TRedis

Release 0.8.0

Jul 20, 2018

Contents

An asynchronous Redis client for Tornado

Note: `TRedis` is a work in progress and does not support the entire Redis command set. For a list of the currently supported commands by category, see the *Supported Commands* documentation.

CHAPTER 1

Installation

tredis is available from the Python Package Index and can be installed by running **`pip install tredis`**

CHAPTER 2

Contents

2.1 API

The *Client* class is the primary API interface for interacting with Redis. While the per-method documentation attempts to be as complete as possible, the best documentation source for each Redis command is available on the redis site.

See the *Supported Commands* documentation if you are not able to find a Redis command you are looking for.

class tredis.**Client**(*hosts*, *on_close=None*, *io_loop=None*, *clustering=False*, *auto_connect=True*)

Asynchronous Redis client that supports Redis with master/slave failover and clustering. When `clustering` is `True`, the client will automatically discover all of the nodes in the cluster and connect to them.

The `hosts` argument should contain a list of Redis servers to connect to. The connection information for the server should be a `dict`. In the following example, the client will connect to Redis running at `127.0.0.1` on port `6379` using database # `2`:

```
class RequestHandler(web.RequestHandler):

    @gen.coroutine
    def connect_to_redis(self)
        client = tredis.Client([{
                'host': '127.0.0.1', 'port': 6379, 'db': 2
            }], auto_connect=False, clustering=True)
        yield client.connect()
```

When `auto_connect` is set to `True`, the connection to the Redis server or the Redis cluster starts on creation of the client. You should be aware that this will not block on creation and the connection will be established asynchronously in the background. Any requests made with the client while it is connecting will block until the connection is available.

When `auto_connect` is set to `False`, you will need to invoke the *connect()* method, yielding to the `Future` that it returns.

Parameters

- **hosts** (*list(dict)*) – A list of host connection values.
- **io_loop** (*tornado.ioloop.IOLoop*) – Override the current Tornado IOLoop instance
- **on_close** (*method*) – The method to call if the connection is closed
- **clustering** (*bool*) – Toggle the cluster support in the client
- **auto_connect** (*bool*) – Toggle the auto-connect on creation feature

append(*key*, *value*)

If key already exists and is a string, this command appends the value at the end of the string. If key does not exist it is created and set as an empty string, so *append()* will be similar to *set()* in this special case.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`. The amortized time complexity is `O(1)` assuming the appended value is small and the already present value is of any size, since the dynamic string library used by Redis will double the free space available on every reallocation.

Parameters

- **key** (`str`, `bytes`) – The key to get
- **value** (`str`, `bytes`) – The value to append to the key

Returns The length of the string after the append operation

Return type int

Raises *RedisError*

auth(*password*)

Request for authentication in a password-protected Redis server. Redis can be instructed to require a password before allowing clients to execute commands. This is done using the `requirepass` directive in the configuration file.

If the password does not match, an *AuthError* exception will be raised.

Parameters **password** (`str`, `bytes`) – The password to authenticate with

Return type bool

Raises *AuthError*, *RedisError*

bitcount(*key*, *start=None*, *end=None*)

Count the number of set bits (population counting) in a string.

By default all the bytes contained in the string are examined. It is possible to specify the counting operation only in an interval passing the additional arguments start and end.

Like for the *getrange()* command start and end can contain negative values in order to index bytes starting from the end of the string, where `-1` is the last byte, `-2` is the penultimate, and so forth.

Non-existent keys are treated as empty strings, so the command will return zero.

New in version 0.2.0.

Note: **Time complexity**: `O(N)`

Parameters

- **key** (*str*, *bytes*) – The key to get
- **start** (*int*) – The start position to evaluate in the string
- **end** (*int*) – The end position to evaluate in the string

Return type int

Raises *RedisError*, ValueError

bitop(*operation*, *dest_key*, **keys*)

Perform a bitwise operation between multiple keys (containing string values) and store the result in the destination key.

The values for operation can be one of:

- `b'AND'`
- `b'OR'`
- `b'XOR'`
- `b'NOT'`
- `tredis.BITOP_AND` or `b'&'`
- `tredis.BITOP_OR` or `b'|'`
- `tredis.BITOP_XOR` or `b'^'`
- `tredis.BITOP_NOT` or `b'~'`

`b'NOT'` is special as it only takes an input key, because it performs inversion of bits so it only makes sense as an unary operator.

The result of the operation is always stored at `dest_key`.

Handling of strings with different lengths

When an operation is performed between strings having different lengths, all the strings shorter than the longest string in the set are treated as if they were zero-padded up to the length of the longest string.

The same holds true for non-existent keys, that are considered as a stream of zero bytes up to the length of the longest string.

New in version 0.2.0.

Note: Time complexity: `O(N)`

Parameters

- **operation** (*bytes*) – The operation to perform
- **dest_key** (*str*, *bytes*) – The key to store the bitwise operation results to
- **keys** (*str*, *bytes*) – One or more keys as keyword parameters for the bitwise op

Returns The size of the string stored in the destination key, that is equal to the size of the longest input string.

Return type int

Raises *RedisError*, ValueError

bitpos(*key*, *bit*, *start=None*, *end=None*)
Return the position of the first bit set to `1` or `0` in a string.

The position is returned, thinking of the string as an array of bits from left to right, where the first byte's most significant bit is at position 0, the second byte's most significant bit is at position `8`, and so forth.

The same bit position convention is followed by *getbit()* and *setbit()*.

By default, all the bytes contained in the string are examined. It is possible to look for bits only in a specified interval passing the additional arguments start and end (it is possible to just pass start, the operation will assume that the end is the last byte of the string. However there are semantic differences as explained later). 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.

Note that bit positions are returned always as absolute values starting from bit zero even when start and end are used to specify a range.

Like for the *getrange()* command start and end can contain negative values in order to index bytes starting from the end of the string, where `-1` is the last byte, `-2` is the penultimate, and so forth.

Non-existent keys are treated as empty strings.

New in version 0.2.0.

Note: Time complexity: `O(N)`

Parameters

- **key** (`str`, `bytes`) – The key to get
- **bit** (*int*) – The bit value to search for (`1` or `0`)
- **start** (*int*) – The start position to evaluate in the string
- **end** (*int*) – The end position to evaluate in the string

Returns The position of the first bit set to `1` or `0`

Return type int

Raises *RedisError*, `ValueError`

close()
Close any open connections to Redis.

Raises *tredis.exceptions.ConnectionError*

cluster_info()
`CLUSTER INFO` provides `INFO` style information about Redis Cluster vital parameters.

New in version 0.7.0.

Returns A dictionary of current cluster information

Return type dict

Key cluster_state State is ok if the node is able to receive queries. fail if there is at least one hash slot which is unbound (no node associated), in error state (node serving it is flagged with `FAIL` flag), or if the majority of masters can't be reached by this node.

Key cluster_slots_assigned Number of slots which are associated to some node (not unbound). This number should be `16384` for the node to work properly, which means that each hash slot should be mapped to a node.

Key cluster_slots_ok Number of hash slots mapping to a node not in `FAIL` or `PFAIL` state.

Key cluster_slots_pfail Number of hash slots mapping to a node in `PFAIL` state. Note that those hash slots still work correctly, as long as the `PFAIL` state is not promoted to `FAIL` by the failure detection algorithm. `PFAIL` only means that we are currently not able to talk with the node, but may be just a transient error.

Key cluster_slots_fail Number of hash slots mapping to a node in `FAIL` state. If this number is not zero the node is not able to serve queries unless cluster-require-full-coverage is set to no in the configuration.

Key cluster_known_nodes The total number of known nodes in the cluster, including nodes in `HANDSHAKE` state that may not currently be proper members of the cluster.

Key cluster_size The number of master nodes serving at least one hash slot in the cluster.

Key cluster_current_epoch The local Current Epoch variable. This is used in order to create unique increasing version numbers during fail overs.

Key cluster_my_epoch The Config Epoch of the node we are talking with. This is the current configuration version assigned to this node.

Key cluster_stats_messages_sent Number of messages sent via the cluster node-to-node binary bus.

Key cluster_stats_messages_received Number of messages received via the cluster node-to-node binary bus.

Raises *RedisError*

cluster_nodes()

Each node in a Redis Cluster has its view of the current cluster configuration, given by the set of known nodes, the state of the connection we have with such nodes, their flags, properties and assigned slots, and so forth.

`CLUSTER NODES` provides all this information, that is, the current cluster configuration of the node we are contacting, in a serialization format which happens to be exactly the same as the one used by Redis Cluster itself in order to store on disk the cluster state (however the on disk cluster state has a few additional info appended at the end).

Note that normally clients willing to fetch the map between Cluster hash slots and node addresses should use `CLUSTER SLOTS` instead. `CLUSTER NODES`, that provides more information, should be used for administrative tasks, debugging, and configuration inspections. It is also used by `redis-trib` in order to manage a cluster.

New in version 0.7.0.

Return type list(*ClusterNode*)

Raises *RedisError*

connect()

Connect to the Redis server or Cluster.

Return type tornado.concurrent.Future

decr(*key*)

Decrements the number stored at key by one. If the key does not exist, it is set to 0 before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that can not be represented as integer. This operation is limited to 64 bit signed integers.

See *incr()* for extra information on increment/decrement operations.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to decrement

Returns The value of key after the decrement

Return type int

Raises *RedisError*

decrby(*key*, *decrement*)

Decrements the number stored at key by decrement. If the key does not exist, it is set to 0 before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that can not be represented as integer. This operation is limited to 64 bit signed integers.

See *incr()* for extra information on increment/decrement operations.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to decrement
- **decrement** (*int*) – The amount to decrement by

Returns The value of key after the decrement

Return type int

Raises *RedisError*

delete(**keys*)

Removes the specified keys. A key is ignored if it does not exist. Returns `True` if all keys are removed.

Note: **Time complexity**: `O(N)` where `N` is the number of keys that will be removed. When a key to remove holds a value other than a string, the individual complexity for this key is `O(M)` where `M` is the number of elements in the list, set, sorted set or hash. Removing a single key that holds a string value is `O(1)`.

Parameters **keys** (`str`, `bytes`) – One or more keys to remove

Return type bool

Raises *RedisError*

dump(*key*)

Serialize the value stored at key in a Redis-specific format and return it to the user. The returned value can be synthesized back into a Redis key using the *restore()* command.

The serialization format is opaque and non-standard, however it has a few semantic characteristics:

- It contains a 64-bit checksum that is used to make sure errors will be detected. The *restore()* command makes sure to check the checksum before synthesizing a key using the serialized value.

- Values are encoded in the same format used by RDB.
- An RDB version is encoded inside the serialized value, so that different Redis versions with incompatible RDB formats will refuse to process the serialized value.
- The serialized value does NOT contain expire information. In order to capture the time to live of the current value the *pttl()* command should be used.

If key does not exist None is returned.

Note: **Time complexity**: O(1) to access the key and additional O(N*M) to serialized it, where N is the number of Redis objects composing the value and M their average size. For small string values the time complexity is thus O(1)+O(1*M) where M is small, so simply O(1).

Parameters **key** (str, bytes) – The key to dump

Return type bytes, None

echo(*message*)

Returns the message that was sent to the Redis server.

Parameters **message** (str, bytes) – The message to echo

Return type bytes

Raises *RedisError*

eval(*script*, *keys=None*, *args=None*)

eval() and *evalsha()* are used to evaluate scripts using the Lua interpreter built into Redis starting from version 2.6.0.

The first argument of EVAL is a Lua 5.1 script. The script does not need to define a Lua function (and should not). It is just a Lua program that will run in the context of the Redis server.

Note: **Time complexity**: Depends on the script that is executed.

Parameters

- **script** (*str*) – The Lua script to execute
- **keys** (*list*) – A list of keys to pass into the script
- **args** (*list*) – A list of args to pass into the script

Returns mixed

evalsha(*sha1*, *keys=None*, *args=None*)

Evaluates a script cached on the server side by its SHA1 digest. Scripts are cached on the server side using the *script_load()* command. The command is otherwise identical to *eval()*.

Note: **Time complexity**: Depends on the script that is executed.

Parameters

- **sha1** (*str*) – The sha1 hash of the script to execute
- **keys** (*list*) – A list of keys to pass into the script

- **args** (*list*) – A list of args to pass into the script

Returns mixed

exists(*key*)

Returns `True` if the key exists.

Note: Time complexity: `O(1)`

Command Type: String

Parameters **key** (`str`, `bytes`) – One or more keys to check for

Return type bool

Raises *RedisError*

expire(*key*, *timeout*)

Set a timeout on key. After the timeout has expired, the key will automatically be deleted. A key with an associated timeout is often said to be volatile in Redis terminology.

The timeout is cleared only when the key is removed using the *delete()* method or overwritten using the *set()* or *getset()* methods. This means that all the operations that conceptually alter the value stored at the key without replacing it with a new one will leave the timeout untouched. For instance, incrementing the value of a key with *incr()*, pushing a new value into a list with *lpush()*, or altering the field value of a hash with *hset()* are all operations that will leave the timeout untouched.

The timeout can also be cleared, turning the key back into a persistent key, using the *persist()* method.

If a key is renamed with *rename()*, the associated time to live is transferred to the new key name.

If a key is overwritten by *rename()*, like in the case of an existing key `Key_A` that is overwritten by a call like `client.rename(Key_B, Key_A)` it does not matter if the original `Key_A` had a timeout associated or not, the new key `Key_A` will inherit all the characteristics of `Key_B`.

Note: Time complexity: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to set an expiration for
- **timeout** (*int*) – The number of seconds to set the timeout to

Return type bool

Raises *RedisError*

expireat(*key*, *timestamp*)

expireat() has the same effect and semantic as *expire()*, but instead of specifying the number of seconds representing the TTL (time to live), it takes an absolute Unix timestamp (seconds since January 1, 1970).

Please for the specific semantics of the command refer to the documentation of *expire()*.

Note: Time complexity: `O(1)`

Parameters

- **key** (str, bytes) – The key to set an expiration for
- **timestamp** (*int*) – The UNIX epoch value for the expiration

Return type bool

Raises *RedisError*

get(*key*)

Get the value of key. If the key does not exist the special value None is returned. An error is returned if the value stored at key is not a string, because *get()* only handles string values.

Note: Time complexity: O(1)

Parameters **key** (str, bytes) – The key to get

Return type bytes|None

Raises *RedisError*

getbit(*key*, *offset*)

Returns the bit value at offset in the string value stored at key.

When offset is beyond the string length, the string is assumed to be a contiguous space with 0 bits. When key does not exist it is assumed to be an empty string, so offset is always out of range and the value is also assumed to be a contiguous space with 0 bits.

New in version 0.2.0.

Note: Time complexity: O(1)

Parameters

- **key** (str, bytes) – The key to get the bit from
- **offset** (*int*) – The bit offset to fetch the bit from

Return type bytes|None

Raises *RedisError*

getrange(*key*, *start*, *end*)

Returns the bit value at offset in the string value stored at key.

When offset is beyond the string length, the string is assumed to be a contiguous space with 0 bits. When key does not exist it is assumed to be an empty string, so offset is always out of range and the value is also assumed to be a contiguous space with 0 bits.

New in version 0.2.0.

Note: Time complexity: O(N) where N is the length of the returned string. The complexity is ultimately determined by the returned length, but because creating a substring from an existing string is very cheap, it can be considered O(1) for small strings.

