redis-py Documentation

Release 2.10.5

Andy McCurdy

Feb 17, 2019

Contents

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

CHAPTER 2

Contents:

class redis.**Redis**(*host=u'localhost'*, *port=6379*, *db=0*, *password=None*, *socket_timeout=None*, *socket_connect_timeout=None*, *socket_keepalive=None*, *socket_keepalive_options=None*, *connection_pool=None*, *unix_socket_path=None*, *encoding=u'utf-8'*, *encoding_errors=u'strict'*, *charset=None*, *errors=None*, *decode_responses=False*, *retry_on_timeout=False*, *ssl=False*, *ssl_keyfile=None*, *ssl_certfile=None*, *ssl_cert_reqs=u'required'*, *ssl_ca_certs=None*, *max_connections=None*)

Implementation of the Redis protocol.

This abstract class provides a Python interface to all Redis commands and an implementation of the Redis protocol.

Connection and Pipeline derive from this, implementing how the commands are sent and received to the Redis server

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

bitfield(*key*, *default_overflow=None*)

Return a BitFieldOperation instance to conveniently construct one or more bitfield operations on `key`.

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 RPOP, 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.

bzpopmax(*keys*, *timeout=0*)
ZPOPMAX a value off of the first non-empty sorted set named in the `keys` list.

If none of the sorted sets in `keys` has a value to ZPOPMAX, then block for `timeout` seconds, or until a member gets added to one of the sorted sets.

If timeout is 0, then block indefinitely.

bzpopmin(*keys*, *timeout=0*)
ZPOPMIN a value off of the first non-empty sorted set named in the `keys` list.

If none of the sorted sets in `keys` has a value to ZPOPMIN, then block for `timeout` seconds, or until a member gets added to one of the sorted sets.

If timeout is 0, then block indefinitely.

client_getname()
Returns the current connection name

client_id()
Returns the current connection id

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

client_kill_filter(*_id=None*, *_type=None*, *addr=None*, *skipme=None*)
Disconnects client(s) using a variety of filter options :param id: Kills a client by its unique ID field :param type: Kills a client by type where type is one of 'normal', 'master', 'slave' or 'pubsub' :param addr: Kills a client by its 'address:port' :param skipme: If True, then the client calling the command will not get killed even if it is identified by one of the filter options. If skipme is not provided, the server defaults to skipme=True

client_list(*_type=None*)
Returns a list of currently connected clients. If type of client specified, only that type will be returned. :param _type: optional. one of the client types (normal, master,

replica, pubsub)

client_pause(*timeout*)
Suspend all the Redis clients for the specified amount of time :param timeout: milliseconds to pause clients

client_setname(*name*)
Sets the current connection name

client_unblock(*client_id*, *error=False*)
Unblocks a connection by its client id. If `error` is True, unblocks the client with a special error message. If `error` is False (default), the client is unblocked using the regular timeout mechanism.

config_get(*pattern=u'*'*)
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`

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`

decrby(*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(**names*)
Returns the number of `names` that exist

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(*asynchronous=False*)
Delete all keys in all databases on the current host.

`asynchronous` indicates whether the operation is executed asynchronously by the server.

flushdb(*asynchronous=False*)
Delete all keys in the current database.

`asynchronous` indicates whether the operation is executed asynchronously by the server.

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

For example:

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

Three URL schemes are supported:

- `redis:// <http://www.iana.org/assignments/uri-schemes/prov/redis>`_ creates a normal TCP socket connection
- `rediss:// <http://www.iana.org/assignments/uri-schemes/prov/rediss>`_ creates a SSL wrapped TCP socket connection
- `unix://` creates a Unix Domain Socket connection

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.

geoadd(*name*, **values*)
Add the specified geospatial items to the specified key identified by the `name` argument. The Geospatial items are given as ordered members of the `values` argument, each item or place is formed by the triad longitude, latitude and name.

geodist(*name*, *place1*, *place2*, *unit=None*)
Return the distance between `place1` and `place2` members of the `name` key. The units must be one of the following : m, km mi, ft. By default meters are used.

geohash(*name*, **values*)
Return the geo hash string for each item of `values` members of the specified key identified by the `name` argument.

geopos(*name*, **values*)
Return the positions of each item of `values` as members of the specified key identified by the `name` argument. Each position is represented by the pairs lon and lat.

georadius(*name*, *longitude*, *latitude*, *radius*, *unit=None*, *withdist=False*, *withcoord=False*, *withhash=False*, *count=None*, *sort=None*, *store=None*, *store_dist=None*)
Return the members of the specified key identified by the `name` argument which are within the borders of the area specified with the `latitude` and `longitude` location and the maximum distance from the center specified by the `radius` value.

