionManager*, available in *limpyd.collection*.

If you want to use another class (you own subclass or one provided in contrib, see Extended collection), you can do it simple by declaring the *collection_manager* attribute of the model:

```
class MyOwnCollectionManager(CollectionManager):
    pass

class Person(model.RedisModel):
    database = main_database
    collection_manager = MyOwnCollectionManager

    firstname = fields.InstanceHashField(indexable=True)
    lastname = fields.InstanceHashField(indexable=True)
    birth_year = fields.InstanceHashField(indexable=True)
```

You can also do it on each call to the *collection* method, by passing the class to the *manager* argument (useful if you want to keep the default manager in the model):

```
>>> Person.collection(firstname='John', manager=MyOwnCollectionManager)
```

2.6 Contrib

To keep the core of *limpyd*, say, "limpid", we limited what it contains. But we added some extra stuff in the *contrib* module:

- *Related fields*
- *Pipelines*

2.6.1 Related fields

limpyd provide a way to link models, via the *related* contrib module. It's only shortcuts to already existing stuff, aiming to make relations easy.

Start with an example:

```
from limpyd import fields
from limpyd.contrib import related

class Person(related.RelatedModel):
    database = main_database
    name = fields.PKField()  # redefine a PK just for the example

class Group(related.RelatedModel):
    database = main_database
    name = fields.PKField()
    private = fields.StringField()
    owner = related.FKInstanceHashField('Person')
    members = related.M2MSetField('Person', related_name='membership')
```

With this we can do stuff like this:

```
>>> core_devs = Group(name='limpyd core devs', private=0)
>>> ybon = Person(name='ybon')
>>> twidi = Person(name='twidi')
>>> core_devs.owner.hset(ybon)
1
>>> core_devs.members.sadd(twidi, ybon._pk)  # give a limpyd object, or a pk
2
>>> core_devs.members.smembers()
set(['ybon', 'twidi'])
>>> ybon.group_set(private=0)  # it's a collection, the limpyd way !
['limpyd core devs']
>>> twidi.membership()  # it's a collection too
['limpyd core devs']
```

Related model

To use related fields, you must use *related.RelatedModel* instead of *model.RedisModel*. It handles creation of *related collections* and manage propagation of deletion for us.

Related field types

The *related* module provides 5 field types, based on the standard ones. All have the *indexable* attribute set to True.

There is one big addition on these fields over the normal one. Everywhere you can pass a value to store (in theory you would pass an object's primary key), you can pass an instance of a limpyd model. The primary key of these instances will be extraced for you.

Here are the new field types:

FKStringField

The *FKStringField* type is based on *StringField* and allow setting a foreign key.

It just stores the primary key of the related object in a *StringField*.

FKInstanceHashaField

The *FKInstanceHashaField* type is based on *InstanceHashField* and allow setting a foreign key.

It works like *FKStringField* but, as a *InstanceHashField*, can be retrieved with other fields via the *hmget* method on the instance.

M2MSetField

The *M2MSetField* type is based on *SetField* and allow setting many foreign keys, acting as a Many 2 Many fields.

If no order is needed, it's the best choice for M2M, because it's the lightest M2M field (memory occupation), and it's fast to check if an element is included (*sismember*, O(1)), or to remove one (*srem*, O(N) where N is the number of members to be removed.).

If you need ordering *and* unicity, check *M2MSortedSetField*.

M2MListField

The *M2MListField* type is based on *ListField* and allow setting many foreign keys, acting as a Many 2 Many fields.

It works like *M2MSetField*, with two differences, because it's a list and not a set:

- the list of foreign keys is ordered
- we can have many times the same foreign key

This type is usefull to keep the order of the foreign keys, but as it does not ensure unicity, the use cases are less obvious.

If you need ordering *and* unicity, check *M2MSortedSetField*.

M2MSortedSetField

The *M2MSortedSetField* type is based on *SortedSetField* and allow setting many foreign keys, acting as a Many 2 Many fields.