Parameters

- **key** (str, bytes) – The key to get the bit from
- **start** (*int*) – The start position to evaluate in the string
- **end** (*int*) – The end position to evaluate in the string

Return type bytes|None

Raises *RedisError*

getset(*key*, *value*)
Atomically sets key to value and returns the old value stored at key. Returns an error when key exists but does not hold a string value.

New in version 0.2.0.

Note: **Time complexity**: O(1)

Parameters

- **key** (str, bytes) – The key to remove
- **value** (str, bytes) – The value to set

Returns The previous value

Return type bytes

Raises *RedisError*

hdel(*key*, **fields*)
Remove the specified fields from the hash stored at *key*.

Specified fields that do not exist within this hash are ignored. If *key* does not exist, it is treated as an empty hash and this command returns zero.

Parameters

- **key** (str, bytes) – The key of the hash
- **fields** – iterable of field names to retrieve

Returns the number of fields that were removed from the hash, not including specified by non-existing fields.

Return type int

hexists(*key*, *field*)
Returns if *field* is an existing field in the hash stored at *key*.

Note: *Time complexity*: O(1)

Parameters

- **key** (str, bytes) – The key of the hash
- **field** – name of the field to test for

Return type bool

hget(*key*, *field*)
Returns the value associated with *field* in the hash stored at *key*.

Note: **Time complexity**: always `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **field** – The field in the hash to get

Return type bytes, list

Raises *RedisError*

hgetall(*key*)
Returns all fields and values of the has stored at *key*.

The underlying redis HGETALL command returns an array of pairs. This method converts that to a Python `dict`. It will return an empty `dict` when the key is not found.

Note: **Time complexity**: `O(N)` where `N` is the size of the hash.

Parameters **key** (`str`, `bytes`) – The key of the hash

Returns a `dict` of key to value mappings for all fields in the hash

hincrby(*key*, *field*, *increment*)
Increments the number stored at *field* in the hash stored at *key*.

If *key* does not exist, a new key holding a hash is created. If *field* does not exist the value is set to `0` before the operation is performed. The range of values supported is limited to 64-bit signed integers.

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **field** – name of the field to increment
- **increment** (*int*) – amount to increment by

Returns the value at *field* after the increment occurs

Return type int

hincrbyfloat(*key*, *field*, *increment*)
Increments the number stored at *field* in the hash stored at *key*.

If the increment value is negative, the result is to have the hash field **decremented** instead of incremented. If the field does not exist, it is set to `0` before performing the operation. An error is returned if one of the following conditions occur:

- the field contains a value of the wrong type (not a string)
- the current field content or the specified increment are not parseable as a double precision floating point number

Note: *Time complexity*: `O(1)`

Parameters

- **key** (str, bytes) – The key of the hash
- **field** – name of the field to increment
- **increment** (*float*) – amount to increment by

Returns the value at *field* after the increment occurs

Return type float

hkeys(*key*)
Returns all field names in the hash stored at *key*.

Note: *Time complexity*: `O(N)` where `N` is the size of the hash

---

Parameters **key** (str, bytes) – The key of the hash

Returns the list of fields in the hash

Return type list

hlen(*key*)
Returns the number of fields contained in the hash stored at *key*.

Note: *Time complexity*: `O(1)`

Parameters **key** (str, bytes) – The key of the hash

Returns the number of fields in the hash or zero when *key* does not exist

Return type int

hmget(*key*, **fields*)
Returns the values associated with the specified *fields* in a hash.

For every `field` that does not exist in the hash, `None` is returned. Because a non-existing keys are treated as empty hashes, calling *hmget()* against a non-existing key will return a list of `None` values.

Note: *Time complexity*: `O(N)` where `N` is the number of fields being requested.

Parameters

- **key** (str, bytes) – The key of the hash
- **fields** – iterable of field names to retrieve

Returns a `dict` of field name to value mappings for each of the requested fields

Return type dict

hmset(*key*, *value_dict*)
Sets fields to values as in *value_dict* in the hash stored at *key*.

Sets the specified fields to their respective values in the hash stored at *key*. This command overwrites any specified fields already existing in the hash. If *key* does not exist, a new key holding a hash is created.

Note: **Time complexity**: `O(N)` where `N` is the number of fields being set.

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **value_dict** (`dict`) – field to value mapping

Return type bool

Raises *RedisError*

hset(*key*, *field*, *value*)
Sets *field* in the hash stored at *key* to *value*.

If *key* does not exist, a new key holding a hash is created. If *field* already exists in the hash, it is overwritten.

Note: **Time complexity**: always `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **field** – The field in the hash to set
- **value** – The value to set the field to

Returns `1` if *field* is a new field in the hash and *value* was set; otherwise, `0` if *field* already exists in the hash and the value was updated

Return type int

hsetnx(*key*, *field*, *value*)
Sets *field* in the hash stored at *key* only if it does not exist.

Sets *field* in the hash stored at *key* only if *field* does not yet exist. If *key* does not exist, a new key holding a hash is created. If *field* already exists, this operation has no effect.

Note: *Time complexity*: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **field** – The field in the hash to set
- **value** – The value to set the field to

Returns `1` if *field* is a new field in the hash and *value* was set. `0` if *field* already exists in the hash and no operation was performed

Return type int

hvals(*key*)
Returns all values in the hash stored at *key*.

Note: *Time complexity* `O(N)` where `N` is the size of the hash

Parameters **key** (`str`, `bytes`) – The key of the hash

Returns a `list` of `bytes` instances or an empty list when *key* does not exist

Return type list

incr(*key*)

Increments the number stored at key by one. If the key does not exist, it is set to `0` before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that can not be represented as integer. This operation is limited to 64 bit signed integers.

Note: This is a string operation because Redis does not have a dedicated integer type. The string stored at the key is interpreted as a base-10 64 bit signed integer to execute the operation.

Redis stores integers in their integer representation, so for string values that actually hold an integer, there is no overhead for storing the string representation of the integer.

Note: **Time complexity**: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to increment

Return type int

Raises *RedisError*

incrby(*key*, *increment*)

Increments the number stored at key by increment. If the key does not exist, it is set to 0 before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that can not be represented as integer. This operation is limited to 64 bit signed integers.

See *incr()* for extra information on increment/decrement operations.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to increment
- **increment** (*int*) – The amount to increment by

Returns The value of key after the increment

Return type int

Raises *RedisError*

incrbyfloat(*key*, *increment*)

Increment the string representing a floating point number stored at key by the specified increment. If the

key does not exist, it is set to 0 before performing the operation. An error is returned if one of the following conditions occur:

- The key contains a value of the wrong type (not a string).
- The current key content or the specified increment are not parsable as a double precision floating point number.

If the command is successful the new incremented value is stored as the new value of the key (replacing the old one), and returned to the caller as a string.

Both the value already contained in the string key and the increment argument can be optionally provided in exponential notation, however the value computed after the increment is stored consistently in the same format, that is, an integer number followed (if needed) by a dot, and a variable number of digits representing the decimal part of the number. Trailing zeroes are always removed.

The precision of the output is fixed at 17 digits after the decimal point regardless of the actual internal precision of the computation.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to increment
- **increment** (*float*) – The amount to increment by

Returns The value of key after the increment

Return type bytes

Raises *RedisError*

info(*section=None*)

The INFO command returns information and statistics about the server in a format that is simple to parse by computers and easy to read by humans.

The optional parameter can be used to select a specific section of information:

- server: General information about the Redis server
- clients: Client connections section
- memory: Memory consumption related information
- persistence: RDB and AOF related information
- stats: General statistics
- replication: Master/slave replication information
- cpu: CPU consumption statistics
- commandstats: Redis command statistics
- cluster: Redis Cluster section
- keyspace: Database related statistics

It can also take the following values:

- all: Return all sections

- default: Return only the default set of sections

When no parameter is provided, the default option is assumed.

Parameters **section** (*str*) – Optional

Returns dict

keys(*pattern*)

Returns all keys matching pattern.

While the time complexity for this operation is O(N), the constant times are fairly low. For example, Redis running on an entry level laptop can scan a 1 million key database in 40 milliseconds.

Warning: Consider *keys()* as a command that should only be used in production environments with extreme care. It may ruin performance when it is executed against large databases. This command is intended for debugging and special operations, such as changing your keyspace layout. Don't use *keys()* in your regular application code. If you're looking for a way to find keys in a subset of your keyspace, consider using *scan()* or sets.

Supported glob-style patterns:

- h?llo matches hello, hallo and hxllo
- h*llo matches hllo and heeeello
- h[ae]llo matches hello and hallo, but not hillo
- h[^e]llo matches hallo, hbllo, but not hello
- h[a-b]llo matches hallo and hbllo

Use a backslash (\) to escape special characters if you want to match them verbatim.

Note: **Time complexity**: O(N)

Parameters **pattern** (str, bytes) – The pattern to use when looking for keys

Return type list

Raises *RedisError*

llen(*key*)

Returns the length of the list stored at key.

Parameters **key** (str, bytes) – The list's key

Return type int

Raises *TRedisException*

If key does not exist, it is interpreted as an empty list and 0 is returned. An error is returned when the value stored at key is not a list.

Note: **Time complexity** O(1)

lpop(*key*)

Removes and returns the first element of the list stored at key.

Parameters **key** (str, bytes) – The list's key

Returns the element at the head of the list, None if the list does not exist

Raises *TRedisException*

Note: **Time complexity**: O(1)

---

lpush(*key*, **values*)
Insert all the specified values at the head of the list stored at key.

Parameters

- **key** (str, bytes) – The list's key
- **values** – One or more positional arguments to insert at the beginning of the list. Each value is inserted at the beginning of the list individually (see discussion below).

Returns the length of the list after push operations

Return type int

Raises *TRedisException*

If *key* does not exist, it is created as empty list before performing the push operations. When key holds a value that is not a list, an error is returned.

It is possible to push multiple elements using a single command call just specifying multiple arguments at the end of the command. Elements are inserted one after the other to the head of the list, from the leftmost element to the rightmost element. So for instance `client.lpush('mylist', 'a', 'b', 'c')` will result into a list containing `c` as first element, `b` as second element and `a` as third element.

Note: **Time complexity**: O(1)

---

lpushx(*key*, **values*)
Insert values at the head of an existing list.

Parameters

- **key** (str, bytes) – The list's key
- **values** – One or more positional arguments to insert at the beginning of the list. Each value is inserted at the beginning of the list individually (see discussion below).

Returns the length of the list after push operations, zero if *key* does not refer to a list

Return type int

Raises *TRedisException*

This method inserts *values* at the head of the list stored at *key*, only if *key* already exists and holds a list. In contrary to *lpush()*, no operation will be performed when key does not yet exist.

Note: **Time complexity**: O(1)

---

lrange(*key*, *start*, *end*)
Returns the specified elements of the list stored at key.

Parameters

- **key** (str, bytes) – The list's key
- **start** (*int*) – zero-based index to start retrieving elements from
- **end** (*int*) – zero-based index at which to stop retrieving elements

Return type list

Raises *TRedisException*

The offsets start and stop are zero-based indexes, with 0 being the first element of the list (the head of the list), 1 being the next element and so on.

These offsets can also be negative numbers indicating offsets starting at the end of the list. For example, -1 is the last element of the list, -2 the penultimate, and so on.

Note that if you have a list of numbers from 0 to 100, `lrange(key, 0, 10)` will return 11 elements, that is, the rightmost item is included. This may or may not be consistent with behavior of range-related functions in your programming language of choice (think Ruby's `Range.new`, `Array#slice` or Python's `range()` function).

Out of range indexes will not produce an error. If start is larger than the end of the list, an empty list is returned. If stop is larger than the actual end of the list, Redis will treat it like the last element of the list.

Note: **Time complexity** `O(S+N)` where `S` is the distance of start offset from `HEAD` for small lists, from nearest end (`HEAD` or `TAIL`) for large lists; and `N` is the number of elements in the specified range.

---

ltrim(*key*, *start*, *stop*)

Crop a list to the specified range.

Parameters

- **key** (str, bytes) – The list's key
- **start** (*int*) – zero-based index to first element to retain
- **stop** (*int*) – zero-based index of the last element to retain

Returns did the operation succeed?

Return type bool

Raises *TRedisException*

Trim an existing list so that it will contain only the specified range of elements specified.

Both *start* and *stop* are zero-based indexes, where 0 is the first element of the list (the head), 1 the next element and so on. For example: `ltrim('foobar', 0, 2)` will modify the list stored at `foobar` so that only the first three elements of the list will remain.

start and *stop* can also be negative numbers indicating offsets from the end of the list, where -1 is the last element of the list, -2 the penultimate element and so on.

Out of range indexes will not produce an error: if *start* is larger than the *end* of the list, or *start* > *end*, the result will be an empty list (which causes *key* to be removed). If *end* is larger than the end of the list, Redis will treat it like the last element of the list.

A common use of LTRIM is together with LPUSH / RPUSH. For example:

```
client.lpush('mylist', 'somelement')
client.ltrim('mylist', 0, 99)
```

This pair of commands will push a new element on the list, while making sure that the list will not grow larger than 100 elements. This is very useful when using Redis to store logs for example. It is important to note that when used in this way LTRIM is an O(1) operation because in the average case just one element is removed from the tail of the list.

Note: Time complexity: `O(N)` where *N* is the number of elements to be removed by the operation.

---

mget (**keys*)

Returns the values of all specified keys. For every key that does not hold a string value or does not exist, the special value nil is returned. Because of this, the operation never fails.

New in version 0.2.0.

Note: **Time complexity**: `O(N)` where `N` is the number of keys to retrieve.

Parameters **keys** (`str`, `bytes`) – One or more keys as keyword arguments to the function

Return type list

Raises *RedisError*

migrate (*host*, *port*, *key*, *destination_db*, *timeout*, *copy=False*, *replace=False*)

Atomically transfer a key from a source Redis instance to a destination Redis instance. On success the key is deleted from the original instance and is guaranteed to exist in the target instance.

The command is atomic and blocks the two instances for the time required to transfer the key, at any given time the key will appear to exist in a given instance or in the other instance, unless a timeout error occurs.

Note: **Time complexity**: This command actually executes a DUMP+DEL in the source instance, and a RESTORE in the target instance. See the pages of these commands for time complexity. Also an `O(N)` data transfer between the two instances is performed.

Parameters

- **host** (*bytes, str*) – The host to migrate the key to
- **port** (*int*) – The port to connect on
- **key** (*bytes, str*) – The key to migrate
- **destination_db** (*int*) – The database number to select
- **timeout** (*int*) – The maximum idle time in milliseconds
- **copy** (*bool*) – Do not remove the key from the local instance
- **replace** (*bool*) – Replace existing key on the remote instance

Return type bool

Raises *RedisError*

move (*key*, *db*)

Move key from the currently selected database (see *select()*) to the specified destination database. When key already exists in the destination database, or it does not exist in the source database, it does nothing. It is possible to use *move()* as a locking primitive because of this.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to move
- **db** (*int*) – The database number

Return type bool

Raises *RedisError*

mset(*mapping*)

Sets the given keys to their respective values. *mset()* replaces existing values with new values, just as regular *set()*. See *msetnx()* if you don't want to overwrite existing values.

mset() is atomic, so all given keys are set at once. It is not possible for clients to see that some of the keys were updated while others are unchanged.

New in version 0.2.0.

Note: **Time complexity**: `O(N)` where `N` is the number of keys to set.

Parameters **mapping** (*dict*) – A mapping of key/value pairs to set

Return type bool

Raises *RedisError*

msetnx(*mapping*)

Sets the given keys to their respective values. *msetnx()* will not perform any operation at all even if just a single key already exists.

