redis-limpyd Documentation

Release 1.3.2

Yohan Boniface

May 03, 2020

Contents

1 Show me some code! 3

2 Contents 5

2.1 About . . . 5
2.1.1 Source . . . 5
2.1.2 Install . . . 5
2.1.3 Participation . . . 5
2.1.4 Maintainers . . . 5
2.1.5 Extensions . . . 6
2.2 Database . . . 6
2.2.1 General . . . 6
2.2.2 Switch database . . . 6
2.2.3 Tools . . . 7
2.3 Models . . . 7
2.3.1 Model attributes . . . 8
2.3.1.1 database . . . 8
2.3.1.2 namespace . . . 8
2.3.1.3 abstract . . . 9
2.3.1.4 lockable . . . 9
2.3.2 Model class methods . . . 9
2.3.2.1 get . . . 9
2.3.2.2 get_or_connect . . . 9
2.3.2.3 exists . . . 10
2.3.2.4 lazy_connect . . . 10
2.3.2.5 scan_model_keys . . . 10
2.3.3 Model instance methods . . . 10
2.3.3.1 delete . . . 10
2.3.3.2 scan_keys . . . 10
2.4 Fields . . . 10
2.4.1 Field attributes . . . 11
2.4.1.1 default . . . 11
2.4.1.2 indexable . . . 11
2.4.1.3 unique . . . 12
2.4.1.4 indexes . . . 12
2.4.1.5 lockable . . . 12
2.4.2 Field types . . . 12

2.4.2.1 StringField . . . 12
2.4.2.1.1 Getters . . . 13
2.4.2.1.2 Modifiers . . . 13
2.4.2.2 HashField . . . 13
2.4.2.2.1 Getters . . . 14
2.4.2.2.2 Modifiers . . . 14
2.4.2.3 InstanceHashField . . . 14
2.4.2.3.1 Getters . . . 15
2.4.2.3.2 Modifiers . . . 15
2.4.2.3.3 Deleter . . . 15
2.4.2.3.4 Multi . . . 15
2.4.2.3.4.1 hmget . . . 16
2.4.2.3.4.2 hmset . . . 16
2.4.2.3.4.3 hdel . . . 16
2.4.2.3.4.4 hgetall . . . 17
2.4.2.3.4.5 hkeys . . . 17
2.4.2.3.4.6 hvals . . . 17
2.4.2.3.4.7 hlen . . . 18
2.4.2.4 SetField . . . 18
2.4.2.4.1 Getters . . . 19
2.4.2.4.2 Modifiers . . . 19
2.4.2.5 ListField . . . 19
2.4.2.5.1 Getters . . . 19
2.4.2.5.2 Modifiers . . . 20
2.4.2.6 SortedSetField . . . 20
2.4.2.6.1 Getters . . . 21
2.4.2.6.2 Modifiers . . . 21
2.4.2.7 PKField . . . 21
2.4.2.8 AutoPKField . . . 22
2.5 Collections . . . 22
2.5.1 Filtering . . . 23
2.5.2 Slicing . . . 23
2.5.3 Sorting . . . 24
2.5.4 Instantiating . . . 24
2.5.5 Indexing . . . 25
2.5.5.1 Configuration . . . 26
2.5.5.2 Clean and rebuild . . . 27
2.5.6 Laziness . . . 28
2.5.7 Subclassing . . . 28
2.6 Related fields . . . 29
2.6.1 Related model . . . 29
2.6.2 Related field types . . . 29
2.6.2.1 FKStringField . . . 30
2.6.2.2 FKInstanceHashField . . . 30
2.6.2.3 M2MSetField . . . 30
2.6.2.4 M2MListField . . . 30
2.6.2.5 M2MSortedSetField . . . 30
2.6.3 Related field arguments . . . 30
2.6.3.1 to . . . 31
2.6.3.2 related_name . . . 31
2.6.4 Related collection . . . 32
2.6.5 Retrieving the other side . . . 33
2.6.5.1 Foreign keys . . . 33
2.6.5.2 Many to Many . . . 33

