redislite Documentation

Release 3.2.313

Yahoo Inc.

May 23, 2017

Contents

Redislite provides a self-configuring Redis Key-Value store in a Python module.

This makes it possible to use Redis without the need to install and configure a redis server.

CHAPTER 1

Table of Contents

What is redislite

The *redislite* module contains a complete redis server along with enhanced redis bindings that automatically set up and run the embedded redis server when accessed and shutdown and cleanup the redis server when python exits.

Enhanced Redis Bindings

The enhanced redis bindings include extended versions of the `redis.Redis()` and `redis.StrictRedis()` classes. It also contains functions to patch the redis module to use these new extended classes.

Secure By Default

Redislite defaults to a redis server configuration that is more secure than the default configuration for the redis server. This is due to the following differences:

- Redislite defaults to using unix domain sockets for the redis connection. So the server is not accessible over the computer network by default.
- Redislite locks down the permissions of the unix domain sockets used to only allow the creating user access.

Backwards Compatibility

To provide compatibility with existing python code that uses the redis bindings. Redislite provides functions to patch the `redis` module to use the *redislite* module. This allows most existing code that uses redis to use the redislite features.

How redislite works

redislite provides enhanced versions of the `redis.Redis()` and `redis.StrictRedis()` classes.

These enhanced classes accept the same arguments as the corresponding redis classes.

Both enhanced classes accept one additional optional argument, which is the filename to use for the redis rdb file.

These enhanced classes provide the following additional functionality:

- They configure and start an embedded copy of the redis server running on a unix domain socket in the redislite temp directory for the communication to the redis service.
- TCP communication is disabled by default, unless server settings are passed to enable it.
- **The classes have two additional attributes:**
  - db - The path to the db backing file for the redis instance.
  - pid - The process id (pid) of the embedded redis server.
- If the db argument is passed and there is another redislite object using that db, it will create a new connection to that redislite instance.
- The redis server for a redislite object is shutdown and it's configuration is deleted when the last redislite connection to the server is terminated.
- If a redis rdb filename is specified, the cleanup will not delete the rdb file so it can be used again.

Using redislite with existing python code

The redislite patch functionality can be used to make many existing python modules that use redis to work with minimal modifications.

Celery

```
# settings.py

from redislite import Redis

# Create a Redis instance using redislite
REDIS_DB_PATH = os.path.join('/tmp/my_redis.db')
rdb = Redis(REDIS_DB_PATH)
REDIS_SOCKET_PATH = 'redis+socket://%s' % (rdb.socket_file, )

# Use redislite for the Celery broker
BROKER_URL = REDIS_SOCKET_PATH

# (Optionally) use redislite for the Celery result backend
CELERY_RESULT_BACKEND = REDIS_SOCKET_PATH
```

```
# your_celery_app.py

from celery import Celery

# for django projects
from django.conf import settings
```

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'settings')
celery_app = Celery('my_app')
celery_app.config_from_object('django.conf:settings')

# for other projects
celery_app = Celery('my_app')
celery_app.config_from_object('settings')
```

Note that only one Redis instance is created, and others merely discover the running instance and use its existing socket file path.

redis-collections

Use a redisliste.StrictRedis() object for the redis-collections redis keyword argument to use it with redislite. If the redis-collection variable needs to be persistent make sure to pass the redislite.StrictRedis() class a dbfilename argument to use.

```
>>> import redislite
>>> import redis_collections
>>> example_dict = redis_collections.Dict(redis=redislite.StrictRedis('example.rdb'))
>>> example_dict['test'] = 'This is a test variable'
>>> example_dict
<redis_collections.Dict at 7908fc2cc97d49fda4bce7365df3b373 {'test': 'This is a test␣
↪variable'}>
>>>
```

RQ

When using rq you will need to specify a db_filename for the connection.

To put jobs on queues, you don't have to do anything special, just define your typically lengthy or blocking function:

```
import requests

def count_words_at_url(url):
    resp = requests.get(url)
    return len(resp.text.split())
```

Then, create a RQ queue:

```
from redislite import Redis
from rq import Queue

q = Queue(connection=Redis('RQ_example.rdb'))

And enqueue the function call:

from my_module import count_words_at_url
result = q.enqueue(
             count_words_at_url, 'http://nvie.com')
```

For a more complete example, refer to the RQ docs. To start executing enqueued function calls in the background, start a worker from your project's directory:

```
$ rqworker
*** Listening for work on default
Got count_words_at_url('http://nvie.com') from default
Job result = 818
*** Listening for work on default
```

Walrus

First, install both walrus and redislite.

Install both modules:

```
$ pip install walrus redislite
```

Then patch redis before using walrus. Optionally specifying a redis db if the result needs to be usable after the script finishes running.

```
>>> from redislite.patch import patch_redis
>>> patch_redis('/tmp/walrus.db')
>>> from walrus import *
>>> db = Database()
>>> huey = db.Hash('huey')
>>> huey.update(color='white', temperament='ornery', type='kitty')
<Hash "huey": {'color': 'white', 'type': 'kitty', 'temperament': 'ornery'}>
>>> huey.keys()
['color', 'type', 'temperament']
>>> 'color' in huey
True
>>> huey['color']
'white'
```

Code Documentation

Module

This module provides access to a redis server using a redis server embedded in the module. It provides enhanced redis bindings that are able to configure run, and cleanup a redis server when they are accessed.

`redislite.__version__`
str – The version of the redislite module.

`redislite.__redis_executable__`
str – The full path to the embedded redis-server executable.

`redislite.__redis_server_version__`
str – The version of the embedded redis-server built intot he module.

`redislite.__git_source_url__`
str – The github web url for the source code used to generate this module. This will be an empty string if the module was not built from a github repo.

`redislite.__git_version__`
str – Version number derived from the number of git revisions. This will be an empty string if not built from a git repo.

redislite.**__git_origin__**
str – The git origin of the source repository the module was built from. This will be an empty string if not built from a git repo.

redislite.**__git_branch__**
str – The git branch the module was built from. This will be an empty string if not built from a git repo.

redislite.**__git_hash__**
str – The git hash value for the code used to build this module.

Example

To access redis using a newly installed and configured redis server, then set and retrieve some data:

```
>>> import redislite
>>> connection = redislite.Redis()
>>> connection.set('key', 'value')
True
>>> connection.get('key')
'value'
>>>
```

redislite.Redis() Class

class redislite.**Redis**(**args*, ***kwargs*)
Bases: redislite.client.RedisMixin, redis.client.Redis

This class provides an enhanced version of the redis.Redis() class that uses an embedded redis-server by default.

Parameters

- **dbfilename** (*str, optional*) – The name of the Redis db file to be used.

  This argument is only used if the embedded redis-server is used.

  The value of this argument is provided as the "dbfilename" setting in the embedded redis server configuration. This will result in the embedded redis server dumping it's database to this file on exit/close.

  This will also result in the embedded redis server using an existing redis rdb database if the file exists on start.

  If this file exists and is in use by another redislite instance, this class will get a reference to the existing running redis instance so both instances share the same redis-server process and don't corrupt the db file.

- **serverconfig** (*dict, optional*) – A dictionary of additional redis-server configuration settings. The key is the name of the setting in the configuration file, the values may be list, str, or None.

  If the value is a list the setting will be repeated in the configuration, once for each value.

  If the value is a string, the setting will occur once with that string as the setting.

  If the value is None, the setting will be removed from the default setting values if it exists in the defaults.

- **host** (*str, optional*) – The hostname or ip address of the redis server to connect to.

  If this argument is specified the embedded redis server will not be used.

- **port** (*int, optional*) – The port number of the redis server to connect to.

  If this argument is specified, the embedded redis server will not be used.

- ****kwargs** (*optional*) – All other keyword arguments supported by the `redis.Redis()` class are supported.

Returns

Return type A *redislite.Redis()* object

Raises `RedisLiteServerStartError` – The embedded Redis server failed to start

Example

redis_connection = `redislite.Redis('/tmp/redis.db')`

Notes

If the dbfilename argument is not provided each instance will get a different redis-server instance.

db
str – The fully qualified filename associated with the redis dbfilename configuration setting. This attribute is read only.

logfile
str – The name of the redis-server logfile

pid
int – Pid of the running embedded redis server, this attribute is read only.

redis_log
str – The contents of the redis-server log file

start_timeout
float – Number of seconds to wait for the redis-server process to start before generating a RedisLiteServerStartError exception.

append(*key*, *value*)
Appends the string `value` to the value at `key`. If `key` doesn't already exist, create it with a value of `value`. Returns the new length of the value at `key`.

bgrewriteaof()
Tell the Redis server to rewrite the AOF file from data in memory.

bgsave()
Tell the Redis server to save its data to disk. Unlike save(), this method is asynchronous and returns immediately.

bitcount(*key*, *start=None*, *end=None*)
Returns the count of set bits in the value of `key`. Optional `start` and `end` paramaters indicate which bytes to consider

bitop(*operation*, *dest*, **keys*)
Perform a bitwise operation using `operation` between `keys` and store the result in `dest`.

bitpos(*key*, *bit*, *start=None*, *end=None*)
Return the position of the first bit set to 1 or 0 in a string. `start` and `end` difines search range. The range is interpreted as a range of bytes and not a range of bits, so start=0 and end=2 means to look at the first three bytes.

blpop(*keys*, *timeout=0*)
LPOP a value off of the first non-empty list named in the `keys` list.

If none of the lists in `keys` has a value to LPOP, then block for `timeout` seconds, or until a value gets pushed on to one of the lists.

If timeout is 0, then block indefinitely.

brpop(*keys*, *timeout=0*)
RPOP a value off of the first non-empty list named in the `keys` list.

If none of the lists in `keys` has a value to LPOP, then block for `timeout` seconds, or until a value gets pushed on to one of the lists.

If timeout is 0, then block indefinitely.

brpoplpush(*src*, *dst*, *timeout=0*)
Pop a value off the tail of `src`, push it on the head of `dst` and then return it.

This command blocks until a value is in `src` or until `timeout` seconds elapse, whichever is first. A `timeout` value of 0 blocks forever.

client_getname()
Returns the current connection name

client_kill(*address*)
Disconnects the client at `address` (ip:port)

client_list()
Returns a list of currently connected clients

client_setname(*name*)
Sets the current connection name

config_get(*pattern='*'*)
Return a dictionary of configuration based on the `pattern`

config_resetstat()
Reset runtime statistics

config_rewrite()
Rewrite config file with the minimal change to reflect running config

config_set(*name*, *value*)
Set config item `name` with `value`

db
Return the connection string to allow connecting to the same redis server. :return: connection_path

dbsize()
Returns the number of keys in the current database

debug_object(*key*)
Returns version specific meta information about a given key

decr(*name*, *amount=1*)
Decrements the value of `key` by `amount`. If no key exists, the value will be initialized as 0 - `amount`

delete(**names*)
Delete one or more keys specified by `names`

dump(*name*)
Return a serialized version of the value stored at the specified key. If key does not exist a nil bulk reply is returned.

echo(*value*)
Echo the string back from the server

eval(*script*, *numkeys*, **keys_and_args*)
Execute the Lua `script`, specifying the `numkeys` the script will touch and the key names and argument values in `keys_and_args`. Returns the result of the script.

In practice, use the object returned by `register_script`. This function exists purely for Redis API completion.

evalsha(*sha*, *numkeys*, **keys_and_args*)
Use the `sha` to execute a Lua script already registered via EVAL or SCRIPT LOAD. Specify the `numkeys` the script will touch and the key names and argument values in `keys_and_args`. Returns the result of the script.

In practice, use the object returned by `register_script`. This function exists purely for Redis API completion.

execute_command(**args*, ***options*)
Execute a command and return a parsed response

exists(*name*)
Returns a boolean indicating whether key `name` exists

expire(*name*, *time*)
Set an expire flag on key `name` for `time` seconds. `time` can be represented by an integer or a Python timedelta object.

expireat(*name*, *when*)
Set an expire flag on key `name`. `when` can be represented as an integer indicating unix time or a Python datetime object.

flushall()
Delete all keys in all databases on the current host

flushdb()
Delete all keys in the current database

from_url(*url*, *db=None*, ***kwargs*)
Return a Redis client object configured from the given URL.

For example:

```
redis://[:password]@localhost:6379/0
unix://[:password]@/path/to/socket.sock?db=0
```

There are several ways to specify a database number. The parse function will return the first specified option:

1.A `db` querystring option, e.g. redis://localhost?db=0

2.If using the redis:// scheme, the path argument of the url, e.g. redis://localhost/0

3.The `db` argument to this function.

If none of these options are specified, db=0 is used.

Any additional querystring arguments and keyword arguments will be passed along to the ConnectionPool class's initializer. In the case of conflicting arguments, querystring arguments always win.

get(*name*)
Return the value at key `name`, or None if the key doesn't exist

getbit(*name*, *offset*)
Returns a boolean indicating the value of `offset` in `name`

getrange(*key*, *start*, *end*)
Returns the substring of the string value stored at `key`, determined by the offsets `start` and `end` (both are inclusive)

getset(*name*, *value*)
Sets the value at key `name` to `value` and returns the old value at key `name` atomically.

hdel(*name*, **keys*)
Delete `keys` from hash `name`

hexists(*name*, *key*)
Returns a boolean indicating if `key` exists within hash `name`

hget(*name*, *key*)
Return the value of `key` within the hash `name`

hgetall(*name*)
Return a Python dict of the hash's name/value pairs

hincrby(*name*, *key*, *amount=1*)
Increment the value of `key` in hash `name` by `amount`

hincrbyfloat(*name*, *key*, *amount=1.0*)
Increment the value of `key` in hash `name` by floating `amount`

hkeys(*name*)
Return the list of keys within hash `name`

hlen(*name*)
Return the number of elements in hash `name`

hmget(*name*, *keys*, **args*)
Returns a list of values ordered identically to `keys`

hmset(*name*, *mapping*)
Set key to value within hash `name` for each corresponding key and value from the `mapping` dict.

hscan(*name*, *cursor=0*, *match=None*, *count=None*)
Incrementally return key/value slices in a hash. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

hscan_iter(*name*, *match=None*, *count=None*)
Make an iterator using the HSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

hset(*name*, *key*, *value*)
Set `key` to `value` within hash `name` Returns 1 if HSET created a new field, otherwise 0

hsetnx(*name*, *key*, *value*)
Set `key` to `value` within hash `name` if `key` does not exist. Returns 1 if HSETNX created a field, otherwise 0.

hvals(*name*)
Return the list of values within hash `name`

incr(*name*, *amount=1*)
Increments the value of `key` by `amount`. If no key exists, the value will be initialized as `amount`

incrby(*name*, *amount=1*)
Increments the value of `key` by `amount`. If no key exists, the value will be initialized as `amount`

incrbyfloat(*name*, *amount=1.0*)
Increments the value at key `name` by floating `amount`. If no key exists, the value will be initialized as `amount`

info(*section=None*)
Returns a dictionary containing information about the Redis server

The `section` option can be used to select a specific section of information

The section option is not supported by older versions of Redis Server, and will generate ResponseError

keys(*pattern='*'*)
Returns a list of keys matching `pattern`

lastsave()
Return a Python datetime object representing the last time the Redis database was saved to disk

lindex(*name*, *index*)
Return the item from list `name` at position `index`

Negative indexes are supported and will return an item at the end of the list

linsert(*name*, *where*, *refvalue*, *value*)
Insert `value` in list `name` either immediately before or after [`where`] `refvalue`

Returns the new length of the list on success or -1 if `refvalue` is not in the list.

llen(*name*)
Return the length of the list `name`

lock(*name*, *timeout=None*, *sleep=0.1*, *blocking_timeout=None*, *lock_class=None*, *thread_local=True*)
Return a new Lock object using key `name` that mimics the behavior of threading.Lock.

If specified, `timeout` indicates a maximum life for the lock. By default, it will remain locked until release() is called.

`sleep` indicates the amount of time to sleep per loop iteration when the lock is in blocking mode and another client is currently holding the lock.

`blocking_timeout` indicates the maximum amount of time in seconds to spend trying to acquire the lock. A value of `None` indicates continue trying forever. `blocking_timeout` can be specified as a float or integer, both representing the number of seconds to wait.

`lock_class` forces the specified lock implementation.

`thread_local` indicates whether the lock token is placed in thread-local storage. By default, the token is placed in thread local storage so that a thread only sees its token, not a token set by another thread. Consider the following timeline:

time: 0, thread-1 acquires *my-lock*, with a timeout of 5 seconds. thread-1 sets the token to "abc"

time: 1, thread-2 blocks trying to acquire *my-lock* using the Lock instance.

time: 5, thread-1 has not yet completed. redis expires the lock key.

time: 5, thread-2 acquired *my-lock* now that it's available. thread-2 sets the token to "xyz"

time: 6, thread-1 finishes its work and calls release(). if the token is *not* stored in thread local storage, then thread-1 would see the token value as "xyz" and would be able to successfully release the thread-2's lock.

In some use cases it's necessary to disable thread local storage. For example, if you have code where one thread acquires a lock and passes that lock instance to a worker thread to release later. If thread local storage isn't disabled in this case, the worker thread won't see the token set by the thread that acquired the lock. Our assumption is that these cases aren't common and as such default to using thread local storage.

lpop(*name*)
Remove and return the first item of the list `name`

lpush(*name*, **values*)
Push `values` onto the head of the list `name`

lpushx(*name*, *value*)
Push `value` onto the head of the list `name` if `name` exists

lrange(*name*, *start*, *end*)
Return a slice of the list `name` between position `start` and `end`

`start` and `end` can be negative numbers just like Python slicing notation

lrem(*name*, *value*, *num=0*)
Remove the first `num` occurrences of elements equal to `value` from the list stored at `name`.

The `num` argument influences the operation in the following ways: num > 0: Remove elements equal to value moving from head to tail. num < 0: Remove elements equal to value moving from tail to head. num = 0: Remove all elements equal to value.

lset(*name*, *index*, *value*)
Set `position` of list `name` to `value`

ltrim(*name*, *start*, *end*)
Trim the list `name`, removing all values not within the slice between `start` and `end`

`start` and `end` can be negative numbers just like Python slicing notation

mget(*keys*, **args*)
Returns a list of values ordered identically to `keys`

move(*name*, *db*)
Moves the key `name` to a different Redis database `db`

mset(**args*, ***kwargs*)
Sets key/values based on a mapping. Mapping can be supplied as a single dictionary argument or as kwargs.

msetnx(**args*, ***kwargs*)
Sets key/values based on a mapping if none of the keys are already set. Mapping can be supplied as a single dictionary argument or as kwargs. Returns a boolean indicating if the operation was successful.

object(*infotype*, *key*)
Return the encoding, idletime, or refcount about the key

parse_response(*connection*, *command_name*, ***options*)
Parses a response from the Redis server

persist(*name*)
Removes an expiration on `name`

pexpire(*name*, *time*)
Set an expire flag on key `name` for `time` milliseconds. `time` can be represented by an integer or a Python timedelta object.

pexpireat(*name*, *when*)
Set an expire flag on key `name`. `when` can be represented as an integer representing unix time in milliseconds (unix time * 1000) or a Python datetime object.

pfadd(*name*, **values*)
Adds the specified elements to the specified HyperLogLog.

pfcount(**sources*)
Return the approximated cardinality of the set observed by the HyperLogLog at key(s).

pfmerge(*dest*, **sources*)
Merge N different HyperLogLogs into a single one.

pid
Get the current redis-server process id.

Returns The process id of the redis-server process associated with this redislite instance or None. If the redis-server is not running.

Return type *pid*(int)

ping()
Ping the Redis server

pipeline(*transaction=True*, *shard_hint=None*)
Return a new pipeline object that can queue multiple commands for later execution. `transaction` indicates whether all commands should be executed atomically. Apart from making a group of operations atomic, pipelines are useful for reducing the back-and-forth overhead between the client and server.

psetex(*name*, *time_ms*, *value*)
Set the value of key `name` to `value` that expires in `time_ms` milliseconds. `time_ms` can be represented by an integer or a Python timedelta object

pttl(*name*)
Returns the number of milliseconds until the key `name` will expire

publish(*channel*, *message*)
Publish `message` on `channel`. Returns the number of subscribers the message was delivered to.

pubsub(***kwargs*)
Return a Publish/Subscribe object. With this object, you can subscribe to channels and listen for messages that get published to them.

randomkey()
Returns the name of a random key

redis_log
Redis server log content as a string

Returns Log contents

Return type str

redis_log_tail(*lines=1*, *width=80*)
The redis log output

Parameters

- **lines** (*int, optional*) – Number of lines from the end of the logfile to return, a value of 0 will return all lines, default=1
- **width** (*int, optional*) – The expected average width of a log file line, this is used to determine the chunksize of the seek operations, default=80

Returns List of strings containing the lines from the logfile requested

Return type list

register_script(*script*)