Because of this semantic *msetnx()* can be used in order to set different keys representing different fields of an unique logic object in a way that ensures that either all the fields or none at all are set.

msetnx() is atomic, so all given keys are set at once. It is not possible for clients to see that some of the keys were updated while others are unchanged.

New in version 0.2.0.

Note: **Time complexity**: `O(N)` where `N` is the number of keys to set.

Parameters **mapping** (*dict*) – A mapping of key/value pairs to set

Return type bool

Raises *RedisError*

object_encoding(*key*)

Return the kind of internal representation used in order to store the value associated with a key

Note: **Time complexity**: `O(1)`

Parameters **key** (str, bytes) – The key to get the encoding for

Return type bytes

Raises *RedisError*

object_idle_time(*key*)
Return the number of seconds since the object stored at the specified key is idle (not requested by read or write operations). While the value is returned in seconds the actual resolution of this timer is 10 seconds, but may vary in future implementations of Redis.

Note: **Time complexity**: O(1)

Parameters **key** (str, bytes) – The key to get the idle time for

Return type int

Raises *RedisError*

object_refcount(*key*)
Return the number of references of the value associated with the specified key. This command is mainly useful for debugging.

Note: **Time complexity**: O(1)

Parameters **key** (str, bytes) – The key to get the refcount for

Return type int

Raises *RedisError*

persist(*key*)
Remove the existing timeout on key, turning the key from volatile (a key with an expire set) to persistent (a key that will never expire as no timeout is associated).

Note: **Time complexity**: O(1)

Parameters **key** (str, bytes) – The key to move

Return type bool

Raises *RedisError*

pexpire(*key*, *timeout*)
This command works exactly like *pexpire()* but the time to live of the key is specified in milliseconds instead of seconds.

Note: **Time complexity**: O(1)

Parameters

- **key** (str, bytes) – The key to set an expiration for

- **timeout** (*int*) – The number of milliseconds to set the timeout to

Return type bool

Raises *RedisError*

pexpireat(*key*, *timestamp*)

pexpireat() has the same effect and semantic as *expireat()*, but the Unix time at which the key will expire is specified in milliseconds instead of seconds.

Note: Time complexity: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to set an expiration for
- **timestamp** (*int*) – The expiration UNIX epoch value in milliseconds

Return type bool

Raises *RedisError*

pfadd(*key*, **elements*)

Adds all the element arguments to the HyperLogLog data structure stored at the variable name specified as first argument.

As a side effect of this command the HyperLogLog internals may be updated to reflect a different estimation of the number of unique items added so far (the cardinality of the set).

If the approximated cardinality estimated by the HyperLogLog changed after executing the command, *pfadd()* returns `1`, otherwise `0` is returned. The command automatically creates an empty HyperLogLog structure (that is, a Redis String of a specified length and with a given encoding) if the specified key does not exist.

To call the command without elements but just the variable name is valid, this will result into no operation performed if the variable already exists, or just the creation of the data structure if the key does not exist (in the latter case `1` is returned).

For an introduction to HyperLogLog data structure check *pfcount()*.

New in version 0.2.0.

Note: Time complexity: `O(1)` to add every element.

Parameters

- **key** (`str`, `bytes`) – The key to add the elements to
- **elements** (`str`, `bytes`) – One or more elements to add

Return type bool

Raises *RedisError*

pfcount(**keys*)

When called with a single key, returns the approximated cardinality computed by the HyperLogLog data structure stored at the specified variable, which is `0` if the variable does not exist.

When called with multiple keys, returns the approximated cardinality of the union of the HyperLogLogs passed, by internally merging the HyperLogLogs stored at the provided keys into a temporary HyperLogLog.

The HyperLogLog data structure can be used in order to count unique elements in a set using just a small constant amount of memory, specifically 12k bytes for every HyperLogLog (plus a few bytes for the key itself).

The returned cardinality of the observed set is not exact, but approximated with a standard error of 0.81%.

For example in order to take the count of all the unique search queries performed in a day, a program needs to call `pfcount()` every time a query is processed. The estimated number of unique queries can be retrieved with `pfcount()` at any time.

Note: as a side effect of calling this function, it is possible that the HyperLogLog is modified, since the last 8 bytes encode the latest computed cardinality for caching purposes. So `pfcount()` is technically a write command.

New in version 0.2.0.

Note: **Time complexity**: `O(1)` with every small average constant times when called with a single key. `O(N)` with `N` being the number of keys, and much bigger constant times, when called with multiple keys.

Parameters **keys** (`str`, `bytes`) – One or more keys

Return type int

Returns The approximated number of unique elements observed

Raises *RedisError*

pfmerge(*dest_key*, **keys*)

Merge multiple HyperLogLog values into an unique value that will approximate the cardinality of the union of the observed Sets of the source HyperLogLog structures.

The computed merged HyperLogLog is set to the destination variable, which is created if does not exist (defaulting to an empty HyperLogLog).

New in version 0.2.0.

Note: **Time complexity**: `O(N)` to merge `N` HyperLogLogs, but with high constant times.

Parameters

- **dest_key** (`str`, `bytes`) – The destination key
- **keys** (`str`, `bytes`) – One or more keys

Return type bool

Raises *RedisError*

ping()

Returns `PONG` if no argument is provided, otherwise return a copy of the argument as a bulk. This command is often used to test if a connection is still alive, or to measure latency.

If the client is subscribed to a channel or a pattern, it will instead return a multi-bulk with a `pong` in the first position and an empty bulk in the second position, unless an argument is provided in which case it returns a copy of the argument.

Return type bytes

Raises *RedisError*

psetex(*key*, *milliseconds*, *value*)

psetex() works exactly like *psetex()* with the sole difference that the expire time is specified in milliseconds instead of seconds.

New in version 0.2.0.

Note: Time complexity: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to set
- **milliseconds** (*int*) – Number of milliseconds for TTL
- **value** (`str`, `bytes`) – The value to set

Return type bool

Raises *RedisError*

pttl(*key*)

Like *ttl()* this command returns the remaining time to live of a key that has an expire set, with the sole difference that *ttl()* returns the amount of remaining time in seconds while *pttl()* returns it in milliseconds.

In Redis 2.6 or older the command returns `-1` if the key does not exist or if the key exist but has no associated expire.

Starting with Redis 2.8 the return value in case of error changed:

- The command returns `-2` if the key does not exist.
- The command returns `-1` if the key exists but has no associated expire.

Note: Time complexity: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to get the PTTL for

Return type int

Raises *RedisError*

quit()

Ask the server to close the connection. The connection is closed as soon as all pending replies have been written to the client.

Return type bool

Raises *RedisError*

randomkey()
Return a random key from the currently selected database.

Note: Time complexity: O(1)

Return type bytes

Raises *RedisError*

ready
Indicates that the client is connected to the Redis server or cluster and is ready for use.

Return type bool

rename(*key*, *new_key*)
Renames key to new_key. It returns an error when the source and destination names are the same, or when key does not exist. If new_key already exists it is overwritten, when this happens *rename()* executes an implicit *delete()* operation, so if the deleted key contains a very big value it may cause high latency even if *rename()* itself is usually a constant-time operation.

Note: Time complexity: O(1)

Parameters

- **key** (str, bytes) – The key to rename
- **new_key** (str, bytes) – The key to rename it to

Return type bool

Raises *RedisError*

renamenx(*key*, *new_key*)
Renames key to new_key if new_key does not yet exist. It returns an error under the same conditions as *rename()*.

Note: Time complexity: O(1)

Parameters

- **key** (str, bytes) – The key to rename
- **new_key** (str, bytes) – The key to rename it to

Return type bool

Raises *RedisError*

restore(*key*, *ttl*, *value*, *replace=False*)
Create a key associated with a value that is obtained by deserializing the provided serialized value (obtained via *dump()*).

If ttl is 0 the key is created without any expire, otherwise the specified expire time (in milliseconds) is set.

restore() will return a `Target key name is busy` error when key already exists unless you use the *restore()* modifier (Redis 3.0 or greater).

restore() checks the RDB version and data checksum. If they don't match an error is returned.

Note: Time complexity: `O(1)` to create the new key and additional `O(N*M)` to reconstruct the serialized value, where `N` is the number of Redis objects composing the value and `M` their average size. For small string values the time complexity is thus `O(1)+O(1*M)` where `M` is small, so simply `O(1)`. However for sorted set values the complexity is `O(N*M*log(N))` because inserting values into sorted sets is `O(log(N))`.

Parameters

- **key** (`str`, `bytes`) – The key to get the TTL for
- **ttl** (*int*) – The number of seconds to set the timeout to
- **value** (`str`, `bytes`) – The value to restore to the key
- **replace** (*bool*) – Replace a pre-existing key

Return type bool

Raises *RedisError*

rpop(*key*)

Removes and returns the last element of the list stored at key.

Parameters **key** (`str`, `bytes`) – The list's key

Returns the length of the list after push operations or zero if *key* does not refer to a list

Returns the element at the tail of the list, `None` if the list does not exist

Return type int

Raises *TRedisException*

rpush(*key*, **values*)

Insert all the specified values at the tail of the list stored at key.

Parameters

- **key** (`str`, `bytes`) – The list's key
- **values** – One or more positional arguments to insert at the tail of the list.

Returns the length of the list after push operations

Return type int

Raises *TRedisException*

If *key* does not exist, it is created as empty list before performing the push operation. When *key* holds a value that is not a list, an error is returned.

It is possible to push multiple elements using a single command call just specifying multiple arguments at the end of the command. Elements are inserted one after the other to the tail of the list, from the leftmost element to the rightmost element. So for instance the command `client.rpush('mylist', 'a', 'b', 'c')` will result in a list containing `a` as first element, `b` as second element and `c` as third element.

Note: **Time complexity**: `O(1)`

rpushx(*key*, **values*)
Insert values at the tail of an existing list.

Parameters

- **key** (`str`, `bytes`) – The list's key
- **values** – One or more positional arguments to insert at the tail of the list.

Returns the length of the list after push operations or zero if *key* does not refer to a list

Return type int

Raises *TRedisException*

This method inserts value at the tail of the list stored at *key*, only if *key* already exists and holds a list. In contrary to method:.*rpush*, no operation will be performed when *key* does not yet exist.

Note: **Time complexity**: `O(1)`

sadd(*key*, **members*)
Add the specified members to the set stored at key. Specified members that are already a member of this set are ignored. If key does not exist, a new set is created before adding the specified members.

An error is returned when the value stored at key is not a set.

Returns `True` if all requested members are added. If more than one member is passed in and not all members are added, the number of added members is returned.

Note: **Time complexity**: `O(N)` where `N` is the number of members to be added.

Parameters

- **key** (`str`, `bytes`) – The key of the set
- **members** – One or more positional arguments to add to the set

Returns Number of items added to the set

Return type bool, int

scan(*cursor=0*, *pattern=None*, *count=None*)
The *scan()* command and the closely related commands *sscan()*, `hscan()` and `zscan()` are used in order to incrementally iterate over a collection of elements.

- *scan()* iterates the set of keys in the currently selected Redis database.
- *sscan()* iterates elements of Sets types.
- `hscan()` iterates fields of Hash types and their associated values.
- `zscan()` iterates elements of Sorted Set types and their associated scores.

Basic usage

scan() is a cursor based iterator. This means that at every call of the command, the server returns an updated cursor that the user needs to use as the cursor argument in the next call.

An iteration starts when the cursor is set to `0`, and terminates when the cursor returned by the server is `0`.

For more information on *`scan()`*, visit the Redis docs on scan.

Note: Time complexity: `O(1)` for every call. `O(N)` for a complete iteration, including enough command calls for the cursor to return back to `0`. `N` is the number of elements inside the collection.

Parameters

- **`cursor`** (*`int`*) – The server specified cursor value or `0`
- **`pattern`** (`str`, `bytes`) – An optional pattern to apply for key matching
- **`count`** (*`int`*) – An optional amount of work to perform in the scan

Return type int, list

Returns A tuple containing the cursor and the list of keys

Raises *`RedisError`*

`scard`(*key*)

Returns the set cardinality (number of elements) of the set stored at key.

Note: Time complexity: `O(1)`

Parameters **`key`** (`str`, `bytes`) – The key of the set

Return type int

Raises *`RedisError`*

`script_exists`(**hashes*)

Returns information about the existence of the scripts in the script cache.

This command accepts one or more SHA1 digests and returns a list of ones or zeros to signal if the scripts are already defined or not inside the script cache. This can be useful before a pipelining operation to ensure that scripts are loaded (and if not, to load them using *`script_load()`*) so that the pipelining operation can be performed solely using *`evalsha()`* instead of *`eval()`* to save bandwidth.

Please refer to the *`eval()`* documentation for detailed information about Redis Lua scripting.

Note: Time complexity: `O(N)` with `N` being the number of scripts to check (so checking a single script is an `O(1)` operation).

Parameters **`hashes`** (*`str`*) – One or more sha1 hashes to check for in the cache

Return type list

Returns Returns a list of `1` or `0` indicating if the specified script(s) exist in the cache.

`script_flush`()

Flush the Lua scripts cache.

Please refer to the *`eval()`* documentation for detailed information about Redis Lua scripting.

Note: **Time complexity**: `O(N)` with `N` being the number of scripts in cache

Return type bool

script_kill()
Kills the currently executing Lua script, assuming no write operation was yet performed by the script.

This command is mainly useful to kill a script that is running for too much time(for instance because it entered an infinite loop because of a bug). The script will be killed and the client currently blocked into *eval()* will see the command returning with an error.

If the script already performed write operations it can not be killed in this way because it would violate Lua script atomicity contract. In such a case only SHUTDOWN NOSAVE is able to kill the script, killing the Redis process in an hard way preventing it to persist with half-written information.

Please refer to the *eval()* documentation for detailed information about Redis Lua scripting.

Note: **Time complexity**: `O(1)`

Return type bool

script_load(*script*)
Load a script into the scripts cache, without executing it. After the specified command is loaded into the script cache it will be callable using *evalsha()* with the correct SHA1 digest of the script, exactly like after the first successful invocation of *eval()*.

The script is guaranteed to stay in the script cache forever (unless *script_flush()* is called).

The command works in the same way even if the script was already present in the script cache.

Please refer to the *eval()* documentation for detailed information about Redis Lua scripting.

Note: **Time complexity**: `O(N)` with `N` being the length in bytes of the script body.

Parameters **script** (*str*) – The script to load into the script cache

Returns str

sdiff(**keys*)
Returns the members of the set resulting from the difference between the first set and all the successive sets.

For example:

```
key1 = {a,b,c,d}
key2 = {c}
key3 = {a,c,e}
SDIFF key1 key2 key3 = {b,d}
```

Keys that do not exist are considered to be empty sets.

Note: **Time complexity**: `O(N)` where `N` is the total number of elements in all given sets.

Parameters **keys** (str, bytes) – Two or more set keys as positional arguments

Return type list

Raises *RedisError*

sdiffstore(*destination*, **keys*)
This command is equal to *sdiff()*, but instead of returning the resulting set, it is stored in destination.

If destination already exists, it is overwritten.

Note: **Time complexity**: O(N) where N is the total number of elements in all given sets.

Parameters

- **destination** (str, bytes) – The set to store the diff into
- **keys** (str, bytes) – One or more set keys as positional arguments

Return type int

Raises *RedisError*

select(*index=0*)
Select the DB with having the specified zero-based numeric index. New connections always use DB 0.