It works like *M2MSetField*, with one differences, because it's a sorted set and not a simple set: each foreign key has a score attached to it, and the list for foreign keys is sorted by this score.

This score is usefull to keep the entries unique AND sorted. It can be a date (as a timestamp because the score must be numeric), allowing, in our example (Person/Group), to keep list of members in the order they joined the group.

Related field arguments

The related fields accept two new arguments when declaring them. One to tell to which model it's related (*to*), and one to give a name to the *related collection*

to

The first new argument (and the first in the list of accepted ones, useful to pass it without naming it), is *to*, the name of the model on which this field is related to.

Note that the related model must be on the same Database.

It can accept a *RelatedModel*:

```
class Person(related.RelatedModel):
    database = main_database
    name = StringField()

class Group(related.RelatedModel):
    database = main_database
    name = StringField()
    owner = FKStringField(Person)
```

In this case the *Related model* must be defined before the current model.

And it can accept a string. There is two ways to define model with a string:

- the name of a *RelatedModel*:

```
class Group(related.RelatedModel):
    database = main_database
    owner = FKStringField('Person')
```

If you want to link to a model with a different namespace than the one for the current model, you can add it:

```
class Group(related.RelatedModel):
    database = main_database
    owner = FKStringField('my_namespace:Person')
```

- 'self', to define a link to the same model on which the related field is defined:

```
class Group(related.RelatedModel):
    database = main_database
    parent = FKStringField('self')
```

related_name

The *related_name* argument is not mandatory, except in some cases described below.

This argument is the name which will be used to create the *Related collection* on the related model (the on described by the *to* argument)

If defined, it must be a string. This string can accept to formatable arguments: *%(namespace)s* and *%(model)s* which will be replaced by the namespace and name of the model on which the related field is defined. It's usefull for subclassing:

```
class Person(related.RelatedModel):
    database = main_database
    name = StringField()

class BaseGroup(related.RelatedModel):
    database = main_database
    namespace = 'groups'
    abstract = True

    name = StringField()
    owner = FKStringField('Person', related_name='%(namespace)s_%(model)s_set')

class PublicGroup(BaseGroup):
    pass

class PrivateGroup(BaseGroup):
    pass
```

In this example, a person will have two related collections:

- *groups_publicgroup_set*, liked to the *parent* field of *PublicGroup*
- *groups_privategroup_set*, liked to the *parent* field of *PrivateGroup*

Note that, exept for namespace that will be automatically converted if needed, related names should be valid python identifiers.

Related collection

Related collections are the other side of the relation. They are created on the related model, based on the *related_name* argument used when creating the related field.

They are a shortcut to the real collection, but available to ease writing.

Let's define some models:

```
class Person(related.RelatedModel):
    database = main_database
    name = PKStringField()

class Group(related.RelatedModel):
    database = main_database
    name = PKStringField()
    private = fields.StringField(defaut=0)
    owner = FKStringField('Person', related_name='owned_groups')
```

Now we can do:

```
>>> group1 = Group(name='group 1')
>>> group2 = Group(name='group 1', private=1)
>>> person1 = Person(name='person 1')
>>> group1.owner.set(person1)
>>> group2.owner.set(person1)
```

To retrieve groups owned by *person1*, we can use the standard way:

```
>>> Group.collection(owner=person1.pk.get())
['group 1', 'group 2']
```

or, with the related collection:

```
>>> person1.owned_groups()
['group 1', 'group 2']
```

These two lines return exactly the same thing, a lazy collection (See Collections).

You can pass other filters too:

```
>>> person1.owned_groups(private=1)
['group 2']
```

Note that the collection manager of all related fields is the *ExtendedCollectionManager*, so you can do things like:

```
>>> owned = person1.owned_groups()
>>> owned.filter(private=1)
['group 2']
```

Retrieving the other side

Foreign keys

It's easy to set a foreign key, and easy to retrieve it using the default API.