Register a Lua `script` specifying the `keys` it will touch. Returns a Script object that is callable and hides the complexity of deal with scripts, keys, and shas. This is the preferred way to work with Lua scripts.

rename(*src*, *dst*)
Rename key `src` to `dst`

renamenx(*src*, *dst*)
Rename key `src` to `dst` if `dst` doesn't already exist

restore(*name*, *ttl*, *value*)
Create a key using the provided serialized value, previously obtained using DUMP.

rpop(*name*)
Remove and return the last item of the list `name`

rpoplpush(*src*, *dst*)
RPOP a value off of the `src` list and atomically LPUSH it on to the `dst` list. Returns the value.

rpush(*name*, **values*)
Push `values` onto the tail of the list `name`

rpushx(*name*, *value*)
Push `value` onto the tail of the list `name` if `name` exists

sadd(*name*, **values*)
Add `value(s)` to set `name`

save()
Tell the Redis server to save its data to disk, blocking until the save is complete

scan(*cursor=0*, *match=None*, *count=None*)
Incrementally return lists of key names. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

scan_iter(*match=None*, *count=None*)
Make an iterator using the SCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

scard(*name*)
Return the number of elements in set `name`

script_exists(**args*)
Check if a script exists in the script cache by specifying the SHAs of each script as `args`. Returns a list of boolean values indicating if if each already script exists in the cache.

script_flush()
Flush all scripts from the script cache

script_kill()
Kill the currently executing Lua script

script_load(*script*)
Load a Lua `script` into the script cache. Returns the SHA.

sdiff(*keys*, **args*)
Return the difference of sets specified by `keys`

sdiffstore(*dest*, *keys*, **args*)
Store the difference of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

sentinel(**args*)
Redis Sentinel's SENTINEL command.

sentinel_get_master_addr_by_name(*service_name*)
Returns a (host, port) pair for the given `service_name`

sentinel_master(*service_name*)
Returns a dictionary containing the specified masters state.

sentinel_masters()
Returns a list of dictionaries containing each master's state.

sentinel_monitor(*name*, *ip*, *port*, *quorum*)
Add a new master to Sentinel to be monitored

sentinel_remove(*name*)
Remove a master from Sentinel's monitoring

sentinel_sentinels(*service_name*)
Returns a list of sentinels for `service_name`

sentinel_set(*name*, *option*, *value*)
Set Sentinel monitoring parameters for a given master

sentinel_slaves(*service_name*)
Returns a list of slaves for `service_name`

set(*name*, *value*, *ex=None*, *px=None*, *nx=False*, *xx=False*)
Set the value at key `name` to `value`

`ex` sets an expire flag on key `name` for `ex` seconds.

`px` sets an expire flag on key `name` for `px` milliseconds.

nx if set to True, set the value at key `name` to `value` if it does not already exist.

xx if set to True, set the value at key `name` to `value` if it already exists.

set_response_callback(*command*, *callback*)
Set a custom Response Callback

setbit(*name*, *offset*, *value*)
Flag the `offset` in `name` as `value`. Returns a boolean indicating the previous value of `offset`.

setex(*name*, *value*, *time*)
Set the value of key `name` to `value` that expires in `time` seconds. `time` can be represented by an integer or a Python timedelta object.

setnx(*name*, *value*)
Set the value of key `name` to `value` if key doesn't exist

setrange(*name*, *offset*, *value*)
Overwrite bytes in the value of `name` starting at `offset` with `value`. If `offset` plus the length of `value` exceeds the length of the original value, the new value will be larger than before. If `offset` exceeds the length of the original value, null bytes will be used to pad between the end of the previous value and the start of what's being injected.

Returns the length of the new string.

shutdown()
Shutdown the server

sinter(*keys*, **args*)
Return the intersection of sets specified by `keys`

sinterstore(*dest*, *keys*, **args*)
Store the intersection of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

sismember(*name*, *value*)
Return a boolean indicating if `value` is a member of set `name`

slaveof(*host=None*, *port=None*)
Set the server to be a replicated slave of the instance identified by the `host` and `port`. If called without arguments, the instance is promoted to a master instead.

slowlog_get(*num=None*)
Get the entries from the slowlog. If `num` is specified, get the most recent `num` items.

slowlog_len()
Get the number of items in the slowlog

slowlog_reset()
Remove all items in the slowlog

smembers(*name*)
Return all members of the set `name`

smove(*src*, *dst*, *value*)
Move `value` from set `src` to set `dst` atomically

sort(*name*, *start=None*, *num=None*, *by=None*, *get=None*, *desc=False*, *alpha=False*, *store=None*, *groups=False*)
Sort and return the list, set or sorted set at `name`.

`start` and `num` allow for paging through the sorted data

by allows using an external key to weight and sort the items. Use an "*" to indicate where in the key the item value is located

get allows for returning items from external keys rather than the sorted data itself. Use an "*" to indicate where int he key the item value is located

`desc` allows for reversing the sort

`alpha` allows for sorting lexicographically rather than numerically

store allows for storing the result of the sort into the key `store`

groups if set to True and if get contains at least two elements, sort will return a list of tuples, each containing the values fetched from the arguments to `get`.

spop(*name*)
Remove and return a random member of set `name`

srandmember(*name*, *number=None*)
If `number` is None, returns a random member of set `name`.

If `number` is supplied, returns a list of `number` random memebers of set `name`. Note this is only available when running Redis 2.6+.

srem(*name*, **values*)
Remove `values` from set `name`

sscan(*name*, *cursor=0*, *match=None*, *count=None*)
Incrementally return lists of elements in a set. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

sscan_iter(*name*, *match=None*, *count=None*)
Make an iterator using the SSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

strlen(*name*)
Return the number of bytes stored in the value of `name`

substr(*name*, *start*, *end=-1*)
Return a substring of the string at key `name`. `start` and `end` are 0-based integers specifying the portion of the string to return.

sunion(*keys*, **args*)
Return the union of sets specified by `keys`

sunionstore(*dest*, *keys*, **args*)
Store the union of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

time()
Returns the server time as a 2-item tuple of ints: (seconds since epoch, microseconds into this second).

transaction(*func*, **watches*, ***kwargs*)
Convenience method for executing the callable *func* as a transaction while watching all keys specified in *watches*. The 'func' callable should expect a single argument which is a Pipeline object.

ttl(*name*)
Returns the number of seconds until the key `name` will expire

type(*name*)
Returns the type of key `name`

unwatch()
Unwatches the value at key `name`, or None of the key doesn't exist

wait(*num_replicas*, *timeout*)
Redis synchronous replication That returns the number of replicas that processed the query when we finally have at least `num_replicas`, or when the `timeout` was reached.

watch(**names*)
Watches the values at keys `names`, or None if the key doesn't exist

zadd(*name*, **args*, ***kwargs*)
NOTE: The order of arguments differs from that of the official ZADD command. For backwards compatability, this method accepts arguments in the form of name1, score1, name2, score2, while the official Redis documents expects score1, name1, score2, name2.

If you're looking to use the standard syntax, consider using the StrictRedis class. See the API Reference section of the docs for more information.

Set any number of element-name, score pairs to the key `name`. Pairs can be specified in two ways:

As *args, in the form of: name1, score1, name2, score2, ... or as **kwargs, in the form of: name1=score1, name2=score2, ...

The following example would add four values to the 'my-key' key: redis.zadd('my-key', 'name1', 1.1, 'name2', 2.2, name3=3.3, name4=4.4)

zcard(*name*)
Return the number of elements in the sorted set `name`

zcount(*name*, *min*, *max*)
Returns the number of elements in the sorted set at key `name` with a score between `min` and `max`.

zincrby(*name*, *value*, *amount=1*)
Increment the score of `value` in sorted set `name` by `amount`

zinterstore(*dest*, *keys*, *aggregate=None*)
Intersect multiple sorted sets specified by `keys` into a new sorted set, `dest`. Scores in the destination will be aggregated based on the `aggregate`, or SUM if none is provided.

zlexcount(*name*, *min*, *max*)
Return the number of items in the sorted set `name` between the lexicographical range `min` and `max`.

zrange(*name*, *start*, *end*, *desc=False*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from sorted set `name` between `start` and `end` sorted in ascending order.

`start` and `end` can be negative, indicating the end of the range.

`desc` a boolean indicating whether to sort the results descendingly

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

`score_cast_func` a callable used to cast the score return value

zrangebylex(*name*, *min*, *max*, *start=None*, *num=None*)
Return the lexicographical range of values from sorted set `name` between `min` and `max`.

If `start` and `num` are specified, then return a slice of the range.

zrangebyscore(*name*, *min*, *max*, *start=None*, *num=None*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from the sorted set `name` with scores between `min` and `max`.

If `start` and `num` are specified, then return a slice of the range.

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