Parameters **index** (*int*) – The database to select

Return type bool

Raises *RedisError*

Raises *InvalidClusterCommand*

set(*key*, *value*, *ex=None*, *px=None*, *nx=False*, *xx=False*)
Set key to hold the string value. If key already holds a value, it is overwritten, regardless of its type. Any previous time to live associated with the key is discarded on successful *set()* operation.

If the value is not one of str, bytes, or int, a ValueError will be raised.

Note: **Time complexity**: O(1)

Parameters

- **key** (str, bytes) – The key to remove
- **value** (str, bytes, int) – The value to set
- **ex** (*int*) – Set the specified expire time, in seconds
- **px** (*int*) – Set the specified expire time, in milliseconds
- **nx** (*bool*) – Only set the key if it does not already exist
- **xx** (*bool*) – Only set the key if it already exist

Return type bool

Raises *RedisError*

Raises ValueError

setbit (*key*, *offset*, *bit*)
Sets or clears the bit at offset in the string value stored at key.

The bit is either set or cleared depending on value, which can be either 0 or 1. When key does not exist, a new string value is created. The string is grown to make sure it can hold a bit at offset. The offset argument is required to be greater than or equal to 0, and smaller than 2^{32} (this limits bitmaps to 512MB). When the string at key is grown, added bits are set to 0.

> **Warning:** When setting the last possible bit (offset equal to 2^{32} -1) and the string value stored at key does not yet hold a string value, or holds a small string value, Redis needs to allocate all intermediate memory which can block the server for some time. On a 2010 MacBook Pro, setting bit number 2^{32} -1 (512MB allocation) takes ~300ms, setting bit number 2^{30} -1 (128MB allocation) takes ~80ms, setting bit number 2^{28} -1 (32MB allocation) takes ~30ms and setting bit number 2^{26} -1 (8MB allocation) takes ~8ms. Note that once this first allocation is done, subsequent calls to *setbit()* for the same key will not have the allocation overhead.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to get the bit from
- **offset** (*int*) – The bit offset to fetch the bit from
- **bit** (*int*) – The value (`0` or `1`) to set for the bit

Return type int

Raises *RedisError*

setex (*key*, *seconds*, *value*)
Set key to hold the string value and set key to timeout after a given number of seconds.

setex() is atomic, and can be reproduced by using *set()* and *expire()* inside an `multi()` / `exec()` block. It is provided as a faster alternative to the given sequence of operations, because this operation is very common when Redis is used as a cache.

An error is returned when seconds is invalid.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to set
- **seconds** (*int*) – Number of seconds for TTL
- **value** (`str`, `bytes`) – The value to set

Return type bool

Raises *RedisError*

setnx(*key*, *value*)

Set key to hold string value if key does not exist. In that case, it is equal to *setnx()*. When key already holds a value, no operation is performed. *setnx()* is short for "SET if Not eXists".

New in version 0.2.0.

Note: **Time complexity**: O(1)

Parameters

- **key** (str, bytes) – The key to set
- **value** (str, bytes, int) – The value to set

Return type bool

Raises *RedisError*

setrange(*key*, *offset*, *value*)

Overwrites part of the string stored at key, starting at the specified offset, for the entire length of value. If the offset is larger than the current length of the string at key, the string is padded with zero-bytes to make offset fit. Non-existing keys are considered as empty strings, so this command will make sure it holds a string large enough to be able to set value at offset.

Note: The maximum offset that you can set is 2^{29} -1 (536870911), as Redis Strings are limited to 512 megabytes. If you need to grow beyond this size, you can use multiple keys.

Warning: When setting the last possible byte and the string value stored at key does not yet hold a string value, or holds a small string value, Redis needs to allocate all intermediate memory which can block the server for some time. On a 2010 MacBook Pro, setting byte number 536870911 (512MB allocation) takes ~300ms, setting byte number 134217728 (128MB allocation) takes ~80ms, setting bit number 33554432 (32MB allocation) takes ~30ms and setting bit number 8388608 (8MB allocation) takes ~8ms. Note that once this first allocation is done, subsequent calls to *setrange()* for the same key will not have the allocation overhead.

New in version 0.2.0.

Note: **Time complexity**: O(1), not counting the time taken to copy the new string in place. Usually, this string is very small so the amortized complexity is O(1). Otherwise, complexity is O(M) with M being the length of the value argument.

Parameters

- **key** (str, bytes) – The key to get the bit from
- **value** (str, bytes, int) – The value to set

Returns The length of the string after it was modified by the command

Return type int

Raises *RedisError*

sinter(**keys*)
Returns the members of the set resulting from the intersection of all the given sets.

For example:

```
key1 = {a,b,c,d}
key2 = {c}
key3 = {a,c,e}
SINTER key1 key2 key3 = {c}
```

Keys that do not exist are considered to be empty sets. With one of the keys being an empty set, the resulting set is also empty (since set intersection with an empty set always results in an empty set).

Note: **Time complexity**: `O(N*M)` worst case where `N` is the cardinality of the smallest set and `M` is the number of sets.

Parameters **keys** (str, bytes) – Two or more set keys as positional arguments

Return type list

Raises *RedisError*

sinterstore(*destination*, **keys*)
This command is equal to *sinter()*, but instead of returning the resulting set, it is stored in destination.

If destination already exists, it is overwritten.

Note: **Time complexity**: `O(N*M)` worst case where `N` is the cardinality of the smallest set and `M` is the number of sets.

Parameters

- **destination** (str, bytes) – The set to store the intersection into
- **keys** (str, bytes) – One or more set keys as positional arguments

Return type int

Raises *RedisError*

sismember(*key*, *member*)
Returns `True` if `member` is a member of the set stored at key.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (str, bytes) – The key of the set to check for membership in
- **member** (str, bytes) – The value to check for set membership with

Return type bool

Raises *RedisError*

smembers(*key*)
Returns all the members of the set value stored at key.

This has the same effect as running *sinter()* with one argument key.

Note: **Time complexity**: O(N) where N is the set cardinality.

Parameters **key** (str, bytes) – The key of the set to return the members from

Return type list

Raises *RedisError*

smove(*source*, *destination*, *member*)
Move member from the set at source to the set at destination. This operation is atomic. In every given moment the element will appear to be a member of source or destination for other clients.

If the source set does not exist or does not contain the specified element, no operation is performed and False is returned. Otherwise, the element is removed from the source set and added to the destination set. When the specified element already exists in the destination set, it is only removed from the source set.

An error is returned if source or destination does not hold a set value.

Note: **Time complexity**: O(1)

Parameters

- **source** (str, bytes) – The source set key
- **destination** (str, bytes) – The destination set key
- **member** (str, bytes) – The member value to move

Return type bool

Raises *RedisError*

sort(*key*, *by=None*, *external=None*, *offset=0*, *limit=None*, *order=None*, *alpha=False*, *store_as=None*)
Returns or stores the elements contained in the list, set or sorted set at key. By default, sorting is numeric and elements are compared by their value interpreted as double precision floating point number.

The external parameter is used to specify the *GET <http://redis.io/commands/sort#retrieving-external-keys>_* parameter for retrieving external keys. It can be a single string or a list of strings.

Note: **Time complexity**: O(N+M*log(M)) where N is the number of elements in the list or set to sort, and M the number of returned elements. When the elements are not sorted, complexity is currently O(N) as there is a copy step that will be avoided in next releases.

Parameters

- **key** (str, bytes) – The key to get the refcount for
- **by** (str, bytes) – The optional pattern for external sorting keys
- **external** (str, bytes, list) – Pattern or list of patterns to return external keys

- **offset** (*int*) – The starting offset when using limit
- **limit** (*int*) – The number of elements to return
- **order** (str, bytes) – The sort order - one of ASC or DESC
- **alpha** (*bool*) – Sort the results lexicographically
- **store_as** (str, bytes, None) – When specified, the key to store the results as

Return type listlint

Raises *RedisError*

Raises ValueError

spop(*key*, *count=None*)

Removes and returns one or more random elements from the set value store at key.

This operation is similar to *srandmember()*, that returns one or more random elements from a set but does not remove it.

The count argument will be available in a later version and is not available in 2.6, 2.8, 3.0

Redis 3.2 will be the first version where an optional count argument can be passed to *spop()* in order to retrieve multiple elements in a single call. The implementation is already available in the unstable branch.

Note: **Time complexity**: Without the count argument O(1), otherwise O(N) where N is the absolute value of the passed count.

Parameters

- **key** (str, bytes) – The key to get one or more random members from
- **count** (*int*) – The number of members to return

Return type bytes, list

Raises *RedisError*

srandmember(*key*, *count=None*)

When called with just the key argument, return a random element from the set value stored at key.

Starting from Redis version 2.6, when called with the additional count argument, return an array of count distinct elements if count is positive. If called with a negative count the behavior changes and the command is allowed to return the same element multiple times. In this case the number of returned elements is the absolute value of the specified count.

When called with just the key argument, the operation is similar to *spop()*, however while *spop()* also removes the randomly selected element from the set, *srandmember()* will just return a random element without altering the original set in any way.

Note: **Time complexity**: Without the count argument O(1), otherwise O(N) where N is the absolute value of the passed count.

Parameters

- **key** (str, bytes) – The key to get one or more random members from
- **count** (*int*) – The number of members to return

Return type bytes, list

Raises *RedisError*

srem(*key*, **members*)

Remove the specified members from the set stored at key. Specified members that are not a member of this set are ignored. If key does not exist, it is treated as an empty set and this command returns `0`.

An error is returned when the value stored at key is not a set.

Returns `True` if all requested members are removed. If more than one member is passed in and not all members are removed, the number of removed members is returned.

Note: Time complexity: `O(N)` where `N` is the number of members to be removed.

Parameters

- **key** (`str`, `bytes`) – The key to remove the member from
- **members** (*mixed*) – One or more member values to remove

Return type bool, int

Raises *RedisError*

sscan(*key*, *cursor=0*, *pattern=None*, *count=None*)

The *sscan()* command and the closely related commands *scan()*, `hscan()` and `zscan()` are used in order to incrementally iterate over a collection of elements.

- *scan()* iterates the set of keys in the currently selected Redis database.
- *sscan()* iterates elements of Sets types.
- `hscan()` iterates fields of Hash types and their associated values.
- `zscan()` iterates elements of Sorted Set types and their associated scores.

Basic usage

sscan() is a cursor based iterator. This means that at every call of the command, the server returns an updated cursor that the user needs to use as the cursor argument in the next call.

An iteration starts when the cursor is set to `0`, and terminates when the cursor returned by the server is `0`.

For more information on *scan()*, visit the Redis docs on scan.

Note: Time complexity: `O(1)` for every call. `O(N)` for a complete iteration, including enough command calls for the cursor to return back to `0`. `N` is the number of elements inside the collection.

Parameters

- **key** (`str`, `bytes`) – The key to scan
- **cursor** (*int*) – The server specified cursor value or `0`
- **pattern** (`str`, `bytes`) – An optional pattern to apply for key matching
- **count** (*int*) – An optional amount of work to perform in the scan

Return type int, list

Returns A tuple containing the cursor and the list of set items

Raises *RedisError*

strlen(*key*)

Returns the length of the string value stored at key. An error is returned when key holds a non-string value

New in version 0.2.0.

Note: Time complexity: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to set

Returns The length of the string at key, or 0 when key does not exist

Return type int

Raises *RedisError*

sunion(**keys*)

Returns the members of the set resulting from the union of all the given sets.

For example:

```
key1 = {a,b,c,d}
key2 = {c}
key3 = {a,c,e}
SUNION key1 key2 key3 = {a,b,c,d,e}
```

Note: Time complexity: `O(N)` where `N` is the total number of elements in all given sets.

Keys that do not exist are considered to be empty sets.

Parameters **keys** (`str`, `bytes`) – Two or more set keys as positional arguments

Return type list

Raises *RedisError*

sunionstore(*destination*, **keys*)

This command is equal to *sunion()*, but instead of returning the resulting set, it is stored in destination.

If destination already exists, it is overwritten.

Note: Time complexity: `O(N)` where `N` is the total number of elements in all given sets.

Parameters

- **destination** (`str`, `bytes`) – The set to store the union into
- **keys** (`str`, `bytes`) – One or more set keys as positional arguments

Return type int

Raises *RedisError*

time()
Retrieve the current time from the redis server.

Return type float

Raises *RedisError*

ttl(*key*)
Returns the remaining time to live of a key that has a timeout. This introspection capability allows a Redis client to check how many seconds a given key will continue to be part of the dataset.

Note: Time complexity: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to get the TTL for

Return type int

Raises *RedisError*

type(*key*)
Returns the string representation of the type of the value stored at key. The different types that can be returned are: `string`, `list`, `set`, `zset`, and `hash`.

Note: Time complexity: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to get the type for

Return type bytes

Raises *RedisError*

wait(*num_slaves*, *timeout=0*)
his command blocks the current client until all the previous write commands are successfully transferred and acknowledged by at least the specified number of slaves. If the timeout, specified in milliseconds, is reached, the command returns even if the specified number of slaves were not yet reached.

The command will always return the number of slaves that acknowledged the write commands sent before the *wait()* command, both in the case where the specified number of slaves are reached, or when the timeout is reached.

Note: Time complexity: `O(1)`

Parameters

- **num_slaves** (*int*) – Number of slaves to acknowledge previous writes
- **timeout** (*int*) – Timeout in milliseconds

Return type int

Raises *RedisError*

zadd(*key*, **members*, ***kwargs*)
Adds all the specified members with the specified scores to the sorted set stored at key. It is possible to

specify multiple score / member pairs. If a specified member is already a member of the sorted set, the score is updated and the element reinserted at the right position to ensure the correct ordering.

If key does not exist, a new sorted set with the specified members as sole members is created, like if the sorted set was empty. If the key exists but does not hold a sorted set, an error is returned.

The score values should be the string representation of a double precision floating point number. +inf and -inf values are valid values as well.

Members parameters

`members` could be either: - a single dict where keys correspond to scores and values to elements - multiple strings paired as score then element

```
yield client.zadd('myzset', {'1': 'one', '2': 'two'})
yield client.zadd('myzset', '1', 'one', '2', 'two')
```

ZADD options (Redis 3.0.2 or greater)

ZADD supports a list of options. Options are:

- `xx`: Only update elements that already exist. Never add elements.
- **`nx`: Don't update already existing elements. Always add new** elements.
- **`ch`: Modify the return value from the number of new elements** added, to the total number of elements changed (CH is an abbreviation of changed). Changed elements are new elements added and elements already existing for which the score was updated. So elements specified in the command having the same score as they had in the past are not counted. Note: normally the return value of `ZADD` only counts the number of new elements added.
- **`incr`: When this option is specified `ZADD` acts like** `zincrby()`. Only one score-element pair can be specified in this mode.

Note: **Time complexity**: `O(log(N))` for each item added, where `N` is the number of elements in the sorted set.

Parameters

- **`key`** (`str`, `bytes`) – The key of the sorted set
- **`members`** (`dict`, `str`, `bytes`) – Elements to add
- **`xx`** (*`bool`*) – Only update elements that already exist
- **`nx`** (*`bool`*) – Don't update already existing elements
- **`ch`** (*`bool`*) – Return the number of changed elements
- **`incr`** (*`bool`*) – Increment the score of an element

Return type int, `str`, `bytes`

Returns Number of elements changed, or the new score if incr is set

Raises *`RedisError`*

`zcard`(*key*)

Returns the set cardinality (number of elements) of the sorted set stored at key.

