redish Documentation

Release 0.2.0

Ask Solem

Sep 14, 2017

Contents

Contents:

CHAPTER 1

redish - Pythonic Redis abstraction built on top of redis-py

Version 0.2.0

Introduction

The client

A connection to a database is represented by the `redish.client.Client` class:

```
>>> from redish.client import Client
>>> db = Client()
>>> db = Client(host="localhost", port=6379, db="") # default settings.
>>> db
<RedisClient: localhost:6379/>
```

Serializers

Clients can be configured to automatically serialize and deserialize values. There are three serializers shipped with `redish`:

- `Plain`

The plain serializer does not serialize values, but does still support compression using the `encoding` argument.

Note that this means you can only store string values in keys.

Example:

```
>>> from redish import serialization
>>> db = Client(serializer=serialization.Plain())
```

- `Pickler`

Uses the `pickle` module to serialize Python objects. This can store any object except lambdas or objects not resolving back to a module.

Example:

```
>>> from redish import serialization
>>> db = Client(serializer=serialization.Pickler())
```

- `JSON`:

Stores values in JSON format. This supports lists, dicts, strings, numbers, and floats. Complex Python objects can not be stored using JSON. The upside is that it is commonly supported by other languages and platforms.

Example:

```
>>> from redish import serialization
>>> db = Client(serializer=serialization.JSON())
```

Compression

In addition these serializers can also be configured to do compression:

```
# Using zlib compression
>>> db = Client(serializer=serialization.Pickler(encoding="zlib"))
```

Working with keys and values

Set a value:

```
>>> db["foo"] = {"name": "George"}
```

Get value by key:

```
>>> db["foo"]
{'name': 'George'}
```

Delete key:

```
>>> del(db["foo"])
```

Getting nonexistent values works like you would expect from Python dictionaries; It raises the KeyError exception:

```
>>> db["foo"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "redish/client.py", line 198, in __getitem__
    raise KeyError(key)
KeyError: 'foo'
```

Set many keys at the same time:

```
>>> db.update({"name": "George Costanza",
...            "company": "Vandelay Industries"})
```

Get a list of keys in the database:

```
>>> db.keys()
['company', 'name']
```

Get a list of keys matching a pattern:

```
>>> db.keys(pattern="na*")
['name']
```

Rename keys:

```
>>> db.rename("name", "user:name")
>>> db.rename("company", "user:company")
>>> db.keys("user:*")
['user:company', 'user:name']
```

Get all items in the database (optionally matching a pattern) as a list of `(key, value)` tuples:

```
>>> db.items(pattern="user:*")
[('user:company', 'Vandelay Industries'), ('user:name', 'George Costanza')]
```

Get all values in the database (optionally where keys matches a pattern):

```
>>> db.values(pattern="user:*")
['Vandelay Industries', 'George Costanza']
```

Iterator versions of `keys`, `values` and `items` are also available, as `iterkeys`, `itervalues`, `iteritems` respectively.

Check for existence of a key in the database:

```
>>> "user:name" in db
True
>>> "user:address" in db
False
>>> "user:address" not in db
True
```

Get and remove key from the database (atomic operation):

```
>>> db.pop("user:name")
'George Costanza'
>>> "user:name" in db
False
```

Get the number of keys present in the database:

```
>>> len(db)
1
```

Lists

Note: Lists does not currently support storing serialized objects.

Create a new list with key `mylist`, and initial items:

```
>>> l = db.List("mylist", ["Jerry", "George"])
```

Get items in the list as a Python `list`:

```
>>> list(l)
['Jerry', 'George']
```

`append` adds items to the end of the list:

```
>>> l.append("Kramer")
>>> list(l)
['Jerry', 'George', 'Kramer']
```

`appendleft` prepends item to the head of the list:

```
>>> l.appendleft("Elaine")
>>> list(l)
['Elaine', 'Jerry', George', 'Kramer']
```

Get item at index (zero based):

```
>>> l[2]
'George'
```

Check if a value is in the list using the `in` operator:

```
>>> "George" in l
True

>>> "Soup-nazi" in l
False
```

`pop` removes and returns the last element of the list:

```
>>> list(l)
['Elaine', 'Jerry', 'George', 'Kramer']
>>> l.pop()
'Kramer'
>>> list(l)
['Elaine', 'Jerry', 'George']
```

`popleft` removes and returns the head of the list:

```
>>> l.popleft()
'Elaine'
>>> list(l)
['Jerry', 'George']
```