The units must be one of the following : m, km mi, ft. By default

`withdist` indicates to return the distances of each place.

`withcoord` indicates to return the latitude and longitude of each place.

`withhash` indicates to return the geohash string of each place.

`count` indicates to return the number of elements up to N.

`sort` indicates to return the places in a sorted way, ASC for nearest to fairest and DESC for fairest to nearest.

`store` indicates to save the places names in a sorted set named with a specific key, each element of the destination sorted set is populated with the score got from the original geo sorted set.

`store_dist` indicates to save the places names in a sorted set named with a specific key, instead of `store` the sorted set destination score is set with the distance.

georadiusbymember(*name*, *member*, *radius*, *unit=None*, *withdist=False*, *withcoord=False*, *withhash=False*, *count=None*, *sort=None*, *store=None*, *store_dist=None*)
This command is exactly like `georadius` with the sole difference that instead of taking, as the center of the area to query, a longitude and latitude value, it takes the name of a member already existing inside the geospatial index represented by the sorted set.

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.

hstrlen(*name*, *key*)
Return the number of bytes stored in the value of `key` within hash `name`

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=u'*'*)
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

memory_purge()
Attempts to purge dirty pages for reclamation by allocator

memory_usage(*key*, *samples=None*)
Return the total memory usage for key, its value and associated administrative overheads.

For nested data structures, samples is the number of elements to sample. If left unspecified, the server's default is 5. Use 0 to sample all elements.

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

migrate(*host*, *port*, *keys*, *destination_db*, *timeout*, *copy=False*, *replace=False*, *auth=None*)
Migrate 1 or more keys from the current Redis server to a different server specified by the host, port and destination_db.

The timeout, specified in milliseconds, indicates the maximum time the connection between the two servers can be idle before the command is interrupted.

If copy is True, the specified keys are NOT deleted from the source server.

If replace is True, this operation will overwrite the keys on the destination server if they exist.

If auth is specified, authenticate to the destination server with the password provided.

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

mset(*mapping*)
Sets key/values based on a mapping. Mapping is a dictionary of key/value pairs. Both keys and values should be strings or types that can be cast to a string via str().

msetnx(*mapping*)
Sets key/values based on a mapping if none of the keys are already set. Mapping is a dictionary of key/value pairs. Both keys and values should be strings or types that can be cast to a string via str(). 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.

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.

pubsub_channels(*pattern=u'*'*)
Return a list of channels that have at least one subscriber

pubsub_numpat()
Returns the number of subscriptions to patterns

pubsub_numsub(**args*)
Return a list of (channel, number of subscribers) tuples for each channel given in `*args`

randomkey()
Returns the name of a random key

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*, *replace=False*)
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 only if it does not exist.

xx if set to True, set the value at key name to value only 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(*save=False*, *nosave=False*)
Shutdown the Redis server. If Redis has persistence configured, data will be flushed before shutdown. If the "save" option is set, a data flush will be attempted even if there is no persistence configured. If the "nosave" option is set, no data flush will be attempted. The "save" and "nosave" options cannot both be set.

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*, *count=None*)
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.

swapdb(*first*, *second*)
Swap two databases

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

touch(**args*)
Alters the last access time of a key(s) `*args`. A key is ignored if it does not exist.

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`

unlink(**names*)
Unlink one or more keys specified by `names`

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

xack(*name*, *groupname*, **ids*)
Acknowledges the successful processing of one or more messages. name: name of the stream. groupname: name of the consumer group. *ids: message ids to acknowlege.

xadd(*name*, *fields*, *id=u'**', *maxlen=None*, *approximate=True*)
Add to a stream. name: name of the stream fields: dict of field/value pairs to insert into the stream id: Location to insert this record. By default it is appended. maxlen: truncate old stream members beyond this size approximate: actual stream length may be slightly more than maxlen

xclaim(*name*, *groupname*, *consumername*, *min_idle_time*, *message_ids*, *idle=None*, *time=None*, *retrycount=None*, *force=False*, *justid=False*)
Changes the ownership of a pending message. name: name of the stream. groupname: name of the consumer group. consumername: name of a consumer that claims the message. min_idle_time: filter messages that were idle less than this amount of milliseconds message_ids: non-empty list or tuple of message IDs to claim idle: optional. Set the idle time (last time it was delivered) of the

message in ms

time: optional integer. This is the same as idle but instead of a relative amount of milliseconds, it sets the idle time to a specific Unix time (in milliseconds).

retrycount: optional integer. set the retry counter to the specified value. This counter is incremented every time a message is delivered again.

force: optional boolean, false by default. Creates the pending message entry in the PEL even if certain specified IDs are not already in the PEL assigned to a different client.