Note: **Time complexity**: `O(1)`

Parameters **key** (`str`, `bytes`) – The key of the set

Return type int

Raises *RedisError*

zrange(*key*, *start=0*, *stop=-1*, *with_scores=False*)

Returns the specified range of elements in the sorted set stored at key. The elements are considered to be ordered from the lowest to the highest score. Lexicographical order is used for elements with equal score.

See *tredis.Client.zrevrange()* when you need the elements ordered from highest to lowest score (and descending lexicographical order for elements with equal score).

Both start and stop are zero-based indexes, where `0` is the first element, `1` is the next element and so on. They can also be negative numbers indicating offsets from the end of the sorted set, with `-1` being the last element of the sorted set, `-2` the penultimate element and so on.

`start` and `stop` are inclusive ranges, so for example `ZRANGE myzset 0 1` will return both the first and the second element of the sorted set.

Out of range indexes will not produce an error. If start is larger than the largest index in the sorted set, or `start > stop`, an empty list is returned. If stop is larger than the end of the sorted set Redis will treat it like it is the last element of the sorted set.

It is possible to pass the `WITHSCORES` option in order to return the scores of the elements together with the elements. The returned list will contain `value1,score1,...,valueN,scoreN` instead of `value1,...,valueN`. Client libraries are free to return a more appropriate data type (suggestion: an array with (value, score) arrays/tuples).

Note: **Time complexity**: `O(log(N)+M)` with `N` being the number of elements in the sorted set and `M` the number of elements returned.

Parameters

- **key** (`str`, `bytes`) – The key of the sorted set
- **start** (*int*) – The starting index of the sorted set
- **stop** (*int*) – The ending index of the sorted set
- **with_scores** (*bool*) – Return the scores with the elements

Return type list

Raises *RedisError*

zrangebyscore(*key*, *min_score*, *max_score*, *with_scores=False*, *offset=0*, *count=0*)

Returns all the elements in the sorted set at key with a score between min and max (including elements with score equal to min or max). The elements are considered to be ordered from low to high scores.

The elements having the same score are returned in lexicographical order (this follows from a property of the sorted set implementation in Redis and does not involve further computation).

The optional `offset` and `count` arguments can be used to only get a range of the matching elements (similar to SELECT LIMIT offset, count in SQL). Keep in mind that if offset is large, the sorted set needs

to be traversed for offset elements before getting to the elements to return, which can add up to O(N) time complexity.

The optional with_scores argument makes the command return both the element and its score, instead of the element alone. This option is available since Redis 2.0.

Exclusive intervals and infinity

min_score and max_score can be -inf and +inf, so that you are not required to know the highest or lowest score in the sorted set to get all elements from or up to a certain score.

By default, the interval specified by min_score and max_score is closed (inclusive). It is possible to specify an open interval (exclusive) by prefixing the score with the character (. For example:

```
ZRANGEBYSCORE zset (1 5
```

Will return all elements with 1 < score <= 5 while:

```
ZRANGEBYSCORE zset (5 (10
```

Will return all the elements with 5 < score < 10 (5 and 10 excluded).

Note: **Time complexity**: O(log(N)+M) with N being the number of elements in the sorted set and M the number of elements being returned. If M is constant (e.g. always asking for the first 10 elements with count), you can consider it O(log(N)).

Parameters

- **key** (str, bytes) – The key of the sorted set
- **min_score** (str, bytes) – Lowest score definition
- **max_score** (str, bytes) – Highest score definition
- **with_scores** (*bool*) – Return elements and scores
- **offset** – The number of elements to skip
- **count** – The number of elements to return

Return type list

Raises *RedisError*

zrem(*key*, **members*)

Removes the specified members from the sorted set stored at key. Non existing members are ignored.

An error is returned when key exists and does not hold a sorted set.

Note: **Time complexity**: O(M*log(N)) with N being the number of elements in the sorted set and M the number of elements to be removed.

Parameters

- **key** (str, bytes) – The key of the sorted set
- **members** (str, bytes) – One or more member values to remove

Return type int

Raises *RedisError*

zremrangebyscore(*key*, *min_score*, *max_score*)
Removes all elements in the sorted set stored at key with a score between min and max.

Intervals are described in *zrangebyscore()*.

Returns the number of elements removed.

Note: **Time complexity**: `O(log(N)+M)` with `N` being the number of elements in the sorted set and M the number of elements removed by the operation.

Parameters

- **key** (`str`, `bytes`) – The key of the sorted set
- **min_score** (`str`, `bytes`) – Lowest score definition
- **max_score** (`str`, `bytes`) – Highest score definition

Return type int

Raises *RedisError*

zrevrange(*key*, *start=0*, *stop=-1*, *with_scores=False*)
Returns the specified range of elements in the sorted set stored at key. The elements are considered to be ordered from the highest to the lowest score. Descending lexicographical order is used for elements with equal score.

Apart from the reversed ordering, *zrevrange()* is similar to *zrange()* .

Note: **Time complexity**: `O(log(N)+M)` with `N` being the number of elements in the sorted set and `M` the number of elements returned.

Parameters

- **key** (`str`, `bytes`) – The key of the sorted set
- **start** (*int*) – The starting index of the sorted set
- **stop** (*int*) – The ending index of the sorted set
- **with_scores** (*bool*) – Return the scores with the elements

Return type list

Raises *RedisError*

zscore(*key*, *member*)
Returns the score of member in the sorted set at key. If member does not exist in the sorted set, or key does not exist None is returned.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key of the set to check for membership in

- **member** (str, bytes) – The value to check for set membership with

Return type str or None

Raises *RedisError*

class tredis.cluster.**ClusterNode**(*id*, *ip*, *port*, *flags*, *master*, *ping_sent*, *pong_recv*, *config_epoch*, *link_state*, *slots*)

tredis.cluster.ClusterNode is a namedtuple that contains the attributes for a single node returned by the CLUSTER NODES command.

Parameters

- **id** (*bytes*) – The node ID
- **ip** (*bytes*) – The IP address of the node
- **port** (*int*) – The node TCP port
- **flags** (*bytes*) – A list of comma separated flags: myself, master, slave, fail?, fail, handshake, noaddr, noflags.
- **master** (*bytes*) – If the node is a slave, and the master is known, the master node ID, otherwise the – character.
- **ping_sent** (*int*) – Milliseconds unix time at which the currently active ping was sent, or zero if there are no pending pings.
- **pong_recv** (*int*) – Milliseconds unix time the last pong was received.
- **config_epoch** (*int*) – The configuration epoch (or version) of the current node (or of the current master if the node is a slave). Each time there is a failover, a new, unique, monotonically increasing configuration epoch is created. If multiple nodes claim to serve the same hash slots, the one with higher configuration epoch wins.
- **link_state** (*bytes*) – The state of the link used for the node-to-node cluster bus. We use this link to communicate with the node. Can be connected or disconnected.
- **slots** (*list(tuple(int, int))*) – A hash slot number or range. There may be up to 16384 entries in total (limit never reached). This is the list of hash slots served by this node. If the entry is just a number, is parsed as such. If it is a range, it is in the form start-end, and means that the node is responsible for all the hash slots from start to end including the start and end values.

class tredis.**RedisClient**(*host='localhost'*, *port=6379*, *db=0*, *on_close=None*, *clustering=False*, *auto_connect=True*)

This is provided for backwards compatibility for versions < 0.7.

Deprecated since version 0.7.

Parameters

- **host** (*str*) – The hostname to connect to
- **port** (*int*) – The port to connect on
- **db** (*int*) – The database number to use
- **on_close** (*method*) – The method to call if the connection is closed
- **clustering** (*bool*) – Toggle the cluster support in the client
- **auto_connect** (*bool*) – Toggle the auto-connect on creation feature

append(*key*, *value*)

If key already exists and is a string, this command appends the value at the end of the string. If key does not exist it is created and set as an empty string, so *append()* will be similar to *set()* in this special case.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`. The amortized time complexity is `O(1)` assuming the appended value is small and the already present value is of any size, since the dynamic string library used by Redis will double the free space available on every reallocation.

Parameters

- **key** (`str`, `bytes`) – The key to get
- **value** (`str`, `bytes`) – The value to append to the key

Returns The length of the string after the append operation

Return type int

Raises *RedisError*

auth(*password*)

Request for authentication in a password-protected Redis server. Redis can be instructed to require a password before allowing clients to execute commands. This is done using the `requirepass` directive in the configuration file.

If the password does not match, an *AuthError* exception will be raised.

Parameters **password** (`str`, `bytes`) – The password to authenticate with

Return type bool

Raises *AuthError*, *RedisError*

bitcount(*key*, *start=None*, *end=None*)

Count the number of set bits (population counting) in a string.

By default all the bytes contained in the string are examined. It is possible to specify the counting operation only in an interval passing the additional arguments start and end.

Like for the *getrange()* command start and end can contain negative values in order to index bytes starting from the end of the string, where `-1` is the last byte, `-2` is the penultimate, and so forth.

Non-existent keys are treated as empty strings, so the command will return zero.

New in version 0.2.0.

Note: **Time complexity**: `O(N)`

Parameters

- **key** (`str`, `bytes`) – The key to get
- **start** (*int*) – The start position to evaluate in the string
- **end** (*int*) – The end position to evaluate in the string

Return type int

Raises *RedisError*, ValueError

bitop (*operation*, *dest_key*, **keys*)

Perform a bitwise operation between multiple keys (containing string values) and store the result in the destination key.

The values for operation can be one of:

- b'AND'
- b'OR'
- b'XOR'
- b'NOT'
- tredis.BITOP_AND or b'&'
- tredis.BITOP_OR or b'|'
- tredis.BITOP_XOR or b'^'
- tredis.BITOP_NOT or b'~'

b'NOT' is special as it only takes an input key, because it performs inversion of bits so it only makes sense as an unary operator.

The result of the operation is always stored at dest_key.

Handling of strings with different lengths

When an operation is performed between strings having different lengths, all the strings shorter than the longest string in the set are treated as if they were zero-padded up to the length of the longest string.

The same holds true for non-existent keys, that are considered as a stream of zero bytes up to the length of the longest string.

New in version 0.2.0.

Note: Time complexity: O(N)

Parameters

- **operation** (*bytes*) – The operation to perform
- **dest_key** (str, bytes) – The key to store the bitwise operation results to
- **keys** (str, bytes) – One or more keys as keyword parameters for the bitwise op

Returns The size of the string stored in the destination key, that is equal to the size of the longest input string.

Return type int

Raises *RedisError*, ValueError

bitpos (*key*, *bit*, *start=None*, *end=None*)

Return the position of the first bit set to 1 or 0 in a string.

The position is returned, thinking of the string as an array of bits from left to right, where the first byte's most significant bit is at position 0, the second byte's most significant bit is at position 8, and so forth.

The same bit position convention is followed by *getbit()* and *setbit()*.

By default, all the bytes contained in the string are examined. It is possible to look for bits only in a specified interval passing the additional arguments start and end (it is possible to just pass start, the operation will assume that the end is the last byte of the string. However there are semantic differences as explained later). 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.

Note that bit positions are returned always as absolute values starting from bit zero even when start and end are used to specify a range.

Like for the *getrange()* command start and end can contain negative values in order to index bytes starting from the end of the string, where `-1` is the last byte, `-2` is the penultimate, and so forth.

Non-existent keys are treated as empty strings.

New in version 0.2.0.

Note: Time complexity: `O(N)`

Parameters

- **key** (`str`, `bytes`) – The key to get
- **bit** (*int*) – The bit value to search for (`1` or `0`)
- **start** (*int*) – The start position to evaluate in the string
- **end** (*int*) – The end position to evaluate in the string

Returns The position of the first bit set to `1` or `0`

Return type int

Raises *RedisError*, `ValueError`

close()

Close any open connections to Redis.

Raises *tredis.exceptions.ConnectionError*

cluster_info()

`CLUSTER INFO` provides `INFO` style information about Redis Cluster vital parameters.

New in version 0.7.0.

Returns A dictionary of current cluster information

Return type dict

Key cluster_state State is ok if the node is able to receive queries. fail if there is at least one hash slot which is unbound (no node associated), in error state (node serving it is flagged with `FAIL` flag), or if the majority of masters can't be reached by this node.

Key cluster_slots_assigned Number of slots which are associated to some node (not unbound). This number should be `16384` for the node to work properly, which means that each hash slot should be mapped to a node.

Key cluster_slots_ok Number of hash slots mapping to a node not in `FAIL` or `PFAIL` state.

Key cluster_slots_pfail Number of hash slots mapping to a node in `PFAIL` state. Note that those hash slots still work correctly, as long as the `PFAIL` state is not promoted to `FAIL` by the failure detection algorithm. `PFAIL` only means that we are currently not able to talk with the node, but may be just a transient error.

Key cluster_slots_fail Number of hash slots mapping to a node in `FAIL` state. If this number is not zero the node is not able to serve queries unless cluster-require-full-coverage is set to no in the configuration.

Key cluster_known_nodes The total number of known nodes in the cluster, including nodes in `HANDSHAKE` state that may not currently be proper members of the cluster.

Key cluster_size The number of master nodes serving at least one hash slot in the cluster.

Key cluster_current_epoch The local Current Epoch variable. This is used in order to create unique increasing version numbers during fail overs.

Key cluster_my_epoch The Config Epoch of the node we are talking with. This is the current configuration version assigned to this node.

Key cluster_stats_messages_sent Number of messages sent via the cluster node-to-node binary bus.

Key cluster_stats_messages_received Number of messages received via the cluster node-to-node binary bus.

Raises *RedisError*

cluster_nodes()

Each node in a Redis Cluster has its view of the current cluster configuration, given by the set of known nodes, the state of the connection we have with such nodes, their flags, properties and assigned slots, and so forth.

`CLUSTER NODES` provides all this information, that is, the current cluster configuration of the node we are contacting, in a serialization format which happens to be exactly the same as the one used by Redis Cluster itself in order to store on disk the cluster state (however the on disk cluster state has a few additional info appended at the end).

Note that normally clients willing to fetch the map between Cluster hash slots and node addresses should use `CLUSTER SLOTS` instead. `CLUSTER NODES`, that provides more information, should be used for administrative tasks, debugging, and configuration inspections. It is also used by `redis-trib` in order to manage a cluster.

New in version 0.7.0.

Return type list(*ClusterNode*)

Raises *RedisError*

connect()

Connect to the Redis server or Cluster.

Return type tornado.concurrent.Future

decr(*key*)

Decrements the number stored at key by one. If the key does not exist, it is set to 0 before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that can not be represented as integer. This operation is limited to 64 bit signed integers.

See *incr()* for extra information on increment/decrement operations.

New in version 0.2.0.

Note: Time complexity: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to decrement

Returns The value of key after the decrement

Return type int

Raises *RedisError*

decrby(*key*, *decrement*)

Decrements the number stored at key by decrement. If the key does not exist, it is set to 0 before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that can not be represented as integer. This operation is limited to 64 bit signed integers.

See *incr()* for extra information on increment/decrement operations.

New in version 0.2.0.

Note: Time complexity: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to decrement
- **decrement** (*int*) – The amount to decrement by

Returns The value of key after the decrement

Return type int

Raises *RedisError*

delete(**keys*)

Removes the specified keys. A key is ignored if it does not exist. Returns `True` if all keys are removed.

Note: Time complexity: `O(N)` where `N` is the number of keys that will be removed. When a key to remove holds a value other than a string, the individual complexity for this key is `O(M)` where `M` is the number of elements in the list, set, sorted set or hash. Removing a single key that holds a string value is `O(1)`.