Get the number of items in the list:

```
>>> len(l)
2
```

`extend` adds another list to the end of the list:

```
>>> l.extend(["Elaine", "Kramer"])
>>> list(l)
['Jerry', 'George', 'Elaine', 'Kramer']
```

`extendleft` adds another list to the head of the list:

```
>>> l.extendleft(["Soup-nazi", "Art"])
>>> list(l)
['Art', 'Soup-nazi', 'Jerry', 'George', 'Elaine', 'Kramer']
```

Get slice of list:

```
>>> l[2:4]
['Jerry', 'George']
```

Iterate over the lists items:

```
>>> it = iter(l)
>>> it.next()
'Art'
```

`remove` finds and removes one or more occurences of `value` from the list:

```
>>> l.remove("Soup-nazi", count=1)
1
>>> list(l)
['Art', 'Jerry', 'George', 'Elaine', 'Kramer']
```

`trim` trims the list to the range in `start`, `stop`:

```
>>> l[2:4]
['George', 'Elaine']
>>> l.trim(start=2, stop=4)
>>> list(l)
['George', 'Elaine']
```

Dicts (Hashes)

Create a new dictionary with initial content:

```
>>> d = db.Dict("mydict", {"name": "George Louis Costanza"})
```

Get the value of key `"name"`:

```
>>> d["name"]
'George Louis Costanza'
```

Set store another key, `"company"`:

```
>>> d["company"] = "Vandelay Industries"
```

Check if a key exists in the dictionary, using the `in` operator:

```
>>> "company" in d
True
```

Remove a key:

```
>>> del(d["company"])
>>> "company" in d
False
```

Get a copy as a Python `dict`:

```
>>> dict(d)
{'name': 'George Louis Costanza'}
```

`update` updates with the contents of a `dict` (`x.update(y)` does a merge where keys in `y` has precedence):

```
>>> d.update({"mother": "Estelle Costanza",
...           "father": "Frank Costanza"})

>>> dict(d)
{'name': 'George Louis Costanza',
 'mother': 'Estelle Costanza',
 'father': 'Frank Costanza'}
```

Get the number of keys in the dictionary:

```
>>> len(d)
3
```

`keys` / `iterkeys` gives a list of the keys in the dictionary:

```
>>> d.keys()
['name', 'father', 'mother']
```

`values` / `itervalues` gives a list of values in the dictionary:

```
>>> d.values()
['George Louis Costanza', 'Frank Costanza', 'Estelle Costanza']
```

`items` / `iteritems` gives a list of `(key, value)` tuples of the items in the dictionary:

```
>>> d.items()
[('father', 'Frank Costanza'),
 ('name', 'George Louis Costanza'),
 ('mother', 'Estelle Costanza')]
```

`setdefault` returns the value of a key if present, otherwise stores a default value:

```
>>> d.setdefault("company", "Vandelay Industries")
'Vandelay Industries'
>>> d["company"] = "New York Yankees"
>>> d.setdefault("company", "Vandelay Industries")
'New York Yankees'
```

`get(key, default=None)` returns the value of a key if present, otherwise returns the default value:

```
>>> d.get("company")
"Vandelay Industries"

>>> d.get("address")
None
```

pop removes a key and returns its value. Also supports an extra parameters, which is the default value to return if the key does not exist:

```
>>> d.pop("company")
'New York Yankees'
>>> d.pop("company")
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "redish/types.py", line 373, in pop
        val = self[key]
    File "redish/types.py", line 290, in __getitem__
        raise KeyError(key)
KeyError: 'company'

# With default value, does not raise KeyError, but returns default value.
>>> d.pop("company", None)
None
```

Sets

Create a new set with the key myset, and initial members "Jerry" and "George":

```
>>> s = db.Set("myset", ["Jerry", "George"])
```

Add member "Elaine" to the set:

```
>>> s.add("Elaine")
```

Check for membership:

```
>>> "Jerry" in s
True

>>> "Cosmo" in s:
False
```

Remove member from set:

```
>>> s.remove("Elaine")
>>> "Elaine" in s
False
```

Get copy of the set as a list:

```
>>> list(s)
['Jerry', 'George']
```

Create another set:

```
>>> s2 = x.Set("myset2", ["Jerry", "Jason", "Julia", "Michael")
```

Get the difference of the second set and the first:

```
>>> s2.difference(s)
set(['Jason', 'Michael', 'Julia'])
```