Using these models:

```
class Person(related.RelatedModel):
    database = main_database
    name = StringField()

class Group(related.RelatedModel):
    database = main_database
    name = StringField()
    owner = FKStringField(Person)
```

We can add data:

```
>>> core_devs = Group(name='limpyd core devs', private=0)
>>> ybon = Person(name='ybon')
>>> core_devs.owner.hset(ybon)
```

And we can retrieve the related object this way:

```
>>> owner_pk = core_devs.owner.hget()
>>> owner = Person(owner_pk)
```

But we can use the *instance* method defined on foreign keys:

```
>>> owner = core_devs.owner.instance()
```

Many to Many

To provide consistency on calling collections on the both sides of a relation, the *M2MSetField*, *M2MListField* and *M2MSortedSetField* are *callable*, simulating a call to a collection, and effectively returning one. It's very useful to sort and/or return *instances*, *values* or *values_list*.

Imagine the model:

```
class Person(related.RelatedModel):
    database = main_database
    name = PKStringField()
    following = M2MSetField('self', related_name='followers')
```

Let's add some data:

```
>>> foo = Person(name='Foo')  # pk=1
>>> bar = Person(name='Bar')  # pk=2
>>> baz = Person(name='Baz')  # pk=3
>>> foo.following.sadd(bar, baz)
>>> baz.following.sadd(bar)
```

So we can retrieve followers via the *Related collection*:

```
>>> bar.followers()
['1', '3']
>>> baz.followers().values_list('name', flat=True)
['foo', 'baz']
```

And on the other side... without simulating a collection when calling a M2MField, it's easy to retrieve primary keys:

```
>>>foo.following.smembers()
['2', '3']
```

But it's not the same "api" (but it sounds ok because it's a SetField), and it's really hard to retrieve names, or other stuff like with *values* and *values_list*, or even *instances*.

With the callable possibility added to M2M fields, you can do this:

```
>>> foo.following()  # return a collection
['1', '3']
>>> foo.following().values_list('name', flat=True)
['bar', 'baz']
```

Note that to provide even more consitency, use can call the *collection* method of a M2M field instead of simple "calling" it. So both lines below are the same:

```
>>> foo.following()
>>> foo.following.collection()
```

Update and deletion

One of the main advantage of using related fields instead of doing it yourself, is that updates and deletions are handled as you would, transparently.

In the previous example, if the owner of a group is updated (or deleted), the previous owner doesn't have this group in its owned_group collections.

The same applies on the other side. If a person who is the owner of a group is deleted, the value of the groups'owner field is deleted too.

And it works with M2M fields too.

2.6.2 Pipelines

In the contrib module, we provide a way to wirk with pipelines as defined in redis-py, providimg abstraction to let the fields connect to the pipeline, not the real Redis connection (this won't be the case if you use the default pipeline in redis-py)

To activate this, you have to import and to use *PipelineDatabase* instead of the default *RedisDatabase*, without touching the arguments.

Instead of doing this:

```
from limpyd.database import RedisDatabase

main_database = RedisDatabase(
    host="localhost",
    port=6379,
    db=0
)
```

Just do:

```
from limpyd.contrib.database import PipelineDatabase

main_database = PipelineDatabase(
    host="localhost",
```

```
    port=6379,
    db=0
)
```

This *PipelineDatabase* class adds two methods: *pipeline* and *transaction*

pipeline

The pipeline provides the same functionnalities as for the default pipeline in redis-py, but it handles transparently the use of the pipeline instead of the default collection for all fields operation.

But be aware that within a pipeline you cannot get values from fields to do something with them. It's because in a pipeline, all commands are sent in bulk, and all results are retrieved in bulk too (one for each command), when exiting the pipeline.

It does not mean that you cannot set many fields in one time in a pipeline, but you must have values not depending of other fields, and, also very important, you cannot update indexable fields ! (so no related fields either, because they are all indexable)

The best use for pipelines in *limpyd*, is to get a lot of values in one pass.