Parameters **keys** (`str`, `bytes`) – One or more keys to remove

Return type bool

Raises *RedisError*

dump(*key*)

Serialize the value stored at key in a Redis-specific format and return it to the user. The returned value can be synthesized back into a Redis key using the *restore()* command.

The serialization format is opaque and non-standard, however it has a few semantic characteristics:

- It contains a 64-bit checksum that is used to make sure errors will be detected. The *restore()* command makes sure to check the checksum before synthesizing a key using the serialized value.
- Values are encoded in the same format used by RDB.
- An RDB version is encoded inside the serialized value, so that different Redis versions with incompatible RDB formats will refuse to process the serialized value.
- The serialized value does NOT contain expire information. In order to capture the time to live of the current value the *pttl()* command should be used.

If key does not exist `None` is returned.

Note: **Time complexity**: `O(1)` to access the key and additional `O(N*M)` to serialized it, where N is the number of Redis objects composing the value and `M` their average size. For small string values the time complexity is thus `O(1)+O(1*M)` where `M` is small, so simply `O(1)`.

Parameters **key** (`str`, `bytes`) – The key to dump

Return type bytes, None

echo(*message*)

Returns the message that was sent to the Redis server.

Parameters **message** (`str`, `bytes`) – The message to echo

Return type bytes

Raises *RedisError*

eval(*script*, *keys=None*, *args=None*)

eval() and *evalsha()* are used to evaluate scripts using the Lua interpreter built into Redis starting from version 2.6.0.

The first argument of EVAL is a Lua 5.1 script. The script does not need to define a Lua function (and should not). It is just a Lua program that will run in the context of the Redis server.

Note: **Time complexity**: Depends on the script that is executed.

Parameters

- **script** (*str*) – The Lua script to execute
- **keys** (*list*) – A list of keys to pass into the script
- **args** (*list*) – A list of args to pass into the script

Returns mixed

evalsha(*sha1*, *keys=None*, *args=None*)

Evaluates a script cached on the server side by its SHA1 digest. Scripts are cached on the server side using the *script_load()* command. The command is otherwise identical to *eval()*.

Note: **Time complexity**: Depends on the script that is executed.

Parameters

- **sha1** (*str*) – The sha1 hash of the script to execute
- **keys** (*list*) – A list of keys to pass into the script
- **args** (*list*) – A list of args to pass into the script

Returns mixed

exists(*key*)
Returns `True` if the key exists.

Note: **Time complexity**: `O(1)`

Command Type: String

Parameters **key** (`str`, `bytes`) – One or more keys to check for

Return type bool

Raises *RedisError*

expire(*key*, *timeout*)
Set a timeout on key. After the timeout has expired, the key will automatically be deleted. A key with an associated timeout is often said to be volatile in Redis terminology.

The timeout is cleared only when the key is removed using the *delete()* method or overwritten using the *set()* or *getset()* methods. This means that all the operations that conceptually alter the value stored at the key without replacing it with a new one will leave the timeout untouched. For instance, incrementing the value of a key with *incr()*, pushing a new value into a list with *lpush()*, or altering the field value of a hash with *hset()* are all operations that will leave the timeout untouched.

The timeout can also be cleared, turning the key back into a persistent key, using the *persist()* method.

If a key is renamed with *rename()*, the associated time to live is transferred to the new key name.

If a key is overwritten by *rename()*, like in the case of an existing key `Key_A` that is overwritten by a call like `client.rename(Key_B, Key_A)` it does not matter if the original `Key_A` had a timeout associated or not, the new key `Key_A` will inherit all the characteristics of `Key_B`.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to set an expiration for
- **timeout** (*int*) – The number of seconds to set the timeout to

Return type bool

Raises *RedisError*

expireat(*key*, *timestamp*)
expireat() has the same effect and semantic as *expire()*, but instead of specifying the number of seconds representing the TTL (time to live), it takes an absolute Unix timestamp (seconds since January 1, 1970).

Please for the specific semantics of the command refer to the documentation of *expire()*.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to set an expiration for
- **timestamp** (*int*) – The UNIX epoch value for the expiration

Return type bool

Raises *RedisError*

get(*key*)
Get the value of key. If the key does not exist the special value None is returned. An error is returned if the value stored at key is not a string, because *get()* only handles string values.

Note: Time complexity: O(1)

Parameters **key** (str, bytes) – The key to get

Return type bytes|None

Raises *RedisError*

getbit(*key*, *offset*)
Returns the bit value at offset in the string value stored at key.

When offset is beyond the string length, the string is assumed to be a contiguous space with 0 bits. When key does not exist it is assumed to be an empty string, so offset is always out of range and the value is also assumed to be a contiguous space with 0 bits.

New in version 0.2.0.

Note: Time complexity: O(1)

Parameters

- **key** (str, bytes) – The key to get the bit from
- **offset** (*int*) – The bit offset to fetch the bit from

Return type bytes|None

Raises *RedisError*

getrange(*key*, *start*, *end*)
Returns the bit value at offset in the string value stored at key.

When offset is beyond the string length, the string is assumed to be a contiguous space with 0 bits. When key does not exist it is assumed to be an empty string, so offset is always out of range and the value is also assumed to be a contiguous space with 0 bits.

New in version 0.2.0.

Note: Time complexity: O(N) where N is the length of the returned string. The complexity is ultimately determined by the returned length, but because creating a substring from an existing string is very cheap, it can be considered O(1) for small strings.

Parameters

- **key** (str, bytes) – The key to get the bit from
- **start** (*int*) – The start position to evaluate in the string

- **end** (*int*) – The end position to evaluate in the string

Return type bytes|None

Raises *RedisError*

getset(*key*, *value*)
Atomically sets key to value and returns the old value stored at key. Returns an error when key exists but does not hold a string value.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to remove
- **value** (`str`, `bytes`) – The value to set

Returns The previous value

Return type bytes

Raises *RedisError*

hdel(*key*, **fields*)
Remove the specified fields from the hash stored at *key*.

Specified fields that do not exist within this hash are ignored. If *key* does not exist, it is treated as an empty hash and this command returns zero.

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **fields** – iterable of field names to retrieve

Returns the number of fields that were removed from the hash, not including specified by non-existing fields.

Return type int

hexists(*key*, *field*)
Returns if *field* is an existing field in the hash stored at *key*.

Note: *Time complexity*: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **field** – name of the field to test for

Return type bool

hget(*key*, *field*)
Returns the value associated with *field* in the hash stored at *key*.

Note: **Time complexity**: always `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **field** – The field in the hash to get

Return type bytes, list

Raises *RedisError*

hgetall (*key*)

Returns all fields and values of the has stored at *key*.

The underlying redis HGETALL command returns an array of pairs. This method converts that to a Python `dict`. It will return an empty `dict` when the key is not found.

Note: **Time complexity**: `O(N)` where `N` is the size of the hash.

Parameters **key** (`str`, `bytes`) – The key of the hash

Returns a `dict` of key to value mappings for all fields in the hash

hincrby (*key*, *field*, *increment*)

Increments the number stored at *field* in the hash stored at *key*.

If *key* does not exist, a new key holding a hash is created. If *field* does not exist the value is set to `0` before the operation is performed. The range of values supported is limited to 64-bit signed integers.

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **field** – name of the field to increment
- **increment** (*int*) – amount to increment by

Returns the value at *field* after the increment occurs

Return type int

hincrbyfloat (*key*, *field*, *increment*)

Increments the number stored at *field* in the hash stored at *key*.

If the increment value is negative, the result is to have the hash field **decremented** instead of incremented. If the field does not exist, it is set to `0` before performing the operation. An error is returned if one of the following conditions occur:

- the field contains a value of the wrong type (not a string)
- the current field content or the specified increment are not parseable as a double precision floating point number

Note: *Time complexity*: `O(1)`

Parameters

- **key** (str, bytes) – The key of the hash
- **field** – name of the field to increment
- **increment** (*float*) – amount to increment by

Returns the value at *field* after the increment occurs

Return type float

hkeys(*key*)
Returns all field names in the hash stored at *key*.

Note: *Time complexity*: `O(N)` where `N` is the size of the hash

Parameters **key** (str, bytes) – The key of the hash

Returns the list of fields in the hash

Return type list

hlen(*key*)
Returns the number of fields contained in the hash stored at *key*.

Note: *Time complexity*: `O(1)`

Parameters **key** (str, bytes) – The key of the hash

Returns the number of fields in the hash or zero when *key* does not exist

Return type int

hmget(*key*, **fields*)
Returns the values associated with the specified *fields* in a hash.

For every `field` that does not exist in the hash, `None` is returned. Because a non-existing keys are treated as empty hashes, calling *hmget()* against a non-existing key will return a list of `None` values.

Note: *Time complexity*: `O(N)` where `N` is the number of fields being requested.

Parameters

- **key** (str, bytes) – The key of the hash
- **fields** – iterable of field names to retrieve

Returns a `dict` of field name to value mappings for each of the requested fields

Return type dict

hmset(*key*, *value_dict*)
Sets fields to values as in *value_dict* in the hash stored at *key*.

Sets the specified fields to their respective values in the hash stored at *key*. This command overwrites any specified fields already existing in the hash. If *key* does not exist, a new key holding a hash is created.

Note: **Time complexity**: `O(N)` where `N` is the number of fields being set.

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **value_dict** (`dict`) – field to value mapping

Return type bool

Raises *RedisError*

hset(*key*, *field*, *value*)
Sets *field* in the hash stored at *key* to *value*.

If *key* does not exist, a new key holding a hash is created. If *field* already exists in the hash, it is overwritten.

Note: **Time complexity**: always `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **field** – The field in the hash to set
- **value** – The value to set the field to

Returns `1` if *field* is a new field in the hash and *value* was set; otherwise, `0` if *field* already exists in the hash and the value was updated

Return type int

hsetnx(*key*, *field*, *value*)
Sets *field* in the hash stored at *key* only if it does not exist.

Sets *field* in the hash stored at *key* only if *field* does not yet exist. If *key* does not exist, a new key holding a hash is created. If *field* already exists, this operation has no effect.

Note: *Time complexity*: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key of the hash
- **field** – The field in the hash to set
- **value** – The value to set the field to

Returns `1` if *field* is a new field in the hash and *value* was set. `0` if *field* already exists in the hash and no operation was performed

Return type int

hvals(*key*)
Returns all values in the hash stored at *key*.

Note: *Time complexity* `O(N)` where `N` is the size of the hash

Parameters **key** (`str`, `bytes`) – The key of the hash

Returns a `list` of `bytes` instances or an empty list when *key* does not exist

Return type list

incr(*key*)

Increments the number stored at key by one. If the key does not exist, it is set to `0` before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that can not be represented as integer. This operation is limited to 64 bit signed integers.

Note: This is a string operation because Redis does not have a dedicated integer type. The string stored at the key is interpreted as a base-10 64 bit signed integer to execute the operation.

Redis stores integers in their integer representation, so for string values that actually hold an integer, there is no overhead for storing the string representation of the integer.

Note: **Time complexity**: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to increment

Return type int

Raises *RedisError*

incrby(*key*, *increment*)

Increments the number stored at key by increment. If the key does not exist, it is set to 0 before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that can not be represented as integer. This operation is limited to 64 bit signed integers.

See *incr()* for extra information on increment/decrement operations.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to increment
- **increment** (*int*) – The amount to increment by

Returns The value of key after the increment

Return type int

Raises *RedisError*

incrbyfloat(*key*, *increment*)

Increment the string representing a floating point number stored at key by the specified increment. If the

key does not exist, it is set to 0 before performing the operation. An error is returned if one of the following conditions occur:

- The key contains a value of the wrong type (not a string).
- The current key content or the specified increment are not parsable as a double precision floating point number.

If the command is successful the new incremented value is stored as the new value of the key (replacing the old one), and returned to the caller as a string.

Both the value already contained in the string key and the increment argument can be optionally provided in exponential notation, however the value computed after the increment is stored consistently in the same format, that is, an integer number followed (if needed) by a dot, and a variable number of digits representing the decimal part of the number. Trailing zeroes are always removed.

The precision of the output is fixed at 17 digits after the decimal point regardless of the actual internal precision of the computation.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters

- **`key`** (`str`, `bytes`) – The key to increment
- **`increment`** (*`float`*) – The amount to increment by

Returns The value of key after the increment

Return type bytes

Raises *`RedisError`*

`info`(*section=None*)

The INFO command returns information and statistics about the server in a format that is simple to parse by computers and easy to read by humans.

The optional parameter can be used to select a specific section of information:

- server: General information about the Redis server
- clients: Client connections section
- memory: Memory consumption related information
- persistence: RDB and AOF related information
- stats: General statistics
- replication: Master/slave replication information
- cpu: CPU consumption statistics
- commandstats: Redis command statistics
- cluster: Redis Cluster section
- keyspace: Database related statistics

It can also take the following values:

- all: Return all sections

- default: Return only the default set of sections

When no parameter is provided, the default option is assumed.

Parameters **section** (*str*) – Optional

Returns dict

keys(*pattern*)

Returns all keys matching pattern.

While the time complexity for this operation is O(N), the constant times are fairly low. For example, Redis running on an entry level laptop can scan a 1 million key database in 40 milliseconds.

Warning: Consider *keys()* as a command that should only be used in production environments with extreme care. It may ruin performance when it is executed against large databases. This command is intended for debugging and special operations, such as changing your keyspace layout. Don't use *keys()* in your regular application code. If you're looking for a way to find keys in a subset of your keyspace, consider using *scan()* or sets.

Supported glob-style patterns:

- h?llo matches hello, hallo and hxllo
- h*llo matches hllo and heeeello
- h[ae]llo matches hello and hallo, but not hillo
- h[^e]llo matches hallo, hbllo, but not hello
- h[a-b]llo matches hallo and hbllo

Use a backslash (\) to escape special characters if you want to match them verbatim.

Note: **Time complexity**: O(N)

Parameters **pattern** (str, bytes) – The pattern to use when looking for keys

Return type list

Raises *RedisError*

llen(*key*)

Returns the length of the list stored at key.

Parameters **key** (str, bytes) – The list's key

Return type int

Raises *TRedisException*

If key does not exist, it is interpreted as an empty list and 0 is returned. An error is returned when the value stored at key is not a list.

Note: **Time complexity** O(1)

lpop(*key*)

Removes and returns the first element of the list stored at key.

Parameters **key** (str, bytes) – The list's key

Returns the element at the head of the list, None if the list does not exist

Raises *TRedisException*

Note: **Time complexity**: O(1)

---

lpush(*key*, **values*)
Insert all the specified values at the head of the list stored at key.

Parameters

- **key** (str, bytes) – The list's key
- **values** – One or more positional arguments to insert at the beginning of the list. Each value is inserted at the beginning of the list individually (see discussion below).

Returns the length of the list after push operations

Return type int

Raises *TRedisException*

If *key* does not exist, it is created as empty list before performing the push operations. When key holds a value that is not a list, an error is returned.

It is possible to push multiple elements using a single command call just specifying multiple arguments at the end of the command. Elements are inserted one after the other to the head of the list, from the leftmost element to the rightmost element. So for instance `client.lpush('mylist', 'a', 'b', 'c')` will result into a list containing `c` as first element, `b` as second element and `a` as third element.

Note: **Time complexity**: O(1)

---

lpushx(*key*, **values*)
Insert values at the head of an existing list.

Parameters

- **key** (str, bytes) – The list's key
- **values** – One or more positional arguments to insert at the beginning of the list. Each value is inserted at the beginning of the list individually (see discussion below).