2.6.6 Update and deletion . . . 34
2.7 Pipelines . . . 34
2.7.1 pipeline . . . 35
2.7.2 transaction . . . 36
2.7.3 Pipelines and threads . . . 37
2.8 Extended collection . . . 37
2.8.1 Retrieving values . . . 37
2.8.1.1 values . . . 38
2.8.1.2 values_list . . . 38
2.8.2 Chaining filters . . . 38
2.8.3 Intersections . . . 39
2.8.4 Sort by score . . . 39
2.8.5 Passing fields . . . 40
2.8.6 Storing . . . 40
2.9 Multi-indexes . . . 41
2.9.1 Usage . . . 41
2.9.1.1 name . . . 42
2.9.1.2 key . . . 42
2.9.1.3 transform . . . 42
2.9.2 DateTimeIndex . . . 42
2.9.2.1 Goal . . . 42
2.9.2.2 Date and time parts . . . 42
2.9.2.3 Range indexes . . . 43
2.9.2.4 Full indexes . . . 43
2.10 Changelog . . . 44
2.10.1 Release *v1.3.2* - 2020-05-03 . . . 44
2.10.2 Release *v1.3.1* - 2019-10-11 . . . 44
2.10.3 Release *v1.3* - 2019-09-22 . . . 45
2.10.4 Release *v1.2* - 2018-01-31 . . . 45
2.10.5 Release *v1.1* - 2018-01-30 . . . 45
2.10.6 Release *v1.0.1* - 2018-01-30 . . . 45
2.10.7 Release *v1.0* - 2018-01-29 . . . 45
2.10.8 Release *v0.2.4* - 2015-12-16 . . . 45
2.10.9 Release *v0.2.3* - 2015-12-16 . . . 45
2.10.10 Release *v0.2.2* - 2015-06-12 . . . 45
2.10.11 Release *v0.2.1* - 2015-01-12 . . . 46
2.10.12 Release *v0.2.0* - 2014-09-07 . . . 46
2.10.13 Release *v0.1.3* - 2013-09-07 . . . 46
2.10.14 Release *v0.1.2* - 2013-08-30 . . . 46
2.10.15 Release *v0.1.1* - 2013-08-26 . . . 46
2.10.16 Release *v0.1.0* - 2013-02-12 . . . 46

`Limpyd` provides an **easy** way to store objects in Redis, **without losing the power and the control of the Redis API**, in a *limpid* way, with 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
- Powerful indexing and filtering
- Keep the power of all the Redis data types in your own code

Source code: https://github.com/limpyd/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']
```

(continues on next page)

(continued from previous page)

```
>>> 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, with the help of Stéphane «Twidi» Angel

2.1.1 Source

The project is hosted on Github at https://github.com/limpyd/redis-limpyd

2.1.2 Install

Python versions `2.7` and `3.5` to `3.8` are supported (CPython and PyPy)

Redis-py versions `>= 2.9.1` and `< 2.11` are supported.

```
pip install redis-limpyd
```

2.1.3 Participation

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.

2.1.4 Maintainers

- Stéphane «Twidi» Angel (main maintainer)
- Yohan Boniface (creator)

2.1.5 Extensions

- A bundle of great extensions: Limpyd-extensions
- A queue/task/job manager: Limpyd-jobs

2.2 Database

2.2.1 General

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 collision problems), but we provide a namespace attribute on models to solve this issue (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 collision 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.

2.2.2 Switch database

Sometimes you may want to change the database used after the models are 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()
```

(continues on next page)

(continued from previous page)

```
    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)
```

2.2.3 Tools

We provide one (for now) method on a database object: `scan_keys`.

It allows to call the SCAN command from Redis for the whole redis database currently used. It will use the same argument as the SCAN command and return a generator of all the keys or the ones matching a pattern:

```
generator = main_database.scan_keys()
while True:
    try:
        do_something_with_key(next(generator))
    except StopIteration:
        break

# ... or ...

generator = main_database.scan_keys(match='something', count=100)  # count is a hint␣
↪for redis for each SCAN call, it's not the max returned

# ... of course it can be casted as a set (or a list, but the returned keys are not␣
↪guaranteed to be unique)

keys = set(main_database.scan_keys(match='something'))
```

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 retrieve instances via other fields than the pk, check the *Collections* section in this documentation.

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, though 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.

2.3.1.1 database

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

2.3.1.2 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.

2.3.1.3 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.

2.3.1.4 lockable

By default, when updating am `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 prefer 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.

2.3.2 Model class methods

2.3.2.1 get

Return an instance of the model given a pk, or some fields to filter on. See the *Collections* section in this documentation.

It will raises a `DoesNotExist` exception if no instance was found with the given arguments, and `ValueError` if more than one instance is found.