Say we have this model:

```
from limpyd.contrib.database import PipelineDatabase

main_database = PipelineDatabase(
    host="localhost",
    port=6379,
    db=0
)

class Person(model.RedisModel):
    database = main_database
    namespace='foo'
    name = fields.StringField()
    city = fields.StringField(indexable=True)
```

Add some data:

```
Person(name='Jean Dupond', city='Paris')
Person(name='Francois Martin', city='Paris')
Person(name='John Smith', city='New York')
Person(name='John Doe', city='San Franciso')
Person(name='Paul Durand', city='Paris')
```

Say we have already a lot of Person saved, we can retrieve all names this way:

```
persons = list(Person.collection(city='Paris').instances())
with main_database.pipeline() as pipeline:
    for person in persons:
        person.name.get()
    names = pipeline.execute()
print names
```

This will result in only one call (within the pipeline):

```
>>> ['Jean Dupond', 'Francois Martin', 'Paul Durand']
```

All in one only call to the Redis server.

Note that in pipelines you can you the *watch* command, but it's easier to use the *transaction* method described below.

transaction

The *transaction* method available on the *PipelineDatabase* object, is the same as the one in redis-py, but using its own *pipeline* method.

The goal is to help using pipelines with watches.

The *watch* mechanism in Redis allow us to read values and use them in a pipeline, being sure that the values got in the first step were not updated by someone else since we read them.

Imagine the *incr* method doesn't exists. Here is a way to implement it with a transaction without race condition (ie without the risk of having our value updated by someone else between the moment we read it, and the moment we save it):

```
class Page(model.RedisModel):
    database = main_database  # a PipelineDatabase object
    url = fields.StringField(indexable=True)
    hits = fields.StringField()

    def incr_hits(self):
        """
        Increment the number of hits without race condition
        """

        def do_incr(pipeline):

            # transaction not started, we can read values
            previous_value = self.hits.get()

            # start the transaction (MANDATORY CALL)
            pipeline.multi()

            # set the new value
            self.hits.set(previous_value+1)


        # run `do_incr` in a transaction, watching for the hits field
        self.database.transaction(do_incr, *[self.hits])
```

In this example, the *do_incr* method will be aborted and executed again, restarting the transaction, each time the *hits* field of the object is updated elsewhere. So we are absolutely sure that we don't have any race conditions.

The argument of the *transaction* method are:

- **func**, the function to run, encaspulated in a transaction. It must accept a *pipeline* argument.
- ***watches**, a list of keys to watch (if a watched key is updated, the transaction is restarted and the function aborted and executed again). Note that you can pass keys as string, or fields of limpyd model instances (so their keys will be retrieved for you).

The *transaction* method returns the value returned by the execution of its internal pipeline. In our example, it will return *[True]*.

Note that as for the *pipeline* method, you cannot update indexables fields in the transaction because read commands are used to update them.

2.6.3 Extended collection

Although the standard collection may be sufficient in most cases, we added an *ExtendedCollectionManager* in contrib, which enhance the base one with some useful stuff:

- ability to retrieve values as dict or liist of tuples
- ability to chain filters
- ability to intersect the final result with a list of primary keys
- ability to sort by the score of a sorted set
- ability to pass fields on some methods
- ability to store results

To use this *ExtendedCollectionManager*, declare it as seen in *Subclassing*.

All of these new capabilities are described below:

2.6.4 Retrieving values

If you don't want only primary keys, but instances are too much, or too slow, you can ask the collection to return values with two methods: *values* and *values_list* (inspired by django)

It can be really useful to quickly iterate on all results when you, for example, only need to display simple values.

values

When calling *values* on a collection, the result of the collection is not a list of primary keys, but a list of dictionaries, one for each matching entry, with each field passed as argument. If no field is passed, all fields are retrieved. Note that only simple fields (*PKField*, *StringField* and *InstanceHashField*) are concerned.