justid: optional boolean, false by default. Return just an array of IDs of messages successfully claimed, without returning the actual message

xdel(*name*, **ids*)
Deletes one or more messages from a stream. name: name of the stream. *ids: message ids to delete.

xgroup_create(*name*, *groupname*, *id=u'$'*, *mkstream=False*)
Create a new consumer group associated with a stream. name: name of the stream. groupname: name of the consumer group. id: ID of the last item in the stream to consider already delivered.

xgroup_delconsumer(*name*, *groupname*, *consumername*)
Remove a specific consumer from a consumer group. Returns the number of pending messages that the consumer had before it was deleted. name: name of the stream. groupname: name of the consumer group. consumername: name of consumer to delete

xgroup_destroy(*name*, *groupname*)
Destroy a consumer group. name: name of the stream. groupname: name of the consumer group.

xgroup_setid(*name*, *groupname*, *id*)
Set the consumer group last delivered ID to something else. name: name of the stream. groupname: name of the consumer group. id: ID of the last item in the stream to consider already delivered.

xinfo_consumers(*name*, *groupname*)
Returns general information about the consumers in the group. name: name of the stream. groupname: name of the consumer group.

xinfo_groups(*name*)
Returns general information about the consumer groups of the stream. name: name of the stream.

xinfo_stream(*name*)
Returns general information about the stream. name: name of the stream.

xlen(*name*)
Returns the number of elements in a given stream.

xpending(*name*, *groupname*)
Returns information about pending messages of a group. name: name of the stream. groupname: name of the consumer group.

xpending_range(*name*, *groupname*, *min*, *max*, *count*, *consumername=None*)
Returns information about pending messages, in a range. name: name of the stream. groupname: name of the consumer group. min: minimum stream ID. max: maximum stream ID. count: number of messages to return consumername: name of a consumer to filter by (optional).

xrange(*name*, *min=u'-'*, *max=u'+'*, *count=None*)
Read stream values within an interval. name: name of the stream. start: first stream ID. defaults to '-',

meaning the earliest available.

finish: last stream ID. defaults to '+', meaning the latest available.

count: if set, only return this many items, beginning with the earliest available.

xread(*streams*, *count=None*, *block=None*)
Block and monitor multiple streams for new data. streams: a dict of stream names to stream IDs, where

IDs indicate the last ID already seen.

count: if set, only return this many items, beginning with the earliest available.

block: number of milliseconds to wait, if nothing already present.

xreadgroup(*groupname*, *consumername*, *streams*, *count=None*, *block=None*, *noack=False*)
Read from a stream via a consumer group. groupname: name of the consumer group. consumername: name of the requesting consumer. streams: a dict of stream names to stream IDs, where

IDs indicate the last ID already seen.

count: if set, only return this many items, beginning with the earliest available.

block: number of milliseconds to wait, if nothing already present. noack: do not add messages to the PEL

xrevrange(*name*, *max=u'+'*, *min=u'-'*, *count=None*)
Read stream values within an interval, in reverse order. name: name of the stream start: first stream ID. defaults to '+',

meaning the latest available.

finish: last stream ID. defaults to '-', meaning the earliest available.

count: if set, only return this many items, beginning with the latest available.

xtrim(*name*, *maxlen*, *approximate=True*)
Trims old messages from a stream. name: name of the stream. maxlen: truncate old stream messages beyond this size approximate: actual stream length may be slightly more than maxlen

zadd(*name*, *mapping*, *nx=False*, *xx=False*, *ch=False*, *incr=False*)
Set any number of element-name, score pairs to the key `name`. Pairs are specified as a dict of element-names keys to score values.

`nx` forces ZADD to only create new elements and not to update scores for elements that already exist.

`xx` forces ZADD to only update scores of elements that already exist. New elements will not be added.

`ch` modifies the return value to be the numbers of elements changed. Changed elements include new elements that were added and elements whose scores changed.

`incr` modifies ZADD to behave like ZINCRBY. In this mode only a single element/score pair can be specified and the score is the amount the existing score will be incremented by. When using this mode the return value of ZADD will be the new score of the element.

The return value of ZADD varies based on the mode specified. With no options, ZADD returns the number of new elements added to the sorted set.

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*, *amount*, *value*)
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`.

zpopmax(*name*, *count=None*)
Remove and return up to `count` members with the highest scores from the sorted set `name`.

zpopmin(*name*, *count=None*)
Remove and return up to `count` members with the lowest scores from the sorted set `name`.

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.

redis.**StrictRedis**
alias of `redis.client.Redis`

class redis.**ConnectionPool**(*connection_class=<class 'redis.connection.Connection'>*, *max_connections=None*, ***connection_kwargs*)
Generic connection pool

disconnect()
Disconnects all connections in the pool

classmethod from_url(*url*, *db=None*, *decode_components=False*, ***kwargs*)
Return a connection pool configured from the given URL.