```
article = Article.get(12)
article = Article.get(pk=12)
article = Article.get(title='foo', content='bar')
```

2.3.2.2 get_or_connect

Try to get an instance from the database, or create it if it does not exists. Uses the same arguments as `get`.

```
article = Article.get_or_connect(title='foo')
same_article = Article.get_or_connect(title='foo')
```

2.3.2.3 exists

Check if an instance with the given pk or filters exists in the database. Uses the same arguments as `get`.

```
if not Article.exists(title='foo'):
    article = Article(title='foo', content='bar')
```

2.3.2.4 lazy_connect

This is an advanced feature. It takes a PK and create an object with this PK without checking for its existence in the database until an operation is done with the instance.

```
existing = Article.lazy_connect(10)
existing.title.get()  # connects only now to the database

non_existing = Article.lazy_connect(11)
non_existing.title.get()  # will raise ``DoesNotExist``
```

2.3.2.5 scan_model_keys

Also an advanced/debug feature, allows to retrieve (as a generator, or can be casted to a set, for example) all the keys related to this model: collection, max pk used, indexes and all instances fields.

```
print('Keys used for model Article:')
for key in Article.scan_model_keys():
    print(' - ' + key)
```

2.3.3 Model instance methods

2.3.3.1 delete

Will delete the instance and remove its content from the indexes if any.

```
article = Article(title='foo')
article.delete()
```

2.3.3.2 scan_keys

Also an advanced/debug feature, allows to retrieve (as a generator, or can be casted to a set, for example) all the keys related to this instance (ie keys holding all defined fields or the ones with a default values):

```
print('Keys used for Article #%s:' % article.pk.get())
for key in article.scan_keys():
    print(' - ' + key)
```

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:

2.4.1.1 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.

2.4.1.2 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 feature 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`.

When updating an indexable field, a lock is acquired on Redis on this field, for all instances of the model. It isn't possible for this to use pipeline or redis scripting, because both need to know in advance the keys to update, but we don't always know since keys for indexes may be based on values. So all *writing* operations on an indexable field are protected, to ensure consistency 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.

2.4.1.3 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`.

2.4.1.4 indexes

This allow to change the default index used, or use many of them. See the "Indexing" section in *Collections* to know more.

2.4.1.5 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 information 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

2.4.2.1 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:

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

```
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.name.delete()
True
```

The *StringField* type support these Redis string commands:

2.4.2.1.1 Getters

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

2.4.2.1.2 Modifiers

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

2.4.2.2 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'
```

The *HashField* type support these Redis hash commands:

2.4.2.2.1 Getters

- `hget`
- `hgetall`
- `hmget`
- `hkeys`
- `hvals`
- `hexists`
- `hlen`
- `hscan` (returns a generator with all/matching key/value pairs, you don't have to manage the cursor)

2.4.2.2.2 Modifiers

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

2.4.2.3 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:

2.4.2.3.1 Getters

- `hget`

2.4.2.3.2 Modifiers

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

2.4.2.3.3 Deleter

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

2.4.2.3.4 Multi

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

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

2.4.2.3.4.1 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'}
```

2.4.2.3.4.2 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']
```

2.4.2.3.4.3 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 (ie fields without any stored value won't be taken into account).

```
>>> 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
```

2.4.2.3.4.4 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'}
```

2.4.2.3.4.5 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']
```

2.4.2.3.4.6 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']
```

2.4.2.3.4.7 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
```

2.4.2.4 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:

2.4.2.4.1 Getters

- `scard`
- `sismember`
- `smembers`
- `srandmember`
- `sscan` (returns a generator with all/matching values, you don't have to manage the cursor)
- `sort` (with arguments like in redis-py, see redis-py-sort)

2.4.2.4.2 Modifiers

- `sadd`
- `spop`
- `srem`

2.4.2.5 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:

2.4.2.5.1 Getters

- `lindex`

- `llen`
- `lrange`
- `sort` (with arguments like in redis-py, see redis-py-sort)

2.4.2.5.2 Modifiers

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

2.4.2.6 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:

2.4.2.6.1 Getters

- `zcard`
- `zcount`
- `zrange`
- `zrangebyscore`
- `zrank`
- `zrevrange`
- `zrevrangebyscore`
- `zrevrank`
- `zscore`
- `zscan` (returns a generator with all/matching key/score pairs, you don't have to manage the cursor)
- `sort` (with arguments like in redis-py, see redis-py-sort)