Returns the length of the list after push operations, zero if *key* does not refer to a list

Return type int

Raises *TRedisException*

This method inserts *values* at the head of the list stored at *key*, only if *key* already exists and holds a list. In contrary to *lpush()*, no operation will be performed when key does not yet exist.

Note: **Time complexity**: O(1)

---

lrange(*key*, *start*, *end*)
Returns the specified elements of the list stored at key.

Parameters

- **key** (str, bytes) – The list's key
- **start** (*int*) – zero-based index to start retrieving elements from
- **end** (*int*) – zero-based index at which to stop retrieving elements

Return type list

Raises *TRedisException*

The offsets start and stop are zero-based indexes, with 0 being the first element of the list (the head of the list), 1 being the next element and so on.

These offsets can also be negative numbers indicating offsets starting at the end of the list. For example, -1 is the last element of the list, -2 the penultimate, and so on.

Note that if you have a list of numbers from 0 to 100, `lrange(key, 0, 10)` will return 11 elements, that is, the rightmost item is included. This may or may not be consistent with behavior of range-related functions in your programming language of choice (think Ruby's `Range.new`, `Array#slice` or Python's `range()` function).

Out of range indexes will not produce an error. If start is larger than the end of the list, an empty list is returned. If stop is larger than the actual end of the list, Redis will treat it like the last element of the list.

Note: **Time complexity** `O(S+N)` where `S` is the distance of start offset from `HEAD` for small lists, from nearest end (`HEAD` or `TAIL`) for large lists; and `N` is the number of elements in the specified range.

---

ltrim(*key*, *start*, *stop*)

Crop a list to the specified range.

Parameters

- **key** (str, bytes) – The list's key
- **start** (*int*) – zero-based index to first element to retain
- **stop** (*int*) – zero-based index of the last element to retain

Returns did the operation succeed?

Return type bool

Raises *TRedisException*

Trim an existing list so that it will contain only the specified range of elements specified.

Both *start* and *stop* are zero-based indexes, where 0 is the first element of the list (the head), 1 the next element and so on. For example: `ltrim('foobar', 0, 2)` will modify the list stored at `foobar` so that only the first three elements of the list will remain.

start and *stop* can also be negative numbers indicating offsets from the end of the list, where -1 is the last element of the list, -2 the penultimate element and so on.

Out of range indexes will not produce an error: if *start* is larger than the *end* of the list, or *start* > *end*, the result will be an empty list (which causes *key* to be removed). If *end* is larger than the end of the list, Redis will treat it like the last element of the list.

A common use of LTRIM is together with LPUSH / RPUSH. For example:

```
client.lpush('mylist', 'somelement')
client.ltrim('mylist', 0, 99)
```

This pair of commands will push a new element on the list, while making sure that the list will not grow larger than 100 elements. This is very useful when using Redis to store logs for example. It is important to note that when used in this way LTRIM is an O(1) operation because in the average case just one element is removed from the tail of the list.

Note: Time complexity: `O(N)` where *N* is the number of elements to be removed by the operation.

---

mget (**keys*)

Returns the values of all specified keys. For every key that does not hold a string value or does not exist, the special value nil is returned. Because of this, the operation never fails.

New in version 0.2.0.

Note: **Time complexity**: `O(N)` where `N` is the number of keys to retrieve.

Parameters **keys** (`str`, `bytes`) – One or more keys as keyword arguments to the function

Return type list

Raises *RedisError*

migrate (*host*, *port*, *key*, *destination_db*, *timeout*, *copy=False*, *replace=False*)

Atomically transfer a key from a source Redis instance to a destination Redis instance. On success the key is deleted from the original instance and is guaranteed to exist in the target instance.

The command is atomic and blocks the two instances for the time required to transfer the key, at any given time the key will appear to exist in a given instance or in the other instance, unless a timeout error occurs.

Note: **Time complexity**: This command actually executes a DUMP+DEL in the source instance, and a RESTORE in the target instance. See the pages of these commands for time complexity. Also an `O(N)` data transfer between the two instances is performed.

Parameters

- **host** (*bytes, str*) – The host to migrate the key to
- **port** (*int*) – The port to connect on
- **key** (*bytes, str*) – The key to migrate
- **destination_db** (*int*) – The database number to select
- **timeout** (*int*) – The maximum idle time in milliseconds
- **copy** (*bool*) – Do not remove the key from the local instance
- **replace** (*bool*) – Replace existing key on the remote instance

Return type bool

Raises *RedisError*

move (*key*, *db*)

Move key from the currently selected database (see *select()*) to the specified destination database. When key already exists in the destination database, or it does not exist in the source database, it does nothing. It is possible to use *move()* as a locking primitive because of this.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to move
- **db** (*int*) – The database number

Return type bool

Raises *RedisError*

mset(*mapping*)

Sets the given keys to their respective values. *mset()* replaces existing values with new values, just as regular *set()*. See *msetnx()* if you don't want to overwrite existing values.

mset() is atomic, so all given keys are set at once. It is not possible for clients to see that some of the keys were updated while others are unchanged.

New in version 0.2.0.

Note: **Time complexity**: `O(N)` where `N` is the number of keys to set.

Parameters **mapping** (*dict*) – A mapping of key/value pairs to set

Return type bool

Raises *RedisError*

msetnx(*mapping*)

Sets the given keys to their respective values. *msetnx()* will not perform any operation at all even if just a single key already exists.

Because of this semantic *msetnx()* can be used in order to set different keys representing different fields of an unique logic object in a way that ensures that either all the fields or none at all are set.

msetnx() is atomic, so all given keys are set at once. It is not possible for clients to see that some of the keys were updated while others are unchanged.

New in version 0.2.0.

Note: **Time complexity**: `O(N)` where `N` is the number of keys to set.

Parameters **mapping** (*dict*) – A mapping of key/value pairs to set

Return type bool

Raises *RedisError*

object_encoding(*key*)

Return the kind of internal representation used in order to store the value associated with a key

Note: **Time complexity**: `O(1)`

Parameters **key** (str, bytes) – The key to get the encoding for

Return type bytes

Raises *RedisError*

object_idle_time(*key*)
Return the number of seconds since the object stored at the specified key is idle (not requested by read or write operations). While the value is returned in seconds the actual resolution of this timer is 10 seconds, but may vary in future implementations of Redis.

Note: **Time complexity**: O(1)

Parameters **key** (str, bytes) – The key to get the idle time for

Return type int

Raises *RedisError*

object_refcount(*key*)
Return the number of references of the value associated with the specified key. This command is mainly useful for debugging.

Note: **Time complexity**: O(1)

Parameters **key** (str, bytes) – The key to get the refcount for

Return type int

Raises *RedisError*

persist(*key*)
Remove the existing timeout on key, turning the key from volatile (a key with an expire set) to persistent (a key that will never expire as no timeout is associated).

Note: **Time complexity**: O(1)

Parameters **key** (str, bytes) – The key to move

Return type bool

Raises *RedisError*

pexpire(*key*, *timeout*)
This command works exactly like *pexpire()* but the time to live of the key is specified in milliseconds instead of seconds.

Note: **Time complexity**: O(1)

Parameters

- **key** (str, bytes) – The key to set an expiration for

- **timeout** (*int*) – The number of milliseconds to set the timeout to

Return type bool

Raises *RedisError*

pexpireat(*key*, *timestamp*)

pexpireat() has the same effect and semantic as *expireat()*, but the Unix time at which the key will expire is specified in milliseconds instead of seconds.

Note: Time complexity: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to set an expiration for
- **timestamp** (*int*) – The expiration UNIX epoch value in milliseconds

Return type bool

Raises *RedisError*

pfadd(*key*, **elements*)

Adds all the element arguments to the HyperLogLog data structure stored at the variable name specified as first argument.

As a side effect of this command the HyperLogLog internals may be updated to reflect a different estimation of the number of unique items added so far (the cardinality of the set).

If the approximated cardinality estimated by the HyperLogLog changed after executing the command, *pfadd()* returns `1`, otherwise `0` is returned. The command automatically creates an empty HyperLogLog structure (that is, a Redis String of a specified length and with a given encoding) if the specified key does not exist.

To call the command without elements but just the variable name is valid, this will result into no operation performed if the variable already exists, or just the creation of the data structure if the key does not exist (in the latter case `1` is returned).

For an introduction to HyperLogLog data structure check *pfcount()*.

New in version 0.2.0.

Note: Time complexity: `O(1)` to add every element.

Parameters

- **key** (`str`, `bytes`) – The key to add the elements to
- **elements** (`str`, `bytes`) – One or more elements to add

Return type bool

Raises *RedisError*

pfcount(**keys*)

When called with a single key, returns the approximated cardinality computed by the HyperLogLog data structure stored at the specified variable, which is `0` if the variable does not exist.

When called with multiple keys, returns the approximated cardinality of the union of the HyperLogLogs passed, by internally merging the HyperLogLogs stored at the provided keys into a temporary HyperLogLog.

The HyperLogLog data structure can be used in order to count unique elements in a set using just a small constant amount of memory, specifically 12k bytes for every HyperLogLog (plus a few bytes for the key itself).

The returned cardinality of the observed set is not exact, but approximated with a standard error of 0.81%.

For example in order to take the count of all the unique search queries performed in a day, a program needs to call `pfcount()` every time a query is processed. The estimated number of unique queries can be retrieved with `pfcount()` at any time.

Note: as a side effect of calling this function, it is possible that the HyperLogLog is modified, since the last 8 bytes encode the latest computed cardinality for caching purposes. So `pfcount()` is technically a write command.

New in version 0.2.0.

Note: **Time complexity**: `O(1)` with every small average constant times when called with a single key. `O(N)` with `N` being the number of keys, and much bigger constant times, when called with multiple keys.

Parameters **keys** (`str`, `bytes`) – One or more keys

Return type int

Returns The approximated number of unique elements observed

Raises *RedisError*

pfmerge(*dest_key*, **keys*)

Merge multiple HyperLogLog values into an unique value that will approximate the cardinality of the union of the observed Sets of the source HyperLogLog structures.

The computed merged HyperLogLog is set to the destination variable, which is created if does not exist (defaulting to an empty HyperLogLog).

New in version 0.2.0.

Note: **Time complexity**: `O(N)` to merge `N` HyperLogLogs, but with high constant times.

Parameters

- **dest_key** (`str`, `bytes`) – The destination key
- **keys** (`str`, `bytes`) – One or more keys

Return type bool

Raises *RedisError*

ping()

Returns `PONG` if no argument is provided, otherwise return a copy of the argument as a bulk. This command is often used to test if a connection is still alive, or to measure latency.

If the client is subscribed to a channel or a pattern, it will instead return a multi-bulk with a `pong` in the first position and an empty bulk in the second position, unless an argument is provided in which case it returns a copy of the argument.

Return type bytes

Raises *RedisError*

psetex(*key*, *milliseconds*, *value*)

psetex() works exactly like *psetex()* with the sole difference that the expire time is specified in milliseconds instead of seconds.

New in version 0.2.0.

Note: Time complexity: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to set
- **milliseconds** (*int*) – Number of milliseconds for TTL
- **value** (`str`, `bytes`) – The value to set

Return type bool

Raises *RedisError*

pttl(*key*)

Like *ttl()* this command returns the remaining time to live of a key that has an expire set, with the sole difference that *ttl()* returns the amount of remaining time in seconds while *pttl()* returns it in milliseconds.

In Redis 2.6 or older the command returns `-1` if the key does not exist or if the key exist but has no associated expire.

Starting with Redis 2.8 the return value in case of error changed:

- The command returns `-2` if the key does not exist.
- The command returns `-1` if the key exists but has no associated expire.

Note: Time complexity: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to get the PTTL for

Return type int

Raises *RedisError*

quit()

Ask the server to close the connection. The connection is closed as soon as all pending replies have been written to the client.

Return type bool

Raises *RedisError*

randomkey()
Return a random key from the currently selected database.

Note: Time complexity: O(1)

Return type bytes

Raises *RedisError*

ready
Indicates that the client is connected to the Redis server or cluster and is ready for use.

Return type bool

rename(*key*, *new_key*)
Renames key to new_key. It returns an error when the source and destination names are the same, or when key does not exist. If new_key already exists it is overwritten, when this happens *rename()* executes an implicit *delete()* operation, so if the deleted key contains a very big value it may cause high latency even if *rename()* itself is usually a constant-time operation.

Note: Time complexity: O(1)

Parameters

- **key** (str, bytes) – The key to rename
- **new_key** (str, bytes) – The key to rename it to

Return type bool

Raises *RedisError*

renamenx(*key*, *new_key*)
Renames key to new_key if new_key does not yet exist. It returns an error under the same conditions as *rename()*.

Note: Time complexity: O(1)

Parameters

- **key** (str, bytes) – The key to rename
- **new_key** (str, bytes) – The key to rename it to

Return type bool

Raises *RedisError*

restore(*key*, *ttl*, *value*, *replace=False*)
Create a key associated with a value that is obtained by deserializing the provided serialized value (obtained via *dump()*).

If ttl is 0 the key is created without any expire, otherwise the specified expire time (in milliseconds) is set.

restore() will return a `Target key name is busy` error when key already exists unless you use the *restore()* modifier (Redis 3.0 or greater).

restore() checks the RDB version and data checksum. If they don't match an error is returned.

Note: Time complexity: `O(1)` to create the new key and additional `O(N*M)` to reconstruct the serialized value, where `N` is the number of Redis objects composing the value and `M` their average size. For small string values the time complexity is thus `O(1)+O(1*M)` where `M` is small, so simply `O(1)`. However for sorted set values the complexity is `O(N*M*log(N))` because inserting values into sorted sets is `O(log(N))`.

Parameters

- **key** (`str`, `bytes`) – The key to get the TTL for
- **ttl** (*int*) – The number of seconds to set the timeout to
- **value** (`str`, `bytes`) – The value to restore to the key
- **replace** (*bool*) – Replace a pre-existing key

Return type bool

Raises *RedisError*

rpop(*key*)

Removes and returns the last element of the list stored at key.

Parameters **key** (`str`, `bytes`) – The list's key

Returns the length of the list after push operations or zero if *key* does not refer to a list

Returns the element at the tail of the list, `None` if the list does not exist

Return type int

Raises *TRedisException*

rpush(*key*, **values*)

Insert all the specified values at the tail of the list stored at key.

Parameters

- **key** (`str`, `bytes`) – The list's key
- **values** – One or more positional arguments to insert at the tail of the list.

Returns the length of the list after push operations

Return type int

Raises *TRedisException*

If *key* does not exist, it is created as empty list before performing the push operation. When *key* holds a value that is not a list, an error is returned.

It is possible to push multiple elements using a single command call just specifying multiple arguments at the end of the command. Elements are inserted one after the other to the tail of the list, from the leftmost element to the rightmost element. So for instance the command `client.rpush('mylist', 'a', 'b', 'c')` will result in a list containing `a` as first element, `b` as second element and `c` as third element.

Note: Time complexity: `O(1)`

rpushx(*key*, **values*)
Insert values at the tail of an existing list.

Parameters

- **key** (`str`, `bytes`) – The list's key
- **values** – One or more positional arguments to insert at the tail of the list.

Returns the length of the list after push operations or zero if *key* does not refer to a list

Return type int

Raises *TRedisException*

This method inserts value at the tail of the list stored at *key*, only if *key* already exists and holds a list. In contrary to method:.*rpush*, no operation will be performed when *key* does not yet exist.

Note: Time complexity: `O(1)`

sadd(*key*, **members*)
Add the specified members to the set stored at key. Specified members that are already a member of this set are ignored. If key does not exist, a new set is created before adding the specified members.