For example:

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

Three URL schemes are supported:

- `redis:// <https://www.iana.org/assignments/uri-schemes/prov/redis>‘_ creates a normal TCP socket connection
- `rediss:// <https://www.iana.org/assignments/uri-schemes/prov/rediss>‘_ creates a SSL wrapped TCP socket connection
- unix:// creates a Unix Domain Socket connection

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.

The decode_components argument allows this function to work with percent-encoded URLs. If this argument is set to True all %xx escapes will be replaced by their single-character equivalents after the URL has been parsed. This only applies to the hostname, path, and password components.

Any additional querystring arguments and keyword arguments will be passed along to the ConnectionPool class’s initializer. The querystring arguments socket_connect_timeout and socket_timeout if supplied are parsed as float values. The arguments socket_keepalive and retry_on_timeout are parsed to boolean values that accept True/False, Yes/No values to indicate state. Invalid types cause a UserWarning to be raised. In the case of conflicting arguments, querystring arguments always win.

get_connection(*command_name*, **keys*, ***options*)
Get a connection from the pool

get_encoder()
Return an encoder based on encoding settings

make_connection()
Create a new connection

release(*connection*)
Releases the connection back to the pool

class redis.**BlockingConnectionPool**(*max_connections=50*, *timeout=20*, *connection_class=<class 'redis.connection.Connection'>*, *queue_class=<class Queue.LifoQueue>*, ***connection_kwargs*)
Thread-safe blocking connection pool:

```
>>> from redis.client import Redis
>>> client = Redis(connection_pool=BlockingConnectionPool())
```

It performs the same function as the default :py:class: ~redis.connection.ConnectionPool implementation, in that, it maintains a pool of reusable connections that can be shared by multiple redis clients (safely across threads if required).

The difference is that, in the event that a client tries to get a connection from the pool when all of connections are in use, rather than raising a :py:class: ~redis.exceptions.ConnectionError (as the default :py:class: ~redis.connection.ConnectionPool implementation does), it makes the client wait ("blocks") for a specified number of seconds until a connection becomes available.

Use max_connections to increase / decrease the pool size:

```
>>> pool = BlockingConnectionPool(max_connections=10)
```

Use timeout to tell it either how many seconds to wait for a connection to become available, or to block forever:

Block forever. >>> pool = BlockingConnectionPool(timeout=None)

Raise a ConnectionError after five seconds if a connection is # not available. >>> pool = BlockingConnectionPool(timeout=5)

disconnect()
Disconnects all connections in the pool.

get_connection(*command_name*, **keys*, ***options*)
Get a connection, blocking for self.timeout until a connection is available from the pool.

If the connection returned is None then creates a new connection. Because we use a last-in first-out queue, the existing connections (having been returned to the pool after the initial None values were added) will be returned before None values. This means we only create new connections when we need to, i.e.: the actual number of connections will only increase in response to demand.

make_connection()
Make a fresh connection.

release(*connection*)
Releases the connection back to the pool.

class redis.**Connection**(*host=u'localhost'*, *port=6379*, *db=0*, *password=None*, *socket_timeout=None*, *socket_connect_timeout=None*, *socket_keepalive=False*, *socket_keepalive_options=None*, *socket_type=0*, *retry_on_timeout=False*, *encoding=u'utf-8'*, *encoding_errors=u'strict'*, *decode_responses=False*, *parser_class=<class 'redis.connection.PythonParser'>*, *socket_read_size=65536*)
Manages TCP communication to and from a Redis server

can_read(*timeout=0*)
Poll the socket to see if there's data that can be read.

connect()
Connects to the Redis server if not already connected

disconnect()
Disconnects from the Redis server

is_ready_for_command()
Check if the connection is ready for a command

on_connect()
Initialize the connection, authenticate and select a database

pack_command(**args*)
Pack a series of arguments into the Redis protocol

pack_commands(*commands*)
Pack multiple commands into the Redis protocol

read_response()
Read the response from a previously sent command

send_command(**args*)
Pack and send a command to the Redis server

send_packed_command(*command*)
Send an already packed command to the Redis server

redis.**from_url**(*url*, *db=None*, ***kwargs*)
Returns an active Redis client generated from the given database URL.

Will attempt to extract the database id from the path url fragment, if none is provided.

exception redis.**AuthenticationError**

exception redis.**BusyLoadingError**

exception redis.**ConnectionError**

exception redis.**DataError**

exception redis.**InvalidResponse**

exception redis.**PubSubError**

exception redis.**ReadOnlyError**

exception redis.**RedisError**

exception redis.**ResponseError**

exception redis.**TimeoutError**

exception redis.**WatchError**

Python Module Index

r

Index

T

U

W

X

Z