2.4.2.6.2 Modifiers

- `zadd`
- `zincrby`
- `zrem`
- `zremrangebyrank`
- `zremrangebyscore`

2.4.2.7 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 name and 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.
    """
```

(continues on next page)

(continued from previous page)

```
    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 creating an instance.
    """
    database = main_database
    name = fields.PKField()
```

Note that whatever 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
```

2.4.2.8 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 *PKField* 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 queries 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`
- the filtering capabilities are limited and must be thought at the beginning
- all filters are `and`-ed
- no `not` (only able to find matching fields, not to exclude 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*, *Instantiating*, *Indexing*, and *Laziness* below, based on this example:

```
class Person(model.RedisModel):
    database = main_database
    firstname = fields.InstanceHashField(indexable=True)
    lastname = fields.InstanceHashField(indexable=True)
    nickname = fields.InstanceHashField(indexable=True, indexes=[TextRangeIndex])
    birth_year = fields.InstanceHashField(indexable=True, indexes=[NumberRangeIndex])

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

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

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.

To return the only one existing element, use `get` instead of `collection` and an instance will be returned. But it will raises a `DoesNotExist` exception if no instance was found with the given arguments, and `ValueError` if more than one instance is found.

In *Indexing* you'll see more filtering capabilities.

2.5.2 Slicing

To slice the result, simply act as if the result of a collection is a list:

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

(continues on next page)

(continued from previous page)

```
>>> 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 sort 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']
```

Note: using `by='pk'` (or the real name of the `pk` field) is the same as not using `by`: it will sort by primary keys, using a numeric filter (use `alpha=True` if your `pk` is not numeric)

2.5.4 Instantiating

If you want to retrieve already instantiated objects, instead of only primary keys and having to do instantiation 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 chain calls:

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

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']
```

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

2.5.5 Indexing

By default, all fields with `indexable=True` use the default index, `EqualIndex`.

It only allows equality filtering (the only legacy index type supported by `limpyd`), but it is fast.

To filter using this index, you simply pass the field and a value in the collection call:

```
>>> Person.collection(firstname='John').instances()
[<[1] John "Joe" Smith (1960)>, <[2] John "Jon" Doe (1965)>]
```

But you can also be more specific about the fact that you want an equality by using the `__eq` suffix. All other indexes use different suffixes.

This design is inspired by `Django`.

```
>>> Person.collection(firstname__eq='John').instances()
[<[1] John "Joe" Smith (1960)>, <[2] John "Jon" Doe (1965)>]
```

You can also use the `in` suffix and pass an iterable. In this case, all entries that match one of the values is returned.

```
>>> Person.collection(firstname__in=['John', 'Susan']).instances()
[<[1] John "Joe" Smith (1960)>, <[2] John "Jon" Doe (1965)>, <[4] Susan "Sue" Doe␣
↪(1960)>]
```

If you want to do more advanced lookup on a field that contains text, you can use the `TextRangeIndex` (to import from `limpyd.indexes`), as we did for the `nickname` field.

It allows the same filtering as the default index, ie equality without suffix or with the `__eq` or `__in` suffixes, but it is not as efficient.

So if your only usage is equality filtering, prefer `EqualIndex` (which is the default)

But if not, you can take advantage of its capabilities, depending on the suffix you'll use:

- `__gt`: text "Greater Than" the given value
- `__gte`: "Greater Than or Equal"
- `__lt`: "Less Than"
- `__lte`: "Less Than or Equal"
- `__startswith`: text that starts with the given value

Texts are compared in a lexicographical way, as viewed by Redis and explained this way:

> The elements are considered to be ordered from lower to higher strings as compared byte-by-byte using the memcmp() C function. Longer strings are considered greater than shorter strings if the common part is identical.

Some examples:

```
>>> Person.collection(nickname__startswith='Jo').instances()
[<[1] John "Joe" Smith (1960)>, <[2] John "Jon" Doe (1965)>]
>>> Person.collection(nickname__gte='Jo').instances()
[<[1] John "Joe" Smith (1960)>, <[2] John "Jon" Doe (1965)>, <[4] Susan "Sue" Doe␣
↪(1960)>]
>>> Person.collection(nickname__gt='Jo').instances()
[<[4] Susan "Sue" Doe (1960)>]
```

You can filter many times on the same field (more than two times doesn't really make sense):

```
>>> Person.collection(nickname__gte='E', nickname__lte='J').instances()
[<[3] Emily "Emma" Smith (1950)>, <[1] John "Joe" Smith (1960)>, <[2] John "Jon" Doe␣
↪(1965)>]
```

This index works well for text but not for numbers, because lexicographically, `1000 < 11`.

