Redisco Documentation

Release rc3

Sebastien Requiem

November 29, 2016

Contents

Contents:

CHAPTER 1

Object Relation Manager

Redisco allows you to store objects in Redis. Redisco can easily manage object creation, update and deletion. It is strongly Ohm and Django ORM and try to provide a simple approach.

```
>>> from redisco import models
>>> class Person(models.Model):
...     name = models.Attribute(required=True)
...     created_at = models.DateTimeField(auto_now_add=True)
...     fave_colors = models.ListField(str)

>>> person = Person(name="Conchita")
>>> person.is_valid()
True
>>> person.save()
True
>>> conchita = Person.objects.filter(name='Conchita')[0]
>>> conchita.name
'Conchita'
>>> conchita.created_at
datetime.datetime(2010, 5, 24, 16, 0, 31, 954704)
```

1.1 Model

The `Model` class is the class that will contain your object. It upports many different type of attributes and support custom object validation to ensure data integrity when saving.

class `redisco.models.`**`Model`**`(`***kwargs*`)`

attributes
Return the attributes of the model.

Returns a dict with models attribute name as keys and attribute descriptors as values.

attributes_dict
Returns the mapping of the model attributes and their values.

```
>>> from redisco import models
>>> class Foo(models.Model):
...     name = models.Attribute()
...     title = models.Attribute()
...
>>> f = Foo(name="Einstein", title="Mr.")
```

```
>>> f.attributes_dict
{'name': 'Einstein', 'title': 'Mr.'}
```

counters
Returns the mapping of the counters.

db
Returns the Redis client used by the model.

decr(*att*, *val=1*)
Decrements a counter.

```
>>> from redisco import models
>>> class Foo(models.Model):
...     cnt = models.Counter()
...
>>> f = Foo()
>>> f.save()
True
>>> f.incr('cnt', 10)
>>> f.cnt
10
>>> f.decr('cnt', 2)
>>> f.cnt
8
>>> f.delete()
```

delete()
Deletes the object from the datastore.

errors
Returns the list of errors after validation.

classmethod exists(*id*)
Checks if the model with id exists.

fields
Returns the list of field names of the model.

id
Returns the id of the instance.

Raises MissingID if the instance is new.

incr(*att*, *val=1*)
Increments a counter.

```
>>> from redisco import models
>>> class Foo(models.Model):
...     cnt = models.Counter()
...
>>> f = Foo()
>>> f.save()
True
>>> f.incr('cnt', 10)
>>> f.cnt
10
>>> f.delete()
```

indices
Return a list of the indices of the model. ie: all attributes with index=True.

is_new()
Returns True if the instance is new.

Newness is based on the presence of the _id attribute.

is_valid()
Returns True if all the fields are valid, otherwise errors are in the 'errors' attribute

It first validates the fields (required, unique, etc.) and then calls the validate method.

```
>>> from redisco import models
>>> def validate_me(field, value):
...     if value == "Invalid":
...         return (field, "Invalid value")
...
>>> class Foo(models.Model):
...     bar = models.Attribute(validator=validate_me)
...
>>> f = Foo()
>>> f.bar = "Invalid"
>>> f.save()
False
>>> f.errors
['bar', 'Invalid value']
```

> **Warning:** You may want to use `validate` described below to validate your model

key(*att=None*)
Returns the Redis key where the values are stored.

```
>>> from redisco import models
>>> class Foo(models.Model):
...     name = models.Attribute()
...     title = models.Attribute()
...
>>> f = Foo(name="Einstein", title="Mr.")
>>> f.save()
True
>>> f.key() == "%s:%s" % (f.__class__.__name__, f.id)
True
```

lists
Returns the lists of the model.

Returns a dict with models attribute name as keys and ListField descriptors as values.

references
Returns the mapping of reference fields of the model.

save()
Saves the instance to the datastore with the following steps: 1. Validate all the fields 2. Assign an ID if the object is new 3. Save to the datastore.

```
>>> from redisco import models
>>> class Foo(models.Model):
...     name = models.Attribute()
...     title = models.Attribute()
...
>>> f = Foo(name="Einstein", title="Mr.")
>>> f.save()
```

```
        True
        >>> f.delete()
```

update_attributes(***kwargs*)
Updates the attributes of the model.

```
        >>> from redisco import models
        >>> class Foo(models.Model):
        ...     name = models.Attribute()
        ...     title = models.Attribute()
        ...
        >>> f = Foo(name="Einstein", title="Mr.")
        >>> f.update_attributes(name="Tesla")
        >>> f.name
        'Tesla'
```

validate()
Overriden in the model class. The function is here to help you validate your model. The validation should add errors to self._errors.