Get the union of the two sets:

```
>>> s.union(s2)
set(['Jason', 'Michael', 'Jerry', 'Julia', 'George'])
```

Get the intersection of the two sets:

```
>>> s.intersection(s2)
set(['Jerry'])
```

Update the set with the union of another:

```
>>> s.update(s2)
5
>>> s
<Set: ['Jason', 'Michael', 'Jerry', 'Julia', 'George']>
```

Sorted sets

Create a new sorted set with the key `myzset`, and initial members:

```
>>> z = db.SortedSet("myzset", (("foo", 0.9), ("bar", 0.1), ("baz", 0.3)))
```

Casting to list gives the members ordered by score:

```
>>> list(z)
['bar', 'baz', 'foo']
```

`revrange` sorts the members in reverse:

```
>>> z.revrange()
['foo', 'baz', 'bar']
```

`score` gives the current score of a member:

```
>>> z.score("foo")
0.90000000000000002
```

`add` adds another member:

```
>>> z.add("zaz", 1.2)
>>> list(z)
['bar', 'baz', 'foo', 'zaz']
```

`increment` increments the score of a member by `amount` (or 1 by default):

```
>>> z.increment("baz")
1.3
>>> z.increment("bar", 0.2)
0.30000000000000004
>>> list(z)
['bar', 'foo', 'zaz', 'baz']
```

Check for membership using the `in` operator:

```
>>> "bar" in z
True

>>> "xuzzy" in z
False
```

`remove` removes a member:

```
>>> z.remove("zaz")
>>> "zaz" in z
False
```

`update` updates the sorted set with members from an iterable of `(member, score)` tuples:

```
>>> z.update([("foo", 0.1), ("xuzzy", 0.6)])
>>> list(z)
['foo', 'bar', 'xuzzy', 'baz']
```

`rank` gives the position of a member in the set (0-based):

```
>>> z.rank("foo")
0
>>> z.rank("xuzzy")
2
```

`revrank` gives the position of a member in reverse order:

```
>>> z.revrank("foo")
3
>>> z.revrank("baz")
0
```

`range_by_score` gives all the member with score within a range (`min` / `max`):

```
>>> z.range_by_score(min=0.3, max=0.6)
['bar', 'xuzzy']
```

redish.proxy

The proxy submodule offers a different view on the redis datastore: it exposes the strings, integers, lists, hashes, sets and sorted sets within the datastore transparently, as if they were native Python objects accessed by key on the proxy object. They do not store serialized objects as with the rest of redish. For example:

```
>>> from redish import proxy
>>> r = proxy.Proxy()
```

Key access yields an object that acts like the Python equivalent of the underlying Redis structure. That structure can be read and modified as if it is native, local object. Here, that object acts like a dict:

```
>>> r['mydict']
{'father': 'Frank Costanza', 'name': 'George Louis Costanza', 'mother': 'Estelle␣
↪Costanza'}
>>> r['mydict']['name']
'George Louis Costanza'
>>> r['mydict']['name'] = "Georgie"
```

```
>>> r['mydict']['name']
'Georgie'
```

Sometimes, it may be convenient to assign a variable to the proxy object, and use that in subsequent operations:

```
>>> ss = r['myset']
>>> 'George' in ss
True
>>> 'Ringo' in ss
False
```

The Proxy object is a subclass of a normal redis.Client object, and so supports the same methods (other than *__getitem__*, *__setitem__*, *__contains__*, and *__delitem__*). The object that the proxy object returns is an instance of one of the classes from redish.types (with the exception of unicode: those are simply serialized/unserialized from the underlying redis data store as UTF-8).

```
>>> r['mycounter'] = 1
>>> cc = r['mycounter']
>>> cc += 1
>>> cc += 1
>>> r.get('mycounter')
'3'
>>> type(cc)
<class 'redish.types.Int'>
```

Since redis does not support empty sets, lists, or hashes, the proxy object will (thread-)locally 'remember' keys that are explicitly set as empty types. It does not currently remember container types that have been emptied as a product of operations on the underlying store:

```
>>> r['newlist'] = []
>>> r['newlist'].extend([1,2])
>>> len(r['newlist'])
2
```

Finally, you may structure key names into arbitrary "keyspaces" denoted by format strings:

```
>>> name = r.keyspace['user:%04d:name']
>>> parents = r.keyspace['user:%04d:parents']
>>> property = r.keyspace['user:%04d:%s']
>>> name[1] = 'Jerry'
>>> property[1,'parents'] = ['Morty', 'Helen']
>>> parents.items()
('user:0001:parents', ['Morty', 'Helen'])
```

For more information, see the redish.proxy documentation.