For numbers, you can use the `NumberRangeIndex` (to import from `limpyd.indexes`).

It supports the same suffixes than `TextRangeIndex` excepted for `startswith`.

Some things to know about this index:

- values of a field that cannot be casted to a float are converted to 0 for indexing (the stored value doesn't change).
- negative numbers are, of course, supported
- numbers are saved as the score of a Redis sorted set, so a number is, in the index:

  represented as an IEEE 754 floating point number, that is able to represent precisely integer numbers between -(2^53) and +(2^53) included.

  In more practical terms, all the integers between -9007199254740992 and 9007199254740992 are perfectly representable.

  Larger integers, or fractions, are internally represented in exponential form, so it is possible that you get only an approximation of the decimal number, or of the very big integer.

Some examples:

```
>>> Person.collection(birth_year__eq=1960).instances()
[<[1] John "Joe" Smith (1960)>, <[4] Susan "Sue" Doe (1960)>]
>>> Person.collection(birth_year__gt=1960).instances()
[<[2] John "Jon" Doe (1965)>]
>>> Person.collection(birth_year__gte=1960).instances()
[<[1] John "Joe" Smith (1960)>, <[2] John "Jon" Doe (1965)>, <[4] Susan "Sue" Doe␣
↪(1960)>]
>>> Person.collection(birth_year__gt=1940, birth_year__lte=1950).instances()
[<[3] Emily "Emma" Smith (1950)>]
```

And, of course, you can use fields with different indexes in the same query:

```
>>> Person.collection(birth_year__gte=1960, lastname='Doe', nickname__startswith='S').
↪instances()
[<[4] Susan "Sue" Doe (1960)>]
```

2.5.5.1 Configuration

If you want to use an index with a different behavior, you can use the `configure` class method of the index. Note that you can also create a new class by yourself but we provide this ability.

It accepts one or many arguments (`prefix`, `transform` and `handle_uniqueness`) and returns a new index class to be passed to the `indexes` argument of the field.

About the `prefix` argument:

If you use two indexes accepting the same suffix, for example `eq`, you can specify which one to use on the collection by assigning a prefix to the index:

```
class MyModel(model.RedisModel):
    myfield = fields.StringField(indexable=True, indexes=[
        EqualIndex,
        MyOtherIndex.configure(prefix='foo')
    ])

>>> MyModel.collection(myfield='bar')  # will use EqualIndex
>>> MyModel.collection(myfield__foo='bar')  # will use MyOtherIndex
```

About the `transform` argument:

If you want to index on a value different than the one stored on the field, you can transform it by assigning a transform function to the index.

This function accepts a value as argument and should return the value to store (which will be "normalized", ie converted to string for `EqualIndex` and `TextRangeIndex` and to float for `NumberRangeIndex`)

```
def reverse_value(value):
    return value[::-1]

class MyModel(model.RedisModel):
    myfield = fields.StringField(indexable=True, indexes=[EqualIndex.
↪configure(transform=reverse_value)])

>>> MyModel.collection(myfield__foo='rab')  # query with the expected transformed␣
↪value
```

If you need this function to behave like a method of the index class, you can make it accepts two arguments, `self` and `value`.

About the `handle_uniqueness` argument:

It will simply override the default value set on the index class. Useful if your `transform` function make the value not suitable to check uniqueness, so you can pass it to `False`.

Note that if your field is marked as `unique`, you'll need to have at least one index capable of handling uniqueness.

2.5.5.2 Clean and rebuild

Before removing an index from the field declaration, you have to clean it, else the data will stay in redis.

For this, use the `clean_indexes` method of the field.

```
>>> MyModel.get_field('myfield').clean_indexes()
```

You can also rebuild them. It is useful if you decide to index a field with existing data that was not indexed before.

```
>>> MyModel.get_field('myfield').rebuild_indexes()
```

You can pass the named argument `index_class` to limit the clean/rebuild to only indexes of this class.