Example:

```
        >>> from redisco import models
        >>> class Foo(models.Model):
        ...     name = models.Attribute(required=True)
        ...     def validate(self):
        ...         if self.name == "Invalid":
        ...             self._errors.append(('name', 'cannot be Invalid'))
        ...
        >>> f = Foo(name="Invalid")
        >>> f.save()
        False
        >>> f.errors
        [('name', 'cannot be Invalid')]
```

1.2 Attributes

The attributes are the core of any redisco `Model`. Many different attributes are available according to your needs. Read the following documentation to understand the caveats of each attribute.

Attribute Stores unicode strings. If used for large bodies of text, turn indexing of this field off by setting indexed=True.

IntegerField Stores an int. Ints are stringified using unicode() before saving to Redis.

Counter An IntegerField that can only be accessed via Model.incr and Model.decr.

DateTimeField Can store a DateTime object. Saved in the Redis store as a float.

DateField Can store a Date object. Saved in Redis as a float.

TimeDeltaField Can store a TimeDelta object. Saved in Redis as a float.

FloatField Can store floats.

BooleanField Can store bools. Saved in Redis as 1's and 0's.

ReferenceField Can reference other redisco model.

ListField Can store a list of unicode, int, float, as well as other redisco models.

1.3 Attribute Options

required If True, the attirbute cannot be None or empty. Strings are stripped to check if they are empty. Default is False.

default Sets the default value of the attribute. Default is None.

indexed If True, redisco will create index entries for the attribute. Indexes are used in filtering and ordering results of queries. For large bodies of strings, this should be set to False. Default is True.

validator Set this to a callable that accepts two arguments – the field name and the value of the attribute. The callable should return a list of tuples with the first item is the field name, and the second item is the error.

unique The field must be unique. Default is False.

DateField and DateTimeField Options

auto_now_add Automatically set the datetime/date field to now/today when the object is first created. Default is False.

auto_now Automatically set the datetime/date field to now/today everytime the object is saved. Default is False.

1.4 Modelset

The `ModelSet` class is useful for all kind of queries when you want to filter data (like in SQL). You can filter objects by values in their attributes, creation dates and even perform some unions and exclusions.

```
>>> from redisco import models
>>> class Person(models.Model):
...     name = models.Attribute(required=True)
...     created_at = models.DateTimeField(auto_now_add=True)
...     fave_colors = models.ListField(str)
```

```
>>> person = Person(name="Conchita")
>>> person.save()
True
>>> conchita = Person.objects.filter(name='Conchita').first()
```

class `redisco.models.modelset.`**`ModelSet`**(*model_class*)

`all`()
Return all elements of the collection.

`exclude`(***kwargs*)
Exclude a collection within a lookup.

```
>>> from redisco import models
>>> class Foo(models.Model):
...     name = models.Attribute()
...     exclude_me = models.BooleanField()
...
>>> Foo(name="Einstein").save()
True
>>> Foo(name="Edison", exclude_me=True).save()
True
>>> Foo.objects.exclude(exclude_me=True).first().name
u'Einstein'
```

```
>>> [f.delete() for f in Foo.objects.all()]
[...]
```

filter(***kwargs*)
Filter a collection on criteria

```
>>> from redisco import models
>>> class Foo(models.Model):
...     name = models.Attribute()
...
>>> Foo(name="toto").save()
True
>>> Foo(name="toto").save()
True
>>> Foo.objects.filter()
[<Foo:...>, <Foo:...>]
>>> [f.delete() for f in Foo.objects.all()]
[...]
```

first()
Return the first object of a collections.

Returns The object or Non if the lookup gives no result

```
>>> from redisco import models
>>> class Foo(models.Model):
...     name = models.Attribute()
...
>>> f = Foo(name="toto")
>>> f.save()
True
>>> Foo.objects.filter(name="toto").first()
<Foo:...>
>>> [f.delete() for f in Foo.objects.all()]
[...]
```

get_by_id(*id*)
Returns the object definied by `id`.