Installation

You can install `redish` either via the Python Package Index (PyPI) or from source.

To install using `pip`,:

```
$ pip install redish
```

To install using `easy_install`,:

```
$ easy_install redish
```

If you have downloaded a source tarball you can install it by doing the following,:

```
$ python setup.py build
# python setup.py install # as root
```

CHAPTER 2

Accessing Redis via the Proxy object

By mixing the type system from redish with the original redis-py's Redis object, the *redish.proxy* module gives a different kind of access to the key-value store without pickling/unpickling and by respecting the strengths in Redis's types. In other words, it transparently exposes Redis as a data structure server.

Basics

Example:

```
>>> from redish import proxy
>>> x = proxy.Proxy()
```

Ordinary key/value usage encodes strings as UTF-8, and returns unicode objects when accessing by key:

```
>>> x["foo"] = "bar"
>>> x["foo"]
u'bar'
```

Deletion and invalid keys work as expected:

```
>>> del x["foo"]
>>> x["foo"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "redish/proxy.py", line 29, in __getitem__
    raise KeyError(key)
KeyError: 'foo'
```

Integer (or Counter)

When reading an integer, the proxy transparently passes an object that mimics the behaviour of int, but is actually stored and fetched from the Redis store:

```
>>> x["z"] = 1
>>> x["z"]
1
>>> x["z"].__class__
<class 'redish.types.Int'>
```

Increment and decrement operations translate directly to Redis commands:

```
>>> x["z"] += 2
>>> x["z"]
3
```

And all other operations you would expect to perform on an integer are present:

```
>>> 3. * x["z"]
9.0
```

The proxy object can be assigned to another variable, and it will point to the same value in the Redis store. For the most part, you can treat the variable the same way:

```
>>> z = x["z"]
>>> z += 1
>>> z
4
>>> x["z"]
4
```

However, it is not the same in all respects: reassigning the variable is not the same as setting the value on the key:

```
>>> z = 5
>>> x["z"]
4
>>> z
5
>>> z.__class__
<type 'int'>
```

Dictionary

By assigning a key to a dictionary type, a hash data type is created in the Redis store. When the key is accessed, it returns a type that mimics a python dict:

```
>>> x["dictionary"] = {"a": "b", "c": "d"}
>>> x["dictionary"]
{'a': 'b', 'c': 'd'}
```

You may access, create, test, and destroy keys within that hash as if they were native Python keys in a dict:

```
>>> x["dictionary"]["c"]
'd'
>>> x["dictionary"]["e"] = "f"
>>> "e" in x["dictionary"]
True
>>> x["dictionary"].__class__
<class 'redish.types.Dict'>
```

List

By assigning a key in the Proxy object to a list type, a list is created in the Redis store. When the key is accessed, it returns a type that mimics a python list:

```
>>> x["Liszt"] = ['w', 'x', 'y', 'z']
>>> x["Liszt"]
['w', 'x', 'y', 'z']
>>> x["Liszt"].extend(["a", "b", "c"])
>>> x["Liszt"]
['w', 'x', 'y', 'z', 'a', 'b', 'c']
>>> x["Liszt"][-1]
'c'
>>> x["Liszt"].pop()
'c'
>>> x["Liszt"][-1]
'b'
```

Set

By assigning a key in the Proxy object to a set type, a set is created in the Redis store. When the key is accessed, it returns a type that mimics a python set:

```
>>> x["set"] = set(["opera", "firefox", "ie", "safari"])
>>> s = x["set"]
>>> "opera" in s
True
>>> s.remove("safari")
>>> "safari" in s
False
>>> list(s)
['opera', 'ie', 'firefox']
```

It may be useful to point out that assignment to a key on the proxy object copies by value:

```
>>> x["game"] = x["set"]
>>> x["game"].add("mobilesafari")
True
>>> x["game"]
set(['opera', 'ie', 'firefox', 'mobilesafari'])
>>> x["set"]
set(['opera', 'ie', 'firefox'])
```

Sorted Set

There is no native Python equivalent of a Sorted Set. However, it resembles a specialized dictionary in which all the values are numeric. The local implementation of the Sorted Set type (ZSet) uses a dictionary in this way to initialize its values:

```
>>> from redish.types import ZSet
>>> zs = ZSet({'c': 3, 'b': 2, 'a': 1})
>>> zs
['a', 'b', 'c']
>>> zs[-1]
'c'
```