Say you defined your own index:

```
MyIndex = EqualIndex(key='yolo', transform=lambda value: 'yolo' + value)
class MyModel(RedisModel):
    myfield = model.StringField(indexable=True, indexes=[TextRangeIndex, EqualIndex])
```

You can clear/rebuild only your own index this way:

```
>>> MyModel.get_field('myfield').clear(index_class=MyIndex)
```

2.5.6 Laziness

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...

2.5.7 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)
```

]******* 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 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.

Let's 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']
```

2.6.1 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.

2.6.2 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 ones. 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 extracted for you.

Here are the new field types:

2.6.2.1 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*.

2.6.2.2 FKInstanceHashField

The *FKInstanceHashField* 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.

2.6.2.3 M2MSetField

The *M2MSetField* type is based on *SetField* and allow setting many foreign keys, acting as a "Many to 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* uniqueness, check *M2MSortedSetField*.

2.6.2.4 M2MListField

The *M2MListField* type is based on *ListField* and allow setting many foreign keys, acting as a "Many to 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 useful to keep the order of the foreign keys, but as it does not ensure uniqueness, the use cases are less obvious.

If you need ordering *AND* uniqueness, check *M2MSortedSetField*.

2.6.2.5 M2MSortedSetField

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

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

This score is useful 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.

2.6.3 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*

2.6.3.1 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')
```

- use `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')
```

2.6.3.2 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 two 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 useful for subclassing:

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

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

(continues on next page)

(continued from previous page)

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

2.6.4 Related collection

A related collection is the other side of the relation. It is created on the related model, based on the *related_name* argument used when creating the related field.

It's a a shortcut to the real collection, but available to ease writing.

Let's define some models and data:

```
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')

>>> 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 the 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']
```

2.6.5 Retrieving the other side

2.6.5.1 Foreign keys

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

Using these models and data:

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

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

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

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()
```

2.6.5.2 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`.

Using these models and data:

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

>>> foo = Person(name='Foo')  # pk=1
>>> bar = Person(name='Bar')  # pk=2
```

(continues on next page)

(continued from previous page)

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

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 M2M field, 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()  # returns a collection
['1', '3']
>>> foo.following().values_list('name', flat=True)
['bar', 'baz']
```

Note that to provide even more consistency, we can call the `collection` method of a M2M field instead of simply "calling" it. So both lines below are the same:

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

2.6.6 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.

2.7 Pipelines

In the contrib module, we provide a way to work with pipelines as defined in redis-py, providing 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*

2.7.1 pipeline

The pipeline provides the same features as for the default pipeline in redis-py, but it handles transparently the use of the pipeline instead of the default connection 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 and some data:

```
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)

>>> 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']
```

This in one only call to the Redis server.

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

2.7.2 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, encapsulated 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 indexable fields in the transaction because read commands are used to update them.

2.7.3 Pipelines and threads

Database connections are shared between threads. The exception is when a pipeline is started. In this case, the pipeline is only used in the current thread that started it.

Other threads still share the original connection and are able to do real commands, out of the pipeline. This behaviour, generally expected, was added in version 1.1

To get the old behaviour, ie share the pipeline between threads, simply pass `share_in_threads` when creating a pipeline:

```
>>> with main_database.pipeline(share_in_threads=True) as pipeline:
...     for person in persons:
...         person.name.get()
...     names = pipeline.execute()
```

This is also valid with transactions.

2.8 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 list 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:

2.8.1 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.

2.8.1.1 values

When calling the `values` method 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': 'John', 'lastname': 'Doe', 'birth_year': '1965'}]
>>> Person.collection(firstname='John').values('pk', 'lastname')
[{'pk': '1', 'lastname': 'Smith'}, {'pk': '2', 'lastname': 'Doe'}]
```

2.8.1.2 values_list

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

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']
```

2.8.2 Chaining filters

With the standard collection, you can chain method class but you cannot add more filters than the ones defined in the initial call to 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 `and`-ed, so if you pass two filters on the same field, you may have an empty result.

2.8.3 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, sorted_set or list (long operation if a list)
- 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)
```

2.8.4 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 your 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
```

2.8.5 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` argument 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

2.8.6 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 Redis 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 reading the `stored_key` attribute 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 if the list is long. 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 left as an exercise for the reader)
>>> 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)
```

2.9 Multi-indexes

If you found yourself adding the same indexes many times to different fields, the `MultiIndexes` class provided in `limpyd.contrib.indexes` can be useful.