Parameters **id** – the `id` of the objects to lookup.

Returns The object instance or None if not found.

```
>>> from redisco import models
>>> class Foo(models.Model):
...     name = models.Attribute()
...
>>> f = Foo(name="Einstein")
>>> f.save()
True
>>> Foo.objects.get_by_id(f.id) == f
True
>>> [f.delete() for f in Foo.objects.all()]
[...]
```

get_or_create(***kwargs*)
Return an element of the collection or create it if necessary.

```
>>> from redisco import models
>>> class Foo(models.Model):
```

```
...     name = models.Attribute()
...
>>> new_obj = Foo.objects.get_or_create(name="Obama")
>>> get_obj = Foo.objects.get_or_create(name="Obama")
>>> new_obj == get_obj
True
>>> [f.delete() for f in Foo.objects.all()]
[...]
```

limit (*n*, *offset=0*)
Limit the size of the collection to *n* elements.

order (*field*)
Enable ordering in collections when doing a lookup.

> **Warning:** This should only be called once per lookup.

```
>>> from redisco import models
>>> class Foo(models.Model):
...     name = models.Attribute()
...     exclude_me = models.BooleanField()
...
>>> Foo(name="Abba").save()
True
>>> Foo(name="Zztop").save()
True
>>> Foo.objects.all().order("-name").first().name
u'Zztop'
>>> Foo.objects.all().order("name").first().name
u'Abba'
>>> [f.delete() for f in Foo.objects.all()]
[...]
```

CHAPTER 2

Containers

A suite of containers is available to the developer (you!) in order to manipulate some of redis' objects. You can easily create, modify, update, and delete Sets, SortedSets, Lists and Hashes. Pay attention that mnay of the operations are serialized to the redis server and are therefore time consuming.

2.1 Base Class

class `redisco.containers.`**`Container`**(*key*, *db=None*, *pipeline=None*)

Base class for all containers. This class should not be used and does not provide anything except the `db` member. :members:

`clear`()

Remove the container from the redis storage

```
>>> s = Set('test')
>>> s.add('1')
1
>>> s.clear()
>>> s.members
set([])
```

`set_expire`(*time=None*)

Allow the key to expire after `time` seconds.

```
>>> s = Set("test")
>>> s.add("1")
1
>>> s.set_expire(1)
>>> # from time import sleep
>>> # sleep(1)
>>> # s.members
# set([])
>>> s.clear()
```

Parameters **`time`** – time expressed in seconds. If time is not specified, then `default_expire_time` will be used.

Return type None

2.2 Set

class redisco.containers.**Set**(*key*, *db=None*, *pipeline=None*)
This class represent a Set in redis.

add(**values*)
see sadd

copy(*key*)
Copy the set to another key and return the new Set.

> **Warning:** If the new key already contains a value, it will be overwritten.

difference(*key*, **other_sets*)
Return a new Set representing the difference of *n* sets.

Parameters

- **key** – String representing the key where to store the result (the union)
- **other_sets** – list of other Set.

Return type *Set*

```
>>> s1 = Set('key1')
>>> s2 = Set('key2')
>>> s1.add(['a', 'b', 'c'])
3
>>> s2.add(['c', 'e'])
2
>>> s3 = s1.difference('key3', s2)
>>> s3.key
u'key3'
>>> s3.members
set(['a', 'b'])
>>> s1.clear()
>>> s2.clear()
>>> s3.clear()
```

difference_update(**other_sets*)
Update the set, removing elements found in others.

Parameters **other_sets** – list of Set

Return type None

intersection(*key*, **other_sets*)
Return a new Set representing the intersection of *n* sets.

Parameters

- **key** – String representing the key where to store the result (the union)
- **other_sets** – list of other Set.

Return type *Set*

```
>>> s1 = Set('key1')
>>> s2 = Set('key2')
>>> s1.add(['a', 'b', 'c'])
3
>>> s2.add(['c', 'e'])
```

```
        2
        >>> s3 = s1.intersection('key3', s2)
        >>> s3.key
        u'key3'
        >>> s3.members
        set(['c'])
        >>> s1.clear()
        >>> s2.clear()
        >>> s3.clear()
```

intersection_update(**other_sets*)

Update the set, keeping only elements found in it and all other_sets.