Example:

```
>>> Person.collection(firstname='John').values()
[{'pk': '1', 'firstname': 'John', 'lastname': 'Smith', 'birth_year': '1960'}, {'pk': '2', 'firstname
>>> Person.collection(firstname='John').values('pk', 'lastname')
[{'pk': '1', 'lastname': 'Smith'}, {'pk': '2', 'lastname': 'Doe'}]
```

values_list

The *values_list* method works the same as *values* but instead of having the collection return a list of dictionaries, it will return a list of tuples with values for asked fields, in the same order as they are passed as arguments. If no field is passed, all fields are retrieved in the same order as they are defined in the model.

Example:

```
>>> Person.collection(firstname='John').values_list()
[('1', 'John', 'Smith', '1960'), (2', 'John', 'Doe', '1965')]
>>> Person.collection(firstname='John').values_list('pk', 'lastname')
[('1', 'Smith'), ('2', 'Doe')]
```

If you want to retrieve a single field, you can ask to get a flat list as a final result, by passing the *flat* named argument to True:

```
>>> Person.collection(firstname='John').values_list('pk', 'lastname')  # without flat
[('Smith', ), ('Doe', )]
>>> Person.collection(firstname='John').values_list('lastname', flat=True)  # with flat
['Smith', 'Doe']
```

To cancel retrieving values and get the default return format, call the *primary_keys* method:

```
>>> Person.collection(firstname='John').values().primary_keys()  # works with values_list too
>>> ['1', '2']
```

Chaining filters

With the standard collection, you can chain method class but you cannot add more filters than the ones defined in the *collecion* method. The only way was to create a dictionary, populate it, then pass it as named arguments:

```
>>> filters = {'firstname': 'John'}
>>> if want_to_filter_by_city:
>>>     filters['city'] = 'New York'
>>> collection = Person.collection(**filters)
```

With the *ExtendedCollectionManager* available in *contrib.collection*, you can add filters after the initial call:

```
>>> collection = Person.collection(firstname='John')
>>> if want_to_filter_by_city:
>>>     collection.filter(city='New York')
```

filter return the collection object itself, so it can be chained.

Note that all filters are ANDed, so if you pass two filters on the same field, you may have an empty result.

Intersections

Say you already have a list of primary keys, maybe got from a previous filter, and you want to get a collection with some filters but matching this list. With *ExtendedCollectionManager*, you can easily do this with the *intersect* method.

This *intersect* method takes a list of primary keys and will intersect, if possible at the Redis level, the result with this list.

intersect return the collection itself, so it can be chained, as all methods of a collection. You may call this method many times to intersect many lists, but you can also pass many lists in one *intersect* call.

Here is an example:

```
>>> my_friends = [1, 2, 3]
>>> john_people = list(Person.collection(firstname='John'))
>>> my_john_friends_in_newyork = Person.collection(city='New York').intersect(john_people, my_friends
```

intersect is powerful as it can handle a lot of data types:

- a python list
- a python set
- a python tuple
- a string, which must be the key of a Redis set (cannot be a list of sorted set for now)
- a *limpyd* *SetField*, attached to a model
- a *limpyd* *ListField*, attached to a model
- a *limpyd* *SortedSetField*, attached to a model

Imagine you have a list of friends in a *SetField*, you can directly use it to intersect:

```
>>> # current_user is an instance of a model, and friends a SetField_
>>> Person.collection(city='New York').intersect(current_user.friends)
```

Sort by score

Sorted sets in Redis are a powerful feature, as it can store a list of data sorted by a score. Unfortunately, we can't use this score to sort via the Redis *sort* command, which is used in *limpyd* to sort collections.

With *ExtendedCollectionManager*, you can do this using the *sort* method, but with the new *by_score* named argument, instead of the *by* one used in simple sort.