score_cast_func' a callable used to cast the score return value

zrank(*name*, *value*)
Returns a 0-based value indicating the rank of `value` in sorted set `name`

zrem(*name*, **values*)
Remove member `values` from sorted set `name`

zremrangebylex (*name*, *min*, *max*)
Remove all elements in the sorted set `name` between the lexicographical range specified by `min` and `max`.

Returns the number of elements removed.

zremrangebyrank (*name*, *min*, *max*)
Remove all elements in the sorted set `name` with ranks between `min` and `max`. Values are 0-based, ordered from smallest score to largest. Values can be negative indicating the highest scores. Returns the number of elements removed

zremrangebyscore (*name*, *min*, *max*)
Remove all elements in the sorted set `name` with scores between `min` and `max`. Returns the number of elements removed.

zrevrange (*name*, *start*, *end*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from sorted set `name` between `start` and `end` sorted in descending order.

`start` and `end` can be negative, indicating the end of the range.

`withscores` indicates to return the scores along with the values The return type is a list of (value, score) pairs

`score_cast_func` a callable used to cast the score return value

zrevrangebylex (*name*, *max*, *min*, *start=None*, *num=None*)
Return the reversed lexicographical range of values from sorted set `name` between `max` and `min`.

If `start` and `num` are specified, then return a slice of the range.

zrevrangebyscore (*name*, *max*, *min*, *start=None*, *num=None*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from the sorted set `name` with scores between `min` and `max` in descending order.

If `start` and `num` are specified, then return a slice of the range.

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

`score_cast_func` a callable used to cast the score return value

zrevrank (*name*, *value*)
Returns a 0-based value indicating the descending rank of `value` in sorted set `name`

zscan (*name*, *cursor=0*, *match=None*, *count=None*, *score_cast_func=<type 'float'>*)
Incrementally return lists of elements in a sorted set. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

`score_cast_func` a callable used to cast the score return value

zscan_iter (*name*, *match=None*, *count=None*, *score_cast_func=<type 'float'>*)
Make an iterator using the ZSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

`score_cast_func` a callable used to cast the score return value

zscore (*name*, *value*)
Return the score of element `value` in sorted set `name`

zunionstore(*dest*, *keys*, *aggregate=None*)
Union multiple sorted sets specified by `keys` into a new sorted set, `dest`. Scores in the destination will be aggregated based on the `aggregate`, or SUM if none is provided.

redislite.StrictRedis() Class

class `redislite.`**`StrictRedis`**(**args*, ***kwargs*)
Bases: `redislite.client.RedisMixin`, `redis.client.StrictRedis`

This class provides an enhanced version of the `redis.StrictRedis()` class that uses an embedded redis-server by default.

Example

redis_connection = `redislite.StrictRedis('/tmp/redis.db')`

Notes

If the dbfilename argument is not provided each instance will get a different redis-server instance.

Parameters **dbfilename** (*str*) – The name of the Redis db file to be used. This argument is only used if the embedded redis-server is used. The value of this argument is provided as the "dbfilename" setting in the embedded redis server configuration. This will result in the embedded redis server dumping it's database to this file on exit/close. This will also result in the embedded redis server using an existing redis database if the file exists on start. If this file exists and is in use by another redislite instance, this class will get a reference to the existing running redis instance so both instances share the same redis-server process and don't corrupt the db file.

Kwargs:

host(str): The hostname or ip address of the redis server to connect to. If this argument is not None, the embedded redis server will not be used. Defaults to None.

port(int): The port number of the redis server to connect to. If this argument is not None, the embedded redis server will not be used. Defaults to None.

serverconfig(dict): A dictionary of additional redis-server configuration settings. All keys and values must be str. Supported keys are:

activerehashing, aof_rewrite_incremental_fsync, appendfilename, appendfsync, appendonly, auto_aof_rewrite_min_size, auto_aof_rewrite_percentage, aof_load_truncated, databases, hash_max_ziplist_entries, hash_max_ziplist_value, hll_sparse_max_bytes, hz, latency_monitor_threshold, list_max_ziplist_entries, list_max_ziplist_value, logfile, loglevel, lua_time_limit, no_appendfsync_on_rewrite, notify_keyspace_events, port, rdbchecksum, rdbcompression, repl_disable_tcp_nodelay, slave_read_only, slave_serve_stale_data, stop_writes_on_bgsave_error, tcp_backlog, tcp_keepalive, unixsocket, unixsocketperm, slave_priority, timeout, set_max_intset_entries, zset_max_ziplist_entries, zset_max_ziplist_value

Returns A `redis.StrictRedis()` class object if the host or port arguments where set or a *redislite.StrictRedis()* object otherwise.

Raises `RedisLiteServerStartError`

db
string – The fully qualified filename associated with the redis dbfilename configuration setting. This attribute is read only.

pid
int – Pid of the running embedded redis server, this attribute is read only.

start_timeout
float – Number of seconds to wait for the redis-server process to start before generating a RedisLiteServerStartError exception.

append(*key*, *value*)
Appends the string `value` to the value at `key`. If `key` doesn't already exist, create it with a value of `value`. Returns the new length of the value at `key`.

bgrewriteaof()
Tell the Redis server to rewrite the AOF file from data in memory.

bgsave()
Tell the Redis server to save its data to disk. Unlike save(), this method is asynchronous and returns immediately.

bitcount(*key*, *start=None*, *end=None*)
Returns the count of set bits in the value of `key`. Optional `start` and `end` paramaters indicate which bytes to consider

bitop(*operation*, *dest*, **keys*)
Perform a bitwise operation using `operation` between `keys` and store the result in `dest`.

bitpos(*key*, *bit*, *start=None*, *end=None*)
Return the position of the first bit set to 1 or 0 in a string. `start` and `end` difines search range. The range is interpreted as a range of bytes and not a range of bits, so start=0 and end=2 means to look at the first three bytes.

blpop(*keys*, *timeout=0*)
LPOP a value off of the first non-empty list named in the `keys` list.

If none of the lists in `keys` has a value to LPOP, then block for `timeout` seconds, or until a value gets pushed on to one of the lists.

If timeout is 0, then block indefinitely.

brpop(*keys*, *timeout=0*)
RPOP a value off of the first non-empty list named in the `keys` list.

If none of the lists in `keys` has a value to LPOP, then block for `timeout` seconds, or until a value gets pushed on to one of the lists.

If timeout is 0, then block indefinitely.

brpoplpush(*src*, *dst*, *timeout=0*)
Pop a value off the tail of `src`, push it on the head of `dst` and then return it.

This command blocks until a value is in `src` or until `timeout` seconds elapse, whichever is first. A `timeout` value of 0 blocks forever.

client_getname()
Returns the current connection name

client_kill(*address*)
Disconnects the client at `address` (ip:port)

client_list()
Returns a list of currently connected clients

client_setname(*name*)
Sets the current connection name

config_get(*pattern='*'*)
Return a dictionary of configuration based on the `pattern`

config_resetstat()
Reset runtime statistics

config_rewrite()
Rewrite config file with the minimal change to reflect running config

config_set(*name*, *value*)
Set config item `name` with `value`

db
Return the connection string to allow connecting to the same redis server. :return: connection_path

dbsize()
Returns the number of keys in the current database

debug_object(*key*)
Returns version specific meta information about a given key

decr(*name*, *amount=1*)
Decrements the value of `key` by `amount`. If no key exists, the value will be initialized as 0 - `amount`

delete(**names*)
Delete one or more keys specified by `names`

dump(*name*)
Return a serialized version of the value stored at the specified key. If key does not exist a nil bulk reply is returned.

echo(*value*)
Echo the string back from the server

eval(*script*, *numkeys*, **keys_and_args*)
Execute the Lua `script`, specifying the `numkeys` the script will touch and the key names and argument values in `keys_and_args`. Returns the result of the script.

In practice, use the object returned by `register_script`. This function exists purely for Redis API completion.

evalsha(*sha*, *numkeys*, **keys_and_args*)
Use the `sha` to execute a Lua script already registered via EVAL or SCRIPT LOAD. Specify the `numkeys` the script will touch and the key names and argument values in `keys_and_args`. Returns the result of the script.

In practice, use the object returned by `register_script`. This function exists purely for Redis API completion.

execute_command(**args*, ***options*)
Execute a command and return a parsed response

exists(*name*)
Returns a boolean indicating whether key `name` exists

expire(*name*, *time*)
Set an expire flag on key `name` for `time` seconds. `time` can be represented by an integer or a Python timedelta object.

expireat(*name*, *when*)
Set an expire flag on key `name`. `when` can be represented as an integer indicating unix time or a Python datetime object.

flushall()
Delete all keys in all databases on the current host

flushdb()
Delete all keys in the current database

from_url(*url*, *db=None*, ***kwargs*)
Return a Redis client object configured from the given URL.

For example:

```
redis://[:password]@localhost:6379/0
unix://[:password]@/path/to/socket.sock?db=0
```

There are several ways to specify a database number. The parse function will return the first specified option:

1. A `db` querystring option, e.g. redis://localhost?db=0
2. If using the redis:// scheme, the path argument of the url, e.g. redis://localhost/0
3. The `db` argument to this function.

If none of these options are specified, db=0 is used.

Any additional querystring arguments and keyword arguments will be passed along to the ConnectionPool class's initializer. In the case of conflicting arguments, querystring arguments always win.

get(*name*)
Return the value at key `name`, or None if the key doesn't exist

getbit(*name*, *offset*)
Returns a boolean indicating the value of `offset` in `name`

getrange(*key*, *start*, *end*)
Returns the substring of the string value stored at `key`, determined by the offsets `start` and `end` (both are inclusive)

getset(*name*, *value*)
Sets the value at key `name` to `value` and returns the old value at key `name` atomically.

hdel(*name*, **keys*)
Delete `keys` from hash `name`

hexists(*name*, *key*)
Returns a boolean indicating if `key` exists within hash `name`

hget(*name*, *key*)
Return the value of `key` within the hash `name`

hgetall(*name*)
Return a Python dict of the hash's name/value pairs

hincrby(*name*, *key*, *amount=1*)
Increment the value of `key` in hash `name` by `amount`

hincrbyfloat(*name*, *key*, *amount=1.0*)
Increment the value of `key` in hash `name` by floating `amount`

hkeys(*name*)
Return the list of keys within hash `name`

hlen(*name*)
Return the number of elements in hash `name`

hmget(*name*, *keys*, **args*)
Returns a list of values ordered identically to `keys`

hmset(*name*, *mapping*)
Set key to value within hash `name` for each corresponding key and value from the `mapping` dict.

hscan(*name*, *cursor=0*, *match=None*, *count=None*)
Incrementally return key/value slices in a hash. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

hscan_iter(*name*, *match=None*, *count=None*)
Make an iterator using the HSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

hset(*name*, *key*, *value*)
Set `key` to `value` within hash `name` Returns 1 if HSET created a new field, otherwise 0

hsetnx(*name*, *key*, *value*)
Set `key` to `value` within hash `name` if `key` does not exist. Returns 1 if HSETNX created a field, otherwise 0.

hvals(*name*)
Return the list of values within hash `name`

incr(*name*, *amount=1*)
Increments the value of `key` by `amount`. If no key exists, the value will be initialized as `amount`

incrby(*name*, *amount=1*)
Increments the value of `key` by `amount`. If no key exists, the value will be initialized as `amount`

incrbyfloat(*name*, *amount=1.0*)
Increments the value at key `name` by floating `amount`. If no key exists, the value will be initialized as `amount`

info(*section=None*)
Returns a dictionary containing information about the Redis server

The `section` option can be used to select a specific section of information

The section option is not supported by older versions of Redis Server, and will generate ResponseError

keys(*pattern='*'*)
Returns a list of keys matching `pattern`

lastsave()
Return a Python datetime object representing the last time the Redis database was saved to disk

lindex(*name*, *index*)
Return the item from list `name` at position `index`

Negative indexes are supported and will return an item at the end of the list

linsert(*name*, *where*, *refvalue*, *value*)
Insert `value` in list `name` either immediately before or after [`where`] `refvalue`

Returns the new length of the list on success or -1 if `refvalue` is not in the list.

llen(*name*)
Return the length of the list `name`

lock(*name*, *timeout=None*, *sleep=0.1*, *blocking_timeout=None*, *lock_class=None*, *thread_local=True*)
Return a new Lock object using key `name` that mimics the behavior of threading.Lock.

If specified, `timeout` indicates a maximum life for the lock. By default, it will remain locked until release() is called.

`sleep` indicates the amount of time to sleep per loop iteration when the lock is in blocking mode and another client is currently holding the lock.

`blocking_timeout` indicates the maximum amount of time in seconds to spend trying to acquire the lock. A value of `None` indicates continue trying forever. `blocking_timeout` can be specified as a float or integer, both representing the number of seconds to wait.

`lock_class` forces the specified lock implementation.

`thread_local` indicates whether the lock token is placed in thread-local storage. By default, the token is placed in thread local storage so that a thread only sees its token, not a token set by another thread. Consider the following timeline:

time: 0, thread-1 acquires *my-lock*, with a timeout of 5 seconds. thread-1 sets the token to "abc"

time: 1, thread-2 blocks trying to acquire *my-lock* using the Lock instance.

time: 5, thread-1 has not yet completed. redis expires the lock key.

time: 5, thread-2 acquired *my-lock* now that it's available. thread-2 sets the token to "xyz"

time: 6, thread-1 finishes its work and calls release(). if the token is *not* stored in thread local storage, then thread-1 would see the token value as "xyz" and would be able to successfully release the thread-2's lock.

In some use cases it's necessary to disable thread local storage. For example, if you have code where one thread acquires a lock and passes that lock instance to a worker thread to release later. If thread local storage isn't disabled in this case, the worker thread won't see the token set by the thread that acquired the lock. Our assumption is that these cases aren't common and as such default to using thread local storage.

lpop(*name*)
Remove and return the first item of the list `name`

lpush(*name*, **values*)
Push `values` onto the head of the list `name`

lpushx(*name*, *value*)
Push `value` onto the head of the list `name` if `name` exists

lrange(*name*, *start*, *end*)
Return a slice of the list `name` between position `start` and `end`

`start` and `end` can be negative numbers just like Python slicing notation

lrem(*name*, *count*, *value*)
Remove the first `count` occurrences of elements equal to `value` from the list stored at `name`.

The count argument influences the operation in the following ways: count > 0: Remove elements equal to value moving from head to tail. count < 0: Remove elements equal to value moving from tail to head. count = 0: Remove all elements equal to value.

lset(*name*, *index*, *value*)
Set `position` of list `name` to `value`

ltrim(*name*, *start*, *end*)
Trim the list `name`, removing all values not within the slice between `start` and `end`

`start` and `end` can be negative numbers just like Python slicing notation

mget(*keys*, **args*)
Returns a list of values ordered identically to `keys`

move(*name*, *db*)
Moves the key `name` to a different Redis database `db`

mset(**args*, ***kwargs*)
Sets key/values based on a mapping. Mapping can be supplied as a single dictionary argument or as kwargs.

msetnx(**args*, ***kwargs*)
Sets key/values based on a mapping if none of the keys are already set. Mapping can be supplied as a single dictionary argument or as kwargs. Returns a boolean indicating if the operation was successful.

object(*infotype*, *key*)
Return the encoding, idletime, or refcount about the key

parse_response(*connection*, *command_name*, ***options*)
Parses a response from the Redis server

persist(*name*)
Removes an expiration on `name`

pexpire(*name*, *time*)
Set an expire flag on key `name` for `time` milliseconds. `time` can be represented by an integer or a Python timedelta object.

pexpireat(*name*, *when*)
Set an expire flag on key `name`. `when` can be represented as an integer representing unix time in milliseconds (unix time * 1000) or a Python datetime object.

pfadd(*name*, **values*)
Adds the specified elements to the specified HyperLogLog.

pfcount(**sources*)
Return the approximated cardinality of the set observed by the HyperLogLog at key(s).

pfmerge(*dest*, **sources*)
Merge N different HyperLogLogs into a single one.

pid
Get the current redis-server process id.

Returns The process id of the redis-server process associated with this redislite instance or None. If the redis-server is not running.

Return type *pid*(int)

ping()
Ping the Redis server

pipeline(*transaction=True*, *shard_hint=None*)
Return a new pipeline object that can queue multiple commands for later execution. `transaction` indicates whether all commands should be executed atomically. Apart from making a group of operations atomic, pipelines are useful for reducing the back-and-forth overhead between the client and server.

psetex(*name*, *time_ms*, *value*)
Set the value of key `name` to `value` that expires in `time_ms` milliseconds. `time_ms` can be represented by an integer or a Python timedelta object

pttl(*name*)
Returns the number of milliseconds until the key `name` will expire

publish(*channel*, *message*)
Publish `message` on `channel`. Returns the number of subscribers the message was delivered to.

pubsub(***kwargs*)
Return a Publish/Subscribe object. With this object, you can subscribe to channels and listen for messages that get published to them.

randomkey()
Returns the name of a random key

redis_log
Redis server log content as a string

Returns Log contents

Return type str

redis_log_tail(*lines=1*, *width=80*)
The redis log output

Parameters

- **lines** (*int, optional*) – Number of lines from the end of the logfile to return, a value of 0 will return all lines, default=1
- **width** (*int, optional*) – The expected average width of a log file line, this is used to determine the chunksize of the seek operations, default=80

Returns List of strings containing the lines from the logfile requested

Return type list

register_script(*script*)
Register a Lua `script` specifying the `keys` it will touch. Returns a Script object that is callable and hides the complexity of deal with scripts, keys, and shas. This is the preferred way to work with Lua scripts.

rename(*src*, *dst*)
Rename key `src` to `dst`

renamenx(*src*, *dst*)
Rename key `src` to `dst` if `dst` doesn't already exist

restore(*name*, *ttl*, *value*)
Create a key using the provided serialized value, previously obtained using DUMP.

rpop(*name*)
Remove and return the last item of the list `name`

rpoplpush(*src*, *dst*)
RPOP a value off of the `src` list and atomically LPUSH it on to the `dst` list. Returns the value.

rpush(*name*, **values*)
Push `values` onto the tail of the list `name`

rpushx(*name*, *value*)
Push `value` onto the tail of the list `name` if `name` exists

sadd(*name*, **values*)
Add `value(s)` to set `name`

save()
Tell the Redis server to save its data to disk, blocking until the save is complete

scan(*cursor=0*, *match=None*, *count=None*)
Incrementally return lists of key names. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

scan_iter(*match=None*, *count=None*)
Make an iterator using the SCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

scard(*name*)
Return the number of elements in set `name`

script_exists(**args*)
Check if a script exists in the script cache by specifying the SHAs of each script as `args`. Returns a list of boolean values indicating if if each already script exists in the cache.

script_flush()
Flush all scripts from the script cache

script_kill()
Kill the currently executing Lua script

script_load(*script*)
Load a Lua `script` into the script cache. Returns the SHA.

sdiff(*keys*, **args*)
Return the difference of sets specified by `keys`

sdiffstore(*dest*, *keys*, **args*)
Store the difference of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

sentinel(**args*)
Redis Sentinel's SENTINEL command.

sentinel_get_master_addr_by_name(*service_name*)
Returns a (host, port) pair for the given `service_name`

sentinel_master(*service_name*)
Returns a dictionary containing the specified masters state.

sentinel_masters()
Returns a list of dictionaries containing each master's state.

sentinel_monitor(*name*, *ip*, *port*, *quorum*)
Add a new master to Sentinel to be monitored

sentinel_remove(*name*)
Remove a master from Sentinel's monitoring

sentinel_sentinels(*service_name*)
Returns a list of sentinels for `service_name`

sentinel_set(*name*, *option*, *value*)
Set Sentinel monitoring parameters for a given master

sentinel_slaves(*service_name*)
Returns a list of slaves for `service_name`

set(*name*, *value*, *ex=None*, *px=None*, *nx=False*, *xx=False*)
Set the value at key `name` to `value`

`ex` sets an expire flag on key `name` for `ex` seconds.

`px` sets an expire flag on key `name` for `px` milliseconds.

nx if set to True, set the value at key `name` to `value` if it does not already exist.

xx if set to True, set the value at key `name` to `value` if it already exists.