Its aim is to let the field only have one index, but in the background, many indexes are managed. The `DateTimeIndex` presented later in this document is a very good example of what it allows to do.

2.9.1 Usage

This works by composition: you compose one index with many ones. So simply call the `compose` class method of the `MultiIndexes` classes:

```
>>> EqualAndRangeIndex = MultiIndexes.compose([EqualIndex, TextRangeIndex])
```

You can pass some arguments to change the behavior:

2.9.1.1 name

The call to `MultiIndexes.compose` will create a new class. The name will be the name of the new class, instead of `MultiIndexes`.

2.9.1.2 key

If you have many indexes based on the same index class (for example `TextRangeIndex`), if they are not prefixed, they will share the same index key. This collision is in general not wanted.

So pass the `key` argument to compose with any string you want.

2.9.1.3 transform

Each index can accept a transform argument, a callable. Same for the multi-indexes. The one passed to `compose` will be applied before the ones on the indexes it contains.

2.9.2 DateTimeIndex

The `limpyd.contrib.indexes` module provides a `DateTimeIndex` (and other friends). In this section we'll explain how it is constructed using only the `configure` method of the normal indexes, and the `compose` method of `MultiIndexes`

2.9.2.1 Goal

We'll store date+times in the format `YYYY-MM-SS HH:MM:SS`.

We want to be able to: - filter on an exact date+time - filter on ranges on the date+time - filter on dates - filter on times - filter on dates parts (year, month, day) - filter on times parts (hour, minute, second)

2.9.2.2 Date and time parts

Let's separate the date, and the time into `YYYY-MM-SS` and `HH:MM:SS`.

How to filter only on the year of a date? Extract the 4 first characters, and filter it as number, using `NumberRangeIndex`:

Also, we don't want uniqueness on this index, and we want to prefix the part to be able to filter with `myfield__year=`

So this part could be:

```
>>> NumberRangeIndex.configure(prefix='year', transform=lambda value: value[:4],␣
↪handle_uniqueness=False, name='YearIndex')
```

Doing the same for the month and day, and composing a multi-indexes with the three, we have:

```
>>> DateIndexParts = MultiIndexes.compose([
...     NumberRangeIndex.configure(prefix='year', transform=lambda value: value[:4],␣
↪handle_uniqueness=False, name='YearIndex'),
...     NumberRangeIndex.configure(prefix='month', transform=lambda value: value[5:7],
↪ handle_uniqueness=False, name='MonthIndex'),
...     NumberRangeIndex.configure(prefix='day', transform=lambda value: value[8:10],␣
↪handle_uniqueness=False, name='DayIndex'),
... ], name='DateIndexParts')
```

If we do the same for the time only (assuming a time field without date), we have:

```
>>> TimeIndexParts = MultiIndexes.compose([
...     NumberRangeIndex.configure(prefix='hour', transform=lambda value: value[0:2],␣
↪handle_uniqueness=False, name='HourIndex'),
...     NumberRangeIndex.configure(prefix='minute', transform=lambda value:␣
↪value[3:5], handle_uniqueness=False, name='MinuteIndex'),
...     NumberRangeIndex.configure(prefix='second', transform=lambda value:␣
↪value[6:8], handle_uniqueness=False, name='SecondIndex'),
... ], name='TimeIndexParts')
```

2.9.2.3 Range indexes

If we want to filter not only on date/time parts but also on the full date with a `TextRangeIndex`, to be able to do `date_field__gt=2015`, we'll need another index.

We don't want to use a prefix, but if we have another `TextRangeIndex` on the field, we need a key:

```
>>> DateRangeIndex = TextRangeIndex.configure(key='date', transform=lambda value:␣
↪value[:10], name='DateRangeIndex')
```

The same for the time:

```
>>> TimeRangeIndex = TextRangeIndex.configure(key='time', transform=lambda value:␣
↪value[:8], name='TimeRangeIndex')
```

We keep theses two indexes apart from the `DateIndexParts` and `TimeIndexParts` because we'll need them independently later to prefix them when used together.