Parameters **other_sets** – list of `Set`

Return type None

isdisjoint(*other*)

Return True if the set has no elements in common with other.

Parameters **other** – another `Set`

Return type boolean

```
        >>> s1 = Set("key1")
        >>> s2 = Set("key2")
        >>> s1.add(['a', 'b', 'c'])
        3
        >>> s2.add(['c', 'd', 'e'])
        3
        >>> s1.isdisjoint(s2)
        False
        >>> s1.clear()
        >>> s2.clear()
```

issubset(*other_set*)

Test whether every element in the set is in other.

Parameters **other_set** – another `Set` to compare to.

```
        >>> s1 = Set("key1")
        >>> s2 = Set("key2")
        >>> s1.add(['a', 'b', 'c'])
        3
        >>> s2.add('b')
        1
        >>> s2.issubset(s1)
        True
        >>> s1.clear()
        >>> s2.clear()
```

issuperset(*other_set*)

Test whether every element in other is in the set.

Parameters **other_set** – another `Set` to compare to.

```
        >>> s1 = Set("key1")
        >>> s2 = Set("key2")
        >>> s1.add(['a', 'b', 'c'])
        3
        >>> s2.add('b')
```

```
1
>>> s1.issuperset(s2)
True
>>> s1.clear()
>>> s2.clear()
```

members
return the real content of the Set.

pop()
see spop

remove(**values*)
see srem

sadd(**values*)
Add the specified members to the Set.

Parameters **values** – a list of values or a simple value.

Return type integer representing the number of value added to the set.

```
>>> s = Set("test")
>>> s.clear()
>>> s.add(["1", "2", "3"])
3
>>> s.add(["4"])
1
>>> print s
<Set 'test' set(['1', '3', '2', '4'])>
>>> s.clear()
```

scard()
Returns the cardinality of the Set.

Return type String containing the cardinality.

sdiff(**other_sets*)
Performs a difference between two sets and returns the *RAW* result.

Note: This function return an actual `set` object (from python) and not a `Set`. See function difference.

sinter(**other_sets*)
Performs an intersection between Sets and return the *RAW* result.

Note: This function return an actual `set` object (from python) and not a `Set`. See func:`intersection`.

sismember(*value*)
Return `True` if the provided value is in the `Set`.

spop()
Remove and return (pop) a random element from the Set.

Return type String representing the value poped.

```
>>> s = Set("test")
>>> s.add("1")
1
```

```
>>> s.spop()
'1'
>>> s.members
set([])
```

srandmember()
Return a random member of the set.

```
>>> s = Set("test")
>>> s.add(['a', 'b', 'c'])
3
>>> s.srandmember()
'...'
>>> # 'a', 'b' or 'c'
```

srem(**values*)
Remove the values from the Set if they are present.

Parameters **values** – a list of values or a simple value.

Return type boolean indicating if the values have been removed.

```
>>> s = Set("test")
>>> s.add(["1", "2", "3"])
3
>>> s.srem(["1", "3"])
2
>>> s.clear()
```

sunion(**other_sets*)
Performs a union between two sets and returns the *RAW* result.

Note: This function return an actual `set` object (from python) and not a `Set`.

union(*key*, **other_sets*)
Return a new `Set` representing the union of *n* sets.

Parameters

- **key** – String representing the key where to store the result (the union)
- **other_sets** – list of other `Set`.

Return type `Set`

```
>>> s1 = Set('key1')
>>> s2 = Set('key2')
>>> s1.add(['a', 'b', 'c'])
3
>>> s2.add(['d', 'e'])
2
>>> s3 = s1.union('key3', s2)
>>> s3.key
u'key3'
>>> s3.members
set(['a', 'c', 'b', 'e', 'd'])
>>> s1.clear()
>>> s2.clear()
>>> s3.clear()
```

update(**other_sets*)
Update the set, adding elements from all other_sets.

Parameters **other_sets** – list of `Set`

Return type None

2.3 SortedSet

class `redisco.containers.`**SortedSet**(*key*, *db=None*, *pipeline=None*)
This class represents a SortedSet in redis. Use it if you want to arrange your set in any order.

add(*members*, *score=1*)
Add members in the set and assign them the score.