set_response_callback(*command*, *callback*)
Set a custom Response Callback

setbit(*name*, *offset*, *value*)
Flag the `offset` in `name` as `value`. Returns a boolean indicating the previous value of `offset`.

setex(*name*, *time*, *value*)
Set the value of key `name` to `value` that expires in `time` seconds. `time` can be represented by an integer or a Python timedelta object.

setnx(*name*, *value*)
Set the value of key `name` to `value` if key doesn't exist

setrange(*name*, *offset*, *value*)
Overwrite bytes in the value of `name` starting at `offset` with `value`. If `offset` plus the length of `value` exceeds the length of the original value, the new value will be larger than before. If `offset` exceeds the length of the original value, null bytes will be used to pad between the end of the previous value and the start of what's being injected.

Returns the length of the new string.

shutdown()
Shutdown the server

sinter(*keys*, **args*)
Return the intersection of sets specified by `keys`

sinterstore(*dest*, *keys*, **args*)
Store the intersection of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

sismember(*name*, *value*)
Return a boolean indicating if `value` is a member of set `name`

slaveof(*host=None*, *port=None*)
Set the server to be a replicated slave of the instance identified by the `host` and `port`. If called without arguments, the instance is promoted to a master instead.

slowlog_get(*num=None*)
Get the entries from the slowlog. If `num` is specified, get the most recent `num` items.

slowlog_len()
Get the number of items in the slowlog

slowlog_reset()
Remove all items in the slowlog

smembers(*name*)
Return all members of the set `name`

smove(*src*, *dst*, *value*)
Move `value` from set `src` to set `dst` atomically

sort(*name*, *start=None*, *num=None*, *by=None*, *get=None*, *desc=False*, *alpha=False*, *store=None*, *groups=False*)
Sort and return the list, set or sorted set at `name`.

`start` and `num` allow for paging through the sorted data

by allows using an external key to weight and sort the items. Use an "*" to indicate where in the key the item value is located

get allows for returning items from external keys rather than the sorted data itself. Use an "*" to indicate where int he key the item value is located

`desc` allows for reversing the sort

`alpha` allows for sorting lexicographically rather than numerically

store allows for storing the result of the sort into the key `store`

groups if set to True and if get contains at least two elements, sort will return a list of tuples, each containing the values fetched from the arguments to `get`.

spop(*name*)
Remove and return a random member of set `name`

srandmember(*name*, *number=None*)
If `number` is None, returns a random member of set `name`.

If `number` is supplied, returns a list of `number` random memebers of set `name`. Note this is only available when running Redis 2.6+.

srem(*name*, **values*)
Remove `values` from set `name`

sscan(*name*, *cursor=0*, *match=None*, *count=None*)
Incrementally return lists of elements in a set. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

sscan_iter(*name*, *match=None*, *count=None*)
Make an iterator using the SSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

strlen(*name*)
Return the number of bytes stored in the value of `name`

substr(*name*, *start*, *end=-1*)
Return a substring of the string at key `name`. `start` and `end` are 0-based integers specifying the portion of the string to return.

sunion(*keys*, **args*)
Return the union of sets specified by `keys`

sunionstore(*dest*, *keys*, **args*)
Store the union of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

time()
Returns the server time as a 2-item tuple of ints: (seconds since epoch, microseconds into this second).

transaction(*func*, **watches*, ***kwargs*)
Convenience method for executing the callable *func* as a transaction while watching all keys specified in *watches*. The 'func' callable should expect a single argument which is a Pipeline object.

ttl(*name*)
Returns the number of seconds until the key `name` will expire

type(*name*)
Returns the type of key `name`

unwatch()
Unwatches the value at key `name`, or None of the key doesn't exist

wait(*num_replicas*, *timeout*)
Redis synchronous replication That returns the number of replicas that processed the query when we finally have at least `num_replicas`, or when the `timeout` was reached.

watch(**names*)
Watches the values at keys `names`, or None if the key doesn't exist

zadd(*name*, **args*, ***kwargs*)
Set any number of score, element-name pairs to the key `name`. Pairs can be specified in two ways:

As *args, in the form of: score1, name1, score2, name2, ... or as **kwargs, in the form of: name1=score1, name2=score2, ...

The following example would add four values to the 'my-key' key: redis.zadd('my-key', 1.1, 'name1', 2.2, 'name2', name3=3.3, name4=4.4)

zcard(*name*)
Return the number of elements in the sorted set `name`

zcount(*name*, *min*, *max*)
Returns the number of elements in the sorted set at key `name` with a score between `min` and `max`.

zincrby(*name*, *value*, *amount=1*)
Increment the score of `value` in sorted set `name` by `amount`

zinterstore(*dest*, *keys*, *aggregate=None*)
Intersect multiple sorted sets specified by `keys` into a new sorted set, `dest`. Scores in the destination will be aggregated based on the `aggregate`, or SUM if none is provided.

zlexcount(*name*, *min*, *max*)
Return the number of items in the sorted set `name` between the lexicographical range `min` and `max`.

zrange(*name*, *start*, *end*, *desc=False*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from sorted set `name` between `start` and `end` sorted in ascending order.

`start` and `end` can be negative, indicating the end of the range.

`desc` a boolean indicating whether to sort the results descendingly

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

`score_cast_func` a callable used to cast the score return value

zrangebylex(*name*, *min*, *max*, *start=None*, *num=None*)
Return the lexicographical range of values from sorted set `name` between `min` and `max`.

If `start` and `num` are specified, then return a slice of the range.

zrangebyscore(*name*, *min*, *max*, *start=None*, *num=None*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from the sorted set `name` with scores between `min` and `max`.

If `start` and `num` are specified, then return a slice of the range.

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

score_cast_func‘ a callable used to cast the score return value

zrank(*name*, *value*)
Returns a 0-based value indicating the rank of `value` in sorted set `name`

zrem(*name*, **values*)
Remove member `values` from sorted set `name`

zremrangebylex(*name*, *min*, *max*)
Remove all elements in the sorted set `name` between the lexicographical range specified by `min` and `max`.

Returns the number of elements removed.

zremrangebyrank(*name*, *min*, *max*)
Remove all elements in the sorted set `name` with ranks between `min` and `max`. Values are 0-based, ordered from smallest score to largest. Values can be negative indicating the highest scores. Returns the number of elements removed

zremrangebyscore(*name*, *min*, *max*)
Remove all elements in the sorted set `name` with scores between `min` and `max`. Returns the number of elements removed.

zrevrange(*name*, *start*, *end*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from sorted set `name` between `start` and `end` sorted in descending order.

`start` and `end` can be negative, indicating the end of the range.

`withscores` indicates to return the scores along with the values The return type is a list of (value, score) pairs

`score_cast_func` a callable used to cast the score return value

zrevrangebylex(*name*, *max*, *min*, *start=None*, *num=None*)
Return the reversed lexicographical range of values from sorted set `name` between `max` and `min`.

If `start` and `num` are specified, then return a slice of the range.

zrevrangebyscore(*name*, *max*, *min*, *start=None*, *num=None*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from the sorted set `name` with scores between `min` and `max` in descending order.

If `start` and `num` are specified, then return a slice of the range.

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

score_cast_func a callable used to cast the score return value

zrevrank(*name*, *value*)
Returns a 0-based value indicating the descending rank of value in sorted set name

zscan(*name*, *cursor=0*, *match=None*, *count=None*, *score_cast_func=<type 'float'>*)
Incrementally return lists of elements in a sorted set. Also return a cursor indicating the scan position.

match allows for filtering the keys by pattern

count allows for hint the minimum number of returns

score_cast_func a callable used to cast the score return value

zscan_iter(*name*, *match=None*, *count=None*, *score_cast_func=<type 'float'>*)
Make an iterator using the ZSCAN command so that the client doesn't need to remember the cursor position.

match allows for filtering the keys by pattern

count allows for hint the minimum number of returns

score_cast_func a callable used to cast the score return value

zscore(*name*, *value*)
Return the score of element value in sorted set name

zunionstore(*dest*, *keys*, *aggregate=None*)
Union multiple sorted sets specified by keys into a new sorted set, dest. Scores in the destination will be aggregated based on the aggregate, or SUM if none is provided.

Functions to patch the redis module

Functions to replace (monkeypatch) the redis module classes with redislite classes.

redislite.patch.**patch_redis**(*dbfile=None*)
Patch all the redis classes provided by redislite that have been patched.

Example

patch_redis('/tmp/redis.db')

Notes

If the dbfile parameter is not passed, each any instances of redis.StrictRedis() class with no arguments will get a unique instance of the redis server. If the dbfile parameter is provided, all instances of redis.Redis() will share/reference the same instance of the redis server.

Parameters **dbfile** (*str*) – The name of the Redis db file to be used. If this argument is passed all instances of the redis.Redis() class will share a single instance of the embedded redis server.

Returns This function does not return any values.

redislite.patch.**patch_redis_Redis**(*dbfile=None*)
This class patches the redis module to replace the redis.Redis() class with the redislite enhanced *redislite.Redis()* class that uses the embedded redis server.

Example

patch_redis_Redis('/tmp/redis.db')

Notes

If the dbfile parameter is not passed, each instance of the redis.Redis() class with no arguments will get a separate redis server. If the dbfile parameter is provided, all instances of the redis.Redis() class without a host or path argument will share/reference the same redis server.

Parameters **dbfile** (*str*) – The name of the Redis db file to be used. If this argument is passed all instances of the `redis.Redis` class will share a single embedded redis server.

Returns This function does not return any values.

`redislite.patch.`**`patch_redis_StrictRedis`**(*dbfile=None*)

This class patches the redis module to replace the :param dbfile: `redis.StrictRedis()` class with the redislite enhanced *`redislite.StrictRedis()`* class that uses the embedded redis server.

Example

patch_redis_StrictRedis('/tmp/redis.db')

Notes

If the dbfile parameter is not passed, all `redis.StrictRedis()` class with no arguments will get a separate redis server. If the dbfile parameter is provided, all instances of the `redis.Redis()` class passed with the same dbfile value will share/reference the same redis server.

Parameters **dbfile** (*str*) – The name of the Redis db file to be used. If this argument is passed all instances of the `redis.Redis` class will share a single instance of the embedded redis server.

Returns This function does not return any values.

`redislite.patch.`**`unpatch_redis`**`()`

Unpatch all the redis classes provided by *`redislite`* that have been patched.

Example

unpatch_redis()

`redislite.patch.`**`unpatch_redis_Redis`**`()`

This class unpatches the `redis.Redis()` class of the redis module and restores the original `redis.Redis()` class.

Example

unpatch_redis_Redis()

Returns This function does not return any values.

redislite.patch.**unpatch_redis_StrictRedis**()
This class unpatches the redis.StrictRedis() class of the redis module and restores the original redis.StrictRedis() class.

Example

unpatch_redis_StrictRedis()

Returns This function does not return any values.

Functions for troubleshooting

Redislite Debug Utilities

This module contains utility functions useful for troubleshooting redislite.

This module can be run from the command line using the following command:

```
python -m redislite.debug
```

This will output information like the following:

```
$ python -m redislite.debug
Redislite debug information:
    Version: 1.0.171
    Module Path: /tmp/redislite/lib/python3.4/site-packages/redislite

    Installed Redis Server:
        Redis Executable: /tmp/redislite/lib/python3.4/site-packages/redislite/bin/
↪redis-server
        build = 3a2b5dab9c14cd5e
        sha = 4657e47d:1
        bits = 64
        v = 2.8.17
        malloc = libc

    Found redis-server: /tmp/redislite/lib/python3.4/site-packages/redislite/bin/
↪redis-server
        v = 2.8.17
        sha = 4657e47d:1
        malloc = libc
        bits = 64
        build = 3a2b5dab9c14cd5e

    Source Code Information
        Git Source URL: https://github.com/yahoo/redislite/tree/
↪2ebd1b4d9c9ad41c78e8048fda3c69d2917c0348
        Git Hash: 2ebd1b4d9c9ad41c78e8048fda3c69d2917c0348
        Git Version: 1.0.171
        Git Origin: https://github.com/yahoo/redislite.git
        Git Branch: master
```

When run from the command line this will print a dump of information about the module and it's build information.

redislite.debug.**debug_info**()
Return a multi-line string with the debug information :return:

redislite.debug.**debug_info_list**()
Return a list with the debug information :return:

redislite.debug.**print_debug_info**()
Display information about the redislite build, and redis-server on stdout. :return:

Using the Redis Server directly

The *redis-server* application that is built by *redislite* during it's installation is installed into the scripts directory during the installation. This binary is a complete redis server and can be used independent of the *redislite* module.

Since redis-lite installs an actual redis-server it is possible to use the *redislite* *redis-server* binary directly and use more complex configurations than those created automatically by the *redislite.Redis()* and *redislite.StrictRedis()* classes.

It is also possible to start the *redis-server* process with no arguments and use it with the unpatched `redis` module.

See the Redis documentation for details about how to configure and run the *redis-server* directly.

Contributing to redislite

First steps

The redislite project always needs more people to help others. As soon as you learn redislite, you can contribute in many ways:

- Join the pythonredislite group or the pythonreddislite mailing list and answer questions.
- Join the #redislite IRC channel on Freenode and answer questions. By explaining redislite to other users, you're going to learn a lot about the module yourself.
- Report an issue or find a bug to fix on our issue tracker on github.
- Improve the documentation.
- Write unit tests.

Sign the Contributor License Agreement

Prior to submitting a pull request, please complete a Yahoo CLA Agreement.

Set up Development Environment

All redislite development uses the python *tox* tool.

Install it with the python pip packaging tool, like this:

```
pip install tox
```

Writing documentation for redislite just requires a working python with the tox python package installed.

Running and testing code requires a working python development environment and c compiler sufficient to compile redis.

Redislite Development Site

Redislite development occurs on github at: https://github.com/yahoo/redislite

All development should occur on a fork or branch of the redislite github repo.

Building Codebase Familiarity by Reviewing Pull Requests

Look at pull requests to build familiarity with the codebase and the process. Provide comments and feedback if you see issues or can provide more information.

Some things to look for:

- See if the code is missing docs or tests.
- Look through the changes a pull request makes, and keep an eye out for syntax that is incompatible with older but still supported versions of Python.
- Make sure all the tests run and pass under all versions of python.
- Leave comments and feedback!
- Often times the codebase will change between a patch being submitted and the time it gets reviewed. Make sure it an older pull request still aapplies cleanly to the Master branch and functions as expected.

Writing code

Coding style

Code submitted to *redislite* should be compliant with the python pep8 style guide.

Prior to submitting code, check for pep8 conformance by running:

```
tox -e pep8
```

Then fix any issues.

Check Code for Common Errors

Before submitting a pull request it's a good idea to run a code analysis tool to identify any common errors and indicators of bad code. Using python tools such as pylint or flake8.

This can be done by running:

```
tox -e pylint
```

Testing

Any changes to source code should be tested, both for regression and for validation of new code. All tests can be run using the tox tool without any arguments:

```
tox
```

Unit Tests

Working unit tests are required for all code that adds new functionality. Running the unit tests will generate a coverage report at the end of the test output. The report should show 100% coverage on all code. The report looks like:

```
Name                        Stmts   Miss Branch BrMiss  Cover   Missing
-----------------------------------------------------------------------
redislite                       6      0      0      0   100%
redislite.client              122      0     22      0   100%
redislite.configuration        11      0      0      0   100%
redislite.patch                41      0     12      0   100%
-----------------------------------------------------------------------
TOTAL                         180      0     34      0   100%
-----------------------------------------------------------------------
```

To see this report, run:

```
tox
```

Improvements to documentation

Writing documentation

The redislite documentation is good but it can always be improved. Did you find a typo? Do you think that something should be clarified? Go ahead and update the documentation in the docs/source directory.

Once your documentation changes have been made, run the following to generate the html documentation.:

```
tox -e build_docs
```

Then open the `build/sphinx/html/index.html` file in your web browser to ensure the generated documentation looks correct.

Once the documentation looks correct, go ahead and submit a pull request.

Writing style

Code submitted to *redislite* should be compliant with the python pep8 style guide.

Prior to submitting code, check for pep8 conformance by running:

```
tox -e pep8
```

Then fix any issues.

FAQ

How can I help with triaging?

If there is an uncommented issue that reports a bug, try and reproduce it. If you can reproduce it and it seems valid, add a comment that you confirmed the bug. Consider writing a code to test for the bug's behavior, even if you don't fix the bug itself.

Submitting Code

To submit your code for inclusion upstream, do the following to ensure your submission only includes your new changes:

1. Make sure you have Completed a Yahoo CLA Agreement.
2. Redislite development occurs on github at: https://github.com/yahoo/redislite. All development should occur on a fork of the redislite github repo.
3. Prior to submitting a pull request, perform a merge from the MASTER branch of the main redislite repository into your fork. This will ensure your pull request only includes your changes and will allow you to deal with any upstream changes that affect your code.
4. Clear up all PEP8 issues before submission. This will ensure your changesets only include code changes and not formatting changes.
5. Clear up or document exceptions for all PyLint/Flake8 issues. This will ensure the evaluation and review of your code does not have common coding errors and decrease the human time to evaluate changes.

Reviewing Pull Requests

When a pull request is submitted, three automated checks will automatically run, these checks are:

1. Check that the submitter of the pull request has a Yahoo CLA Agreement agreement on file.
2. Check that all tests run without errors on all python releases that redislite supports.
3. Check to ensure the coverage or amount of code that is not tested did not increase.

As these checks run the pull request will be annotated with the results. If any of these checks fail the issue found needs to be fixed before the pull request can be applied.

CI Pipelines

Any new change branches should build correctly using CI prior to being submitted for upstream inclusion.

Local changes can be tested by running:

```
tox
```

in the git root directory.

When a pull request is submitted the travis-ci service will automatically run the tests on the code in the pull request and annotate the pull request with the results.

Pull requests should never be submitted before the travis-ci pipeline indicates the tests all pass.

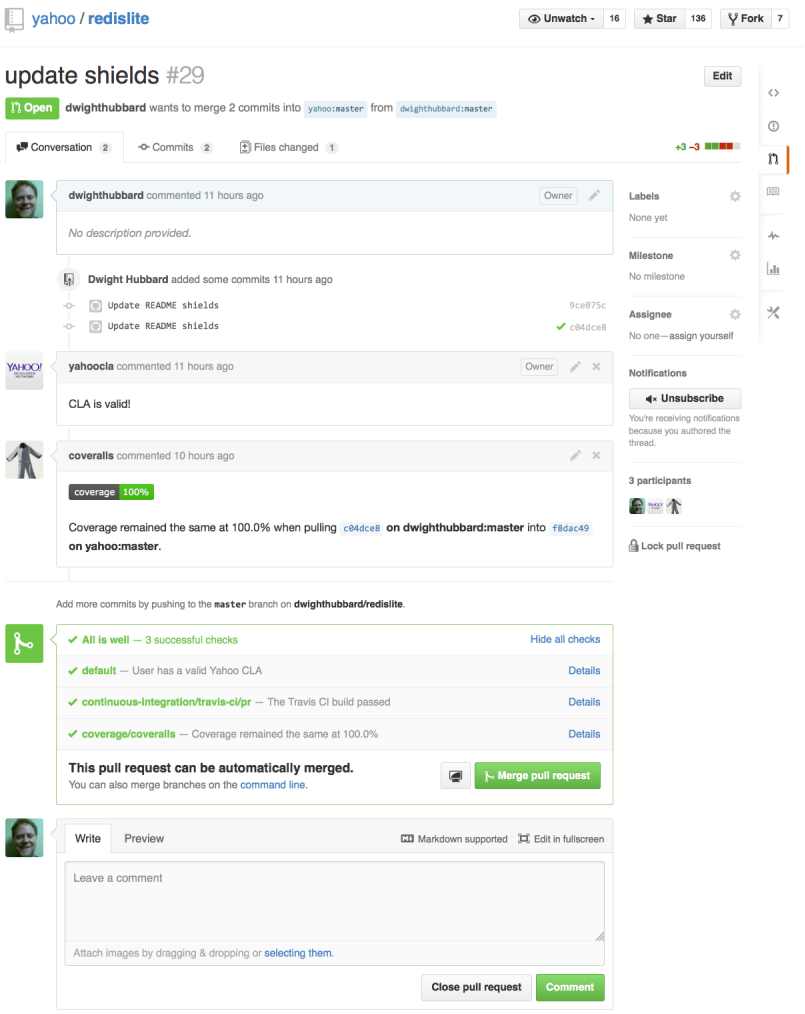

Fig. 1.1: An example of a pull request that successfully passed all automated checks.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

O

T

U

W

Z