The proxied equivalent in which the data resides on the Redis server is created when setting a key to an object of the ZSet class, and is generated when retrieving such a set:

```
>>> x["zs"] = zs
>>> x["zs"].rank("a")
0
>>> x["zs"].range_by_score(2,3)
['b', 'c']
>>> x["zs"].remove("c")
>>> x["zs"].items()
[('a', 1.0), ('b', 2.0)]
```

Proxy objects in general

A Proxy object retains all the normal methods from Redis object:

```
>>> x.keys()
['z', 'dictionary', 'Liszt', 'set', 'game']
>>> x.bgsave()
True
```

Keyspaces in proxy objects

The fact that Redis offers a flat keyspace in each of its databases is a great benefit: it does not presuppose any structure for the keys, and access is fast and unencumbered. However, users are likely to want some structure in using keys, and the Keyspaces feature is a first attempt at making key name patterns accessible to users.

At the heart, a "keyspace" is a formatstring with an associated label. Access is achieved by accessing elements in the proxy with a tuple argument, with the label as the first element of the tuple and the following elements used as inputs to the formatstring:

```
>>> x.register_keyspace('myspace', "person:%04d:name")
'myspace'
>>> x['myspace', 1] = "Bob"
>>> x['person:0001:name']
u'Bob'
```

The label string is returned to facilitate structured and symbolic use of the keyspaces, so the following is equivalent to the above:

```
>>> UNAME = x.register_keyspace('myspace', "person:%04d:name")
>>> x[UNAME, 1] = "Bob"
>>> x['myspace', 1]
u'Bob'
```

One can debug the keyspaces by feeding a tuple to `actual_key`:

```
>>> x.actual_key((UNAME, 202))
'person:0202:name'
```

One can also obtain a keyspace as a subset of all the keys in the database, allowing you to treat the keyspace as a dict:

```
>>> names = x.keyspace(UNAME)
>>> names[1]
u'Bob'
```

If you like, you can bypass labeling altogether and initialize a keyspace using a formatstring alone as a pattern:

```
>>> namez = x.keyspace("person:%04d:name")
>>> namez[1]
u'Bob'
```

Not only can you get `keys` that match a (glob-style) pattern, as in `redis.keys()`, but you can also get `values` and `items`. When fed a keyspace label as an argument, the formatstring is converted to a glob-style pattern. When used with keyspaced proxies, no argument is needed, and the keyspace's formatstring is converted into a glob-style pattern. The following are thus equivalent:

```
>>> r.keys('person:*:name')
['person:0001:name']
>>> r.keys('myspace')
['person:0001:name']
>>> names.keys()
['person:0001:name']
```

All these features can be combined:

```
>>> ZZ = x.register_keyspace('friends', '%(type)s:%(id)04d:friends')
>>> friendstore = x.keyspace(ZZ)
>>> namestore = x.keyspace('%(type)s:%(id)04d:name')
>>> frank = {'type': 'person', 'id': 203,
...          'friends': set([204, 1]), 'name': 'Frank'}
>>> fido = {'type': 'pet', 'id': 204,
...          'name': 'Fido', 'friends': set([1, 202])}
>>> for o in [frank, fido]:
...     friendstore[o] = o['friends']
...     namestore[o] = o['name']
>>> x['person:0203:friends']
<Set: ['1', '204']>
>>> x['pet:0204:friends'].intersection(friendstore[frank])
set(['1'])
>>> friendstore.items()
[('person:0203:friends', <Set: ['1', '204']>),
 ('pet:0204:friends', <Set: ['1', '202']>)]
>>> namestore[frank]
u'Frank'
```

I have no idea at this point if these experimental features are useful to others, but they are fairly minimal, independent, and make sense to me. Feedback is appreciated.

CHAPTER 3

API Reference

Release 0.2

Date Sep 14, 2017

Database - redish.client

Datatypes - redish.types

Models - redish.models

Proxy - redish.proxy

Serialization - redish.serialization

Utilities - redish.utils

redish.utils.**dt_to_timestamp**(*dt*)
Convert `datetime` to UNIX timestamp.

redish.utils.**maybe_datetime**(*timestamp*)
Convert datetime to timestamp, only if timestamp is a datetime object.

CHAPTER 4

Change History

0.0.1 [2010-04-29 04:40 P.M CET]

- Initial release.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index