Parameters

- **members** – a list of item or a single item
- **score** – the score the assign to the item(s)

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.add('b', 20)
1
>>> s.clear()
```

between(*min*, *max*, *limit=None*, *offset=None*)
Returns the list of the members of the set that have scores between min and max.

Note: The min and max are inclusive when comparing the values.

Parameters

- **min** – the minimum score to compare to.
- **max** – the maximum score to compare to.
- **limit** – limit the result to *n* elements
- **offset** – Skip the first *n* elements

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.add('b', 20)
1
>>> s.add('c', 30)
1
>>> s.between(20, 30)
['b', 'c']
>>> s.clear()
```

eq(*value*)
Returns the elements that have `value` for score.

ge(*v*, *limit=None*, *offset=None*, *withscores=False*)
Returns the list of the members of the set that have scores greater than or equal to v.

Parameters

- **v** – the score to compare to.
- **limit** – limit the result to *n* elements
- **offset** – Skip the first *n* elements

gt(*v*, *limit=None*, *offset=None*, *withscores=False*)
Returns the list of the members of the set that have scores greater than v.

incr_by(*att*, *value=1*)
Increment the score of the item by `value`

Parameters

- **att** – the member to increment
- **value** – the value to add to the current score

Returns the new score of the member

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.zincrby("a", 10)
20.0
>>> s.clear()
```

le(*v*, *limit=None*, *offset=None*)
Returns the list of the members of the set that have scores less than or equal to v.

Parameters

- **v** – the score to compare to.
- **limit** – limit the result to *n* elements
- **offset** – Skip the first *n* elements

lt(*v*, *limit=None*, *offset=None*)
Returns the list of the members of the set that have scores less than v.

Parameters

- **v** – the score to compare to.
- **limit** – limit the result to *n* elements
- **offset** – Skip the first *n* elements

members
Returns the members of the set.

rank(*elem*)
Returns the rank of the element.

```
>>> s = SortedSet("foo")
>>> s.add("a", 10)
1
>>> s.zrank("a")
0
>>> s.clear()
```

remove(**values*)
Remove the values from the SortedSet

Returns True if **at least one** value is successfully removed, False otherwise

```
>>> s = SortedSet('foo')
>>> s.add('a', 10)
1
>>> s.zrem('a')
1
>>> s.members
[]
>>> s.clear()
```

revmembers
Returns the members of the set in reverse.

revrank(*member*)
Returns the ranking in reverse order for the member

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.add('b', 20)
1
>>> s.revrank('a')
1
>>> s.clear()
```

score(*elem*)
Return the score of an element

```
>>> s = SortedSet("foo")
>>> s.add("a", 10)
1
>>> s.score("a")
10.0
>>> s.clear()
```

zadd(*members*, *score=1*)
Add members in the set and assign them the score.

Parameters

- **members** – a list of item or a single item
- **score** – the score the assign to the item(s)

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.add('b', 20)
1
>>> s.clear()
```

zcard()
Returns the cardinality of the SortedSet.

```
>>> s = SortedSet("foo")
>>> s.add("a", 1)
1
```

```
>>> s.add("b", 2)
1
>>> s.add("c", 3)
1
>>> s.zcard()
3
>>> s.clear()
```

zincrby(*att*, *value=1*)

Increment the score of the item by `value`

Parameters

- **att** – the member to increment
- **value** – the value to add to the current score

Returns the new score of the member

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.zincrby("a", 10)
20.0
>>> s.clear()
```

zrange(*start*, *stop*, *withscores=False*)

Returns all the elements including between `start` (non included) and `stop` (included).