The *by_score* argument accepts a string which must be the key of a Redis sorted set, or a *SortedSetField* (attached to an instance)

Say you have a list of friends in a sorted set, with the date you met them as a score. And you want to find ones that are in you city, but keep them sorted by the date you met them, ie the score of the sorted set. You can do this this way:

```
# current_user is an instance of a model, with city a field holding a city name
# and friends, a sorted_set with Person's primary keys as value, and the date
# the current_user met them as score.

>>> # start by filtering by city
>>> collection = Person.collection(city=current_user.city.get())
>>> # then intersect with friends
>>> collection.intersect(current_user.friends)
>>> # finally keep sorting by friends meet date
>>> collection.sort(by_score=current_user.friends)
```

With the sort by score, as you have to use the *sort* method, you can still use the *alpha* and *desc* arguments (see *Sorting*)

When using *values* or *values_list* (see *Retrieving values*), you may want to retrieve the score between other fields. To do so, simply use the SORTED_SCORE constant (defined in *contrib.collection*) as a field name to pass to *values* or *values_list*:

```
>>> from limpyd.contrib.collection import SORTED_SCORE
>>> # (following previous example)
>>> collection.sort(by_score=current_user.friends).values('name', SORTED_SCORE)
[{'name': 'John Smith', 'sorted_score': '1985.0'}]  # here 1985.0 is the score
```

Passing fields

In the standard collection, you must never pass fields, only names and values, depending on the methods. In the *contrib* module, we already allow passing fields in some place, as to set FK and M2M in *Related fields*.

Now you can do this also in collection (if you use *ExtendedCollectionManager*):

- the *by* argument of the *sort* method can be a field, and not only a field name
- the *by_score* arguement of the *sort* method can be a *SortedSetField* (attached to an instance), not only the key of a Redis sorted set
- arguments of the *intersect* method can be python list(etc...) but also multi-values *RedisField*
- the right part of filters (passed when calling *collection* or *filter*) can also be a *RedisField*, not only a value. If a *RedisField* (specifically a *SingleValueField*), its value will be fetched from Redis only when the collection will be really called

Storing

For collections with heavy computations, like multiple filters, intersecting with list, sorting by sorted set, it can be useful to store the results.

It's possible with *ExtendedCollectionManager*, simply by calling the *store* method, which take two optional arguments:

- *key*, which is the key where the result will be stored, default to a randomly generated one
- *ttl*, the duration, in seconds, for which we want to keep the stored result in Redis, default to *DEFAULT_STORE_TTL* (60 seconds, defined in *contrib.collection*). You can pass None if you don't want the key to expire in Redis.

When calling *store*, the collection is executed and you got a new *ExtendedCollectionManager* object, pre-filled with the result of the original collection.

Note that only primary keys are stored, even if you called *instances*, *values* or *values_list*. But arguments for these methods are set in the new collection so if you call it, you'll get what you want (instances, dictionaries or tuples). You can call *primary_keys* to reset this.

If you need the key where the data are stored, you can get it by calling the *stored_key* method on the new collection. With it, you can later create a collection based on this key.

One important thing to note: the new collection is based on a Redis list. As you can add filters, or intersections, like any collection, remember that by doing this, the list will be converted into a set, which can take time. It's preferable to do this on the original collection before sorting (but it's possible and you can always store the new filtered collection into an other one.)

A last word: if the key is already expired when you execute the new collection, a *DoesNotExist* exception will be raised.

An example to show all of this, based on the previous example (see *Sort by score*):

```
>>> # Start by making a collection with heavy calculation
>>> collection = Person.collection(city=current_user.city.get())
>>> collection.intersect(current_user.friends)
>>> collection.sort(by_score=current_user.friends)

>>> # then store the result
>>> stored_collection = collection.store(ttl=3600)  # keep the result for one hour
>>> # get, say, pk and names
>>> page_1 = stored_collection.values('pk', 'name')[0:10]

>>> # get the stored key
>>> stored_key = stored_collection.stored_key

>>> # later (less than an hour), in another process (passing the stored_key between the processes is
>>> stored_collection = Person.collection().from_stored(stored_key)
>>> page_2 = stored_collection.values('pk', 'name')[10:20]

>>> # want to extend the expire time of the key ?
>>> my_database.connection.expire(store_key, 36000)  # 10 hours
>>> # or remove this expire time ?
>>> my_database.connection.persist(store_key)
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search