2.9.2.4 Full indexes

If we wan't full indexes for dates and times, including the range and the parts, we can easily compose them:

```
>>> DateIndex = MultiIndexes.compose([DateRangeIndex, DateIndexParts], name='DateIndex
↪')
>>> TimeIndex = MultiIndexes.compose([TimeRangeIndex, TimeIndexParts], name='TimeIndex
↪')
```

Now that we have all what is needed for fields that manage date OR time, we'll combine them. Three things to take in consideration:

- we'll have two `TextRangeIndex`, one for date one for time. So we need to explicitly prefix the filter, to be able to do `datetime_field__date__gt=2015` and `datetime_field__time__gt='15:'`.
- we'll have to extract the date and time separately

- we'll need a `TextRangeIndex` to filter on the whole datetime to be able do to `datetime_field__gt='2015-12-21 15:'`

To start, we want an index without the time parts, to allow filtering on the three "ranges" (full, date, and time), but only on date parts, not time parts. It can be useful if you know you won't have to search on these parts.

So, to summarize, we need:

- a `TextRangeIndex` for the full datetime
- the `DateRangeIndex`, prefixed
- the `DateIndexParts`
- the `TimeRangeIndex`, prefixed

Which gives us:

```
>>> DateSimpleTimeIndex = MultiIndexes.compose([
...     TextRangeIndex.configure(key='full', name='FullDateTimeRangeIndex'),
...     DateRangeIndex.configure(prefix='date'),
...     DateIndexParts,
...     TimeRangeIndex.configure(prefix='time', transform=lambda value: value[11:])
↪# pass only time
... ], name='DateSimpleTimeIndex', transform=lambda value: value[:19])  # restrict on␣
↪date+time
```

And to have the same with the time parts, simply compose a new index with this one and the `TimeIndexPart`:

```
>>> DateTimeIndex = MultiIndexes.compose([
...     DateSimpleTimeIndex,
...     TimeIndexParts.configure(transform=lambda value: value[11:]),  # pass only␣
↪time
... ], name='DateTimeIndex')
```

For simplest cases let's make a `SimpleDateTimeIndex` that doesn't contains parts:

```
>>> SimpleDateTimeIndex = MultiIndexes.compose([
...     TextRangeIndex.configure(key='full', name='FullDateTimeRangeIndex'),
...     DateRangeIndex.configure(prefix='date'),
...     TimeRangeIndex.configure(prefix='time', transform=lambda value: value[11:])
↪# pass only time
... ], name='SimpleDateTimeIndex', transform=lambda value: value[:19])  # restrict on␣
↪date+time
```

And we're done!

2.10 Changelog

2.10.1 Release *v1.3.2* - 2020-05-03

- speed up (more than x1000, no kidding) of limpyd model creation time (backport from v2.1.1)

2.10.2 Release *v1.3.1* - 2019-10-11

- Resolve race condition in *get_or_connect*

2.10.3 Release *v1.3* - 2019-09-22

- Official support for Python 3.7 and 3.8
- Remove support for Python 3.4

2.10.4 Release *v1.2* - 2018-01-31

- Repair packaging

2.10.5 Release *v1.1* - 2018-01-30

- BROKEN RELEASE, sorry
- Official support for redis-py 2.10.6
- Resolve two race conditions (`get` and more important, `pipeline`)
- Add *scan* methods for databases/models/instances/fields (sets, hsets, zsets)
- Add *sort* methods for sets, lists, zsets

2.10.6 Release *v1.0.1* - 2018-01-30

- BROKEN RELEASE, sorry
- Official support for PyPy & PyPy3

2.10.7 Release *v1.0* - 2018-01-29

- BROKEN RELEASE, sorry
- Add real indexing capabilities
- Correct/enhance slicing
- Remove support for python 3.3 (keeps 2.7 and 3.4 to 3.6)

2.10.8 Release *v0.2.4* - 2015-12-16

- Locally solve a locking bug in redis-py

2.10.9 Release *v0.2.3* - 2015-12-16

- Compatibility with Redis-py 2.10

2.10.10 Release *v0.2.2* - 2015-06-12

- Compatibility with pip 6+

2.10.11 Release *v0.2.1* - 2015-01-12

- Stop using dev version of "future"

2.10.12 Release *v0.2.0* - 2014-09-07

- Adding support for python 3.3 and 3.4

2.10.13 Release *v0.1.3* - 2013-09-07

- Add the missing 'hdel' command to the RedisModel class

2.10.14 Release *v0.1.2* - 2013-08-30

- Add the missing 'delete' command to the HashField field

2.10.15 Release *v0.1.1* - 2013-08-26

- Include only the "limpyd" package in setup.py (skip the tests)

2.10.16 Release *v0.1.0* - 2013-02-12

- First public version