Parameters **withscore** – True if the score of the elements should also be returned

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.add('b', 20)
1
>>> s.add('c', 30)
1
>>> s.zrange(1, 3)
['b', 'c']
>>> s.zrange(1, 3, withscores=True)
[('b', 20.0), ('c', 30.0)]
>>> s.clear()
```

zrangebyscore(*min*, *max*, ***kwargs*)

Returns the range of elements included between the scores (min and max)

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.add('b', 20)
1
>>> s.add('c', 30)
1
>>> s.zrangebyscore(20, 30)
['b', 'c']
>>> s.clear()
```

zrank(*elem*)

Returns the rank of the element.

```
>>> s = SortedSet("foo")
>>> s.add("a", 10)
1
>>> s.zrank("a")
0
>>> s.clear()
```

zrem(**values*)
Remove the values from the SortedSet

Returns True if **at least one** value is successfully removed, False otherwise

```
>>> s = SortedSet('foo')
>>> s.add('a', 10)
1
>>> s.zrem('a')
1
>>> s.members
[]
>>> s.clear()
```

zremrangebyrank(*start*, *stop*)
Remove a range of element between the rank `start` and `stop` both included.

Returns the number of item deleted

```
>>> s = SortedSet("foo")
>>> s.add("a", 10)
1
>>> s.add("b", 20)
1
>>> s.add("c", 30)
1
>>> s.zremrangebyrank(1, 2)
2
>>> s.members
['a']
>>> s.clear()
```

zremrangebyscore(*min_value*, *max_value*)
Remove a range of element by between score `min_value` and `max_value` both included.

Returns the number of items deleted.

```
>>> s = SortedSet("foo")
>>> s.add("a", 10)
1
>>> s.add("b", 20)
1
>>> s.add("c", 30)
1
>>> s.zremrangebyscore(10, 20)
2
>>> s.members
['c']
>>> s.clear()
```

zrevrange(*start*, *end*, ***kwargs*)
Returns the range of items included between `start` and `stop` in reverse order (from high to low)

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.add('b', 20)
1
>>> s.add('c', 30)
1
>>> s.zrevrange(1, 2)
['b', 'a']
>>> s.clear()
```

zrevrangebyscore(*max*, *min*, ***kwargs*)
Returns the range of elements included between the scores (min and max)

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.add('b', 20)
1
>>> s.add('c', 30)
1
>>> s.zrangebyscore(20, 20)
['b']
>>> s.clear()
```

zrevrank(*member*)
Returns the ranking in reverse order for the member

```
>>> s = SortedSet("foo")
>>> s.add('a', 10)
1
>>> s.add('b', 20)
1
>>> s.revrank('a')
1
>>> s.clear()
```

zscore(*elem*)
Return the score of an element

```
>>> s = SortedSet("foo")
>>> s.add("a", 10)
1
>>> s.score("a")
10.0
>>> s.clear()
```

2.4 List

class redisco.containers.**List**(*key*, *db=None*, *pipeline=None*)
This class represent a list object as seen in redis.

all()
Returns all items in the list.

append(**values*)
Push the value into the list from the *right* side

Parameters **values** – a list of values or single value to push

Return type long representing the size of the list.

```
>>> l = List("test")
>>> l.lpush(['a', 'b'])
2L
>>> l.rpush(['c', 'd'])
4L
>>> l.members
['b', 'a', 'c', 'd']
>>> l.clear()
```

copy(*key*)
Copy the list to a new list.

..WARNING: If destination key already contains a value, it clears it before copying.

count(*value*)
Return number of occurrences of value.

Parameters **value** – a value tha *may* be contained in the list

extend(*iterable*)
Extend list by appending elements from the iterable.

Parameters **iterable** – an iterable objects.

lindex(*idx*)
Return the value at the index *idx*

Parameters **idx** – the index to fetch the value.

Returns the value or None if out of range.

llen()
Returns the length of the list.

lpop()
Pop the first object from the left.

Returns the popped value.

lpush(**values*)
Push the value into the list from the *left* side

Parameters **values** – a list of values or single value to push

Return type long representing the number of values pushed.

```
>>> l = List("test")
>>> l.lpush(['a', 'b'])
2L
>>> l.clear()
```

lrange(*start*, *stop*)
Returns a range of items.

Parameters

- **start** – integer representing the start index of the range
- **stop** – integer representing the size of the list.

```
>>> l = List("test")
>>> l.push(['a', 'b', 'c', 'd'])
4L
>>> l.lrange(1, 2)
['b', 'c']
>>> l.clear()
```

lrem(*value*, *num=1*)
Remove first occurrence of value.

Returns 1 if the value has been removed, 0 otherwise

lset(*idx*, *value=0*)
Set the value in the list at index *idx*

Returns True is the operation succeed.

```
>>> l = List('test')
>>> l.push(['a', 'b', 'c'])
3L
>>> l.lset(0, 'e')
True
>>> l.members
['e', 'b', 'c']
>>> l.clear()
```

ltrim(*start*, *end*)
Trim the list from start to end.

Returns None

members
Return all items in the list.

pop()
Pop the first object from the right.

Returns the popped value.

pop_onto(*key*)
Remove an element from the list, atomically add it to the head of the list indicated by key

Parameters **key** – the key of the list receiving the popped value.

Returns the popped (and pushed) value

```
>>> l = List('list1')
>>> l.push(['a', 'b', 'c'])
3L
>>> l.rpoplpush('list2')
'c'
>>> l2 = List('list2')
>>> l2.members
['c']
>>> l.clear()
>>> l2.clear()
```

push(**values*)
Push the value into the list from the *right* side

Parameters **values** – a list of values or single value to push

Return type long representing the size of the list.

```
>>> l = List("test")
>>> l.lpush(['a', 'b'])
2L
>>> l.rpush(['c', 'd'])
4L
>>> l.members
['b', 'a', 'c', 'd']
>>> l.clear()
```

remove(*value*, *num=1*)
Remove first occurrence of value.

Returns 1 if the value has been removed, 0 otherwise

reverse()
Reverse the list in place.

Returns None

rpop()
Pop the first object from the right.

Returns the popped value.

rpoplpush(*key*)
Remove an element from the list, atomically add it to the head of the list indicated by key

Parameters **key** – the key of the list receiving the popped value.

Returns the popped (and pushed) value

```
>>> l = List('list1')
>>> l.push(['a', 'b', 'c'])
3L
>>> l.rpoplpush('list2')
'c'
>>> l2 = List('list2')
>>> l2.members
['c']
>>> l.clear()
>>> l2.clear()
```

rpush(**values*)
Push the value into the list from the *right* side

Parameters **values** – a list of values or single value to push

Return type long representing the size of the list.

```
>>> l = List("test")
>>> l.lpush(['a', 'b'])
2L
>>> l.rpush(['c', 'd'])
4L
>>> l.members
['b', 'a', 'c', 'd']
>>> l.clear()
```

shift()
Pop the first object from the left.

Returns the popped value.

trim(*start*, *end*)
Trim the list from start to end.

Returns None

unshift(**values*)
Push the value into the list from the *left* side

Parameters **values** – a list of values or single value to push

Return type long representing the number of values pushed.

```
>>> l = List("test")
>>> l.lpush(['a', 'b'])
2L
>>> l.clear()
```

2.5 Hash

class redisco.containers.**Hash**(*key*, *db=None*, *pipeline=None*)

dict
Returns all the fields and values in the Hash.

Return type *dict*

hdel(**members*)
Delete one or more hash field.

Parameters **members** – on or more fields to remove.

Returns the number of fields that were removed

```
>>> h = Hash("foo")
>>> h.hset("bar", "value")
1L
>>> h.hdel("bar")
1
>>> h.clear()
```

hexists(*field*)
Returns `True` if the field exists, `False` otherwise.

hget(*field*)
Returns the value stored in the field, None if the field doesn't exist.

hgetall()
Returns all the fields and values in the Hash.

Return type *dict*

hincrby(*field*, *increment=1*)
Increment the value of the field. :returns: the value of the field after incrementation

```
>>> h = Hash("foo")
>>> h.hincrby("bar", 10)
10L
>>> h.hincrby("bar", 2)
12L
>>> h.clear()
```

hkeys()
Returns all fields name in the Hash

hlen()
Returns the number of elements in the Hash.

hmget(*fields*)
Returns the values stored in the fields.

hmset(*mapping*)
Sets or updates the fields with their corresponding values.

Parameters **mapping** – a dict with keys and values

hset(*member*, *value*)
Set `member` in the Hash at `value`.

Returns 1 if member is a new field and the value has been stored, 0 if the field existed and the value has been updated.

```
>>> h = Hash("foo")
>>> h.hset("bar", "value")
1L
>>> h.clear()
```

hvals()
Returns all the values in the Hash

Return type list

keys()
Returns all fields name in the Hash

values()
Returns all the values in the Hash

Return type list

CHAPTER 3

Indices and tables

- genindex
- modindex
- search