An error is returned when the value stored at key is not a set.

Returns `True` if all requested members are added. If more than one member is passed in and not all members are added, the number of added members is returned.

Note: Time complexity: `O(N)` where `N` is the number of members to be added.

Parameters

- **key** (`str`, `bytes`) – The key of the set
- **members** – One or more positional arguments to add to the set

Returns Number of items added to the set

Return type bool, int

scan(*cursor=0*, *pattern=None*, *count=None*)
The *scan()* command and the closely related commands *sscan()*, `hscan()` and `zscan()` are used in order to incrementally iterate over a collection of elements.

- *scan()* iterates the set of keys in the currently selected Redis database.
- *sscan()* iterates elements of Sets types.
- `hscan()` iterates fields of Hash types and their associated values.
- `zscan()` iterates elements of Sorted Set types and their associated scores.

Basic usage

scan() is a cursor based iterator. This means that at every call of the command, the server returns an updated cursor that the user needs to use as the cursor argument in the next call.

An iteration starts when the cursor is set to `0`, and terminates when the cursor returned by the server is `0`.

For more information on *`scan()`*, visit the Redis docs on scan.

Note: Time complexity: `O(1)` for every call. `O(N)` for a complete iteration, including enough command calls for the cursor to return back to `0`. `N` is the number of elements inside the collection.

Parameters

- **`cursor`** (*`int`*) – The server specified cursor value or `0`
- **`pattern`** (`str`, `bytes`) – An optional pattern to apply for key matching
- **`count`** (*`int`*) – An optional amount of work to perform in the scan

Return type int, list

Returns A tuple containing the cursor and the list of keys

Raises *`RedisError`*

`scard`(*key*)

Returns the set cardinality (number of elements) of the set stored at key.

Note: Time complexity: `O(1)`

Parameters **`key`** (`str`, `bytes`) – The key of the set

Return type int

Raises *`RedisError`*

`script_exists`(**hashes*)

Returns information about the existence of the scripts in the script cache.

This command accepts one or more SHA1 digests and returns a list of ones or zeros to signal if the scripts are already defined or not inside the script cache. This can be useful before a pipelining operation to ensure that scripts are loaded (and if not, to load them using *`script_load()`*) so that the pipelining operation can be performed solely using *`evalsha()`* instead of *`eval()`* to save bandwidth.

Please refer to the *`eval()`* documentation for detailed information about Redis Lua scripting.

Note: Time complexity: `O(N)` with `N` being the number of scripts to check (so checking a single script is an `O(1)` operation).

Parameters **`hashes`** (*`str`*) – One or more sha1 hashes to check for in the cache

Return type list

Returns Returns a list of `1` or `0` indicating if the specified script(s) exist in the cache.

`script_flush`()

Flush the Lua scripts cache.

Please refer to the *`eval()`* documentation for detailed information about Redis Lua scripting.

Note: **Time complexity**: `O(N)` with `N` being the number of scripts in cache

Return type bool

script_kill()
Kills the currently executing Lua script, assuming no write operation was yet performed by the script.

This command is mainly useful to kill a script that is running for too much time(for instance because it entered an infinite loop because of a bug). The script will be killed and the client currently blocked into *eval()* will see the command returning with an error.

If the script already performed write operations it can not be killed in this way because it would violate Lua script atomicity contract. In such a case only SHUTDOWN NOSAVE is able to kill the script, killing the Redis process in an hard way preventing it to persist with half-written information.

Please refer to the *eval()* documentation for detailed information about Redis Lua scripting.

Note: **Time complexity**: `O(1)`

Return type bool

script_load(*script*)
Load a script into the scripts cache, without executing it. After the specified command is loaded into the script cache it will be callable using *evalsha()* with the correct SHA1 digest of the script, exactly like after the first successful invocation of *eval()*.

The script is guaranteed to stay in the script cache forever (unless *script_flush()* is called).

The command works in the same way even if the script was already present in the script cache.

Please refer to the *eval()* documentation for detailed information about Redis Lua scripting.

Note: **Time complexity**: `O(N)` with `N` being the length in bytes of the script body.

Parameters **script** (*str*) – The script to load into the script cache

Returns str

sdiff(**keys*)
Returns the members of the set resulting from the difference between the first set and all the successive sets.

For example:

```
key1 = {a,b,c,d}
key2 = {c}
key3 = {a,c,e}
SDIFF key1 key2 key3 = {b,d}
```

Keys that do not exist are considered to be empty sets.

Note: **Time complexity**: `O(N)` where `N` is the total number of elements in all given sets.

Parameters **keys** (str, bytes) – Two or more set keys as positional arguments

Return type list

Raises *RedisError*

sdiffstore(*destination*, **keys*)
This command is equal to *sdiff()*, but instead of returning the resulting set, it is stored in destination.

If destination already exists, it is overwritten.

Note: **Time complexity**: O(N) where N is the total number of elements in all given sets.

Parameters

- **destination** (str, bytes) – The set to store the diff into
- **keys** (str, bytes) – One or more set keys as positional arguments

Return type int

Raises *RedisError*

select(*index=0*)
Select the DB with having the specified zero-based numeric index. New connections always use DB 0.

Parameters **index** (*int*) – The database to select

Return type bool

Raises *RedisError*

Raises *InvalidClusterCommand*

set(*key*, *value*, *ex=None*, *px=None*, *nx=False*, *xx=False*)
Set key to hold the string value. If key already holds a value, it is overwritten, regardless of its type. Any previous time to live associated with the key is discarded on successful *set()* operation.

If the value is not one of str, bytes, or int, a ValueError will be raised.

Note: **Time complexity**: O(1)

Parameters

- **key** (str, bytes) – The key to remove
- **value** (str, bytes, int) – The value to set
- **ex** (*int*) – Set the specified expire time, in seconds
- **px** (*int*) – Set the specified expire time, in milliseconds
- **nx** (*bool*) – Only set the key if it does not already exist
- **xx** (*bool*) – Only set the key if it already exist

Return type bool

Raises *RedisError*

Raises ValueError

setbit (*key*, *offset*, *bit*)
Sets or clears the bit at offset in the string value stored at key.

The bit is either set or cleared depending on value, which can be either 0 or 1. When key does not exist, a new string value is created. The string is grown to make sure it can hold a bit at offset. The offset argument is required to be greater than or equal to 0, and smaller than 2^{32} (this limits bitmaps to 512MB). When the string at key is grown, added bits are set to 0.

> **Warning:** When setting the last possible bit (offset equal to 2^{32} -1) and the string value stored at key does not yet hold a string value, or holds a small string value, Redis needs to allocate all intermediate memory which can block the server for some time. On a 2010 MacBook Pro, setting bit number 2^{32} -1 (512MB allocation) takes ~300ms, setting bit number 2^{30} -1 (128MB allocation) takes ~80ms, setting bit number 2^{28} -1 (32MB allocation) takes ~30ms and setting bit number 2^{26} -1 (8MB allocation) takes ~8ms. Note that once this first allocation is done, subsequent calls to *setbit()* for the same key will not have the allocation overhead.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to get the bit from
- **offset** (*int*) – The bit offset to fetch the bit from
- **bit** (*int*) – The value (`0` or `1`) to set for the bit

Return type int

Raises *RedisError*

setex (*key*, *seconds*, *value*)
Set key to hold the string value and set key to timeout after a given number of seconds.

setex() is atomic, and can be reproduced by using *set()* and *expire()* inside an `multi()` / `exec()` block. It is provided as a faster alternative to the given sequence of operations, because this operation is very common when Redis is used as a cache.

An error is returned when seconds is invalid.

New in version 0.2.0.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key to set
- **seconds** (*int*) – Number of seconds for TTL
- **value** (`str`, `bytes`) – The value to set

Return type bool

Raises *RedisError*

setnx(*key*, *value*)
Set key to hold string value if key does not exist. In that case, it is equal to *setnx()*. When key already holds a value, no operation is performed. *setnx()* is short for "SET if Not eXists".

New in version 0.2.0.

Note: **Time complexity**: O(1)

Parameters

- **key** (str, bytes) – The key to set
- **value** (str, bytes, int) – The value to set

Return type bool

Raises *RedisError*

setrange(*key*, *offset*, *value*)
Overwrites part of the string stored at key, starting at the specified offset, for the entire length of value. If the offset is larger than the current length of the string at key, the string is padded with zero-bytes to make offset fit. Non-existing keys are considered as empty strings, so this command will make sure it holds a string large enough to be able to set value at offset.

Note: The maximum offset that you can set is 2^{29} -1 (536870911), as Redis Strings are limited to 512 megabytes. If you need to grow beyond this size, you can use multiple keys.

Warning: When setting the last possible byte and the string value stored at key does not yet hold a string value, or holds a small string value, Redis needs to allocate all intermediate memory which can block the server for some time. On a 2010 MacBook Pro, setting byte number 536870911 (512MB allocation) takes ~300ms, setting byte number 134217728 (128MB allocation) takes ~80ms, setting bit number 33554432 (32MB allocation) takes ~30ms and setting bit number 8388608 (8MB allocation) takes ~8ms. Note that once this first allocation is done, subsequent calls to *setrange()* for the same key will not have the allocation overhead.

New in version 0.2.0.

Note: **Time complexity**: O(1), not counting the time taken to copy the new string in place. Usually, this string is very small so the amortized complexity is O(1). Otherwise, complexity is O(M) with M being the length of the value argument.

Parameters

- **key** (str, bytes) – The key to get the bit from
- **value** (str, bytes, int) – The value to set

Returns The length of the string after it was modified by the command

Return type int

Raises *RedisError*

sinter(**keys*)
Returns the members of the set resulting from the intersection of all the given sets.

For example:

```
key1 = {a,b,c,d}
key2 = {c}
key3 = {a,c,e}
SINTER key1 key2 key3 = {c}
```

Keys that do not exist are considered to be empty sets. With one of the keys being an empty set, the resulting set is also empty (since set intersection with an empty set always results in an empty set).

Note: **Time complexity**: `O(N*M)` worst case where `N` is the cardinality of the smallest set and `M` is the number of sets.

Parameters **keys** (`str`, `bytes`) – Two or more set keys as positional arguments

Return type list

Raises *RedisError*

sinterstore(*destination*, **keys*)
This command is equal to *sinter()*, but instead of returning the resulting set, it is stored in destination.

If destination already exists, it is overwritten.

Note: **Time complexity**: `O(N*M)` worst case where `N` is the cardinality of the smallest set and `M` is the number of sets.

Parameters

- **destination** (`str`, `bytes`) – The set to store the intersection into
- **keys** (`str`, `bytes`) – One or more set keys as positional arguments

Return type int

Raises *RedisError*

sismember(*key*, *member*)
Returns `True` if `member` is a member of the set stored at key.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key of the set to check for membership in
- **member** (`str`, `bytes`) – The value to check for set membership with

Return type bool

Raises *RedisError*

smembers(*key*)
Returns all the members of the set value stored at key.

This has the same effect as running *sinter()* with one argument key.

Note: Time complexity: O(N) where N is the set cardinality.

Parameters **key** (str, bytes) – The key of the set to return the members from

Return type list

Raises *RedisError*

smove(*source*, *destination*, *member*)
Move member from the set at source to the set at destination. This operation is atomic. In every given moment the element will appear to be a member of source or destination for other clients.

If the source set does not exist or does not contain the specified element, no operation is performed and False is returned. Otherwise, the element is removed from the source set and added to the destination set. When the specified element already exists in the destination set, it is only removed from the source set.

An error is returned if source or destination does not hold a set value.

Note: Time complexity: O(1)

Parameters

- **source** (str, bytes) – The source set key
- **destination** (str, bytes) – The destination set key
- **member** (str, bytes) – The member value to move

Return type bool

Raises *RedisError*

sort(*key*, *by=None*, *external=None*, *offset=0*, *limit=None*, *order=None*, *alpha=False*, *store_as=None*)
Returns or stores the elements contained in the list, set or sorted set at key. By default, sorting is numeric and elements are compared by their value interpreted as double precision floating point number.

The external parameter is used to specify the *GET <http://redis.io/commands/sort#retrieving-external-keys>_* parameter for retrieving external keys. It can be a single string or a list of strings.

Note: Time complexity: O(N+M*log(M)) where N is the number of elements in the list or set to sort, and M the number of returned elements. When the elements are not sorted, complexity is currently O(N) as there is a copy step that will be avoided in next releases.

Parameters

- **key** (str, bytes) – The key to get the refcount for
- **by** (str, bytes) – The optional pattern for external sorting keys
- **external** (str, bytes, list) – Pattern or list of patterns to return external keys

- **offset** (*int*) – The starting offset when using limit
- **limit** (*int*) – The number of elements to return
- **order** (str, bytes) – The sort order - one of ASC or DESC
- **alpha** (*bool*) – Sort the results lexicographically
- **store_as** (str, bytes, None) – When specified, the key to store the results as

Return type listlint

Raises *RedisError*

Raises ValueError

spop(*key*, *count=None*)

Removes and returns one or more random elements from the set value store at key.

This operation is similar to *srandmember()*, that returns one or more random elements from a set but does not remove it.

The count argument will be available in a later version and is not available in 2.6, 2.8, 3.0

Redis 3.2 will be the first version where an optional count argument can be passed to *spop()* in order to retrieve multiple elements in a single call. The implementation is already available in the unstable branch.

Note: **Time complexity**: Without the count argument O(1), otherwise O(N) where N is the absolute value of the passed count.

Parameters

- **key** (str, bytes) – The key to get one or more random members from
- **count** (*int*) – The number of members to return

Return type bytes, list

Raises *RedisError*

srandmember(*key*, *count=None*)

When called with just the key argument, return a random element from the set value stored at key.

Starting from Redis version 2.6, when called with the additional count argument, return an array of count distinct elements if count is positive. If called with a negative count the behavior changes and the command is allowed to return the same element multiple times. In this case the number of returned elements is the absolute value of the specified count.

When called with just the key argument, the operation is similar to *spop()*, however while *spop()* also removes the randomly selected element from the set, *srandmember()* will just return a random element without altering the original set in any way.

Note: **Time complexity**: Without the count argument O(1), otherwise O(N) where N is the absolute value of the passed count.

Parameters

- **key** (str, bytes) – The key to get one or more random members from
- **count** (*int*) – The number of members to return

Return type bytes, list

Raises *RedisError*

srem(*key*, **members*)

Remove the specified members from the set stored at key. Specified members that are not a member of this set are ignored. If key does not exist, it is treated as an empty set and this command returns `0`.

An error is returned when the value stored at key is not a set.

Returns `True` if all requested members are removed. If more than one member is passed in and not all members are removed, the number of removed members is returned.

Note: Time complexity: `O(N)` where `N` is the number of members to be removed.

Parameters

- **key** (`str`, `bytes`) – The key to remove the member from
- **members** (*mixed*) – One or more member values to remove

Return type bool, int

Raises *RedisError*

sscan(*key*, *cursor=0*, *pattern=None*, *count=None*)

The *sscan()* command and the closely related commands *scan()*, `hscan()` and `zscan()` are used in order to incrementally iterate over a collection of elements.

- *scan()* iterates the set of keys in the currently selected Redis database.
- *sscan()* iterates elements of Sets types.
- `hscan()` iterates fields of Hash types and their associated values.
- `zscan()` iterates elements of Sorted Set types and their associated scores.

Basic usage

sscan() is a cursor based iterator. This means that at every call of the command, the server returns an updated cursor that the user needs to use as the cursor argument in the next call.

An iteration starts when the cursor is set to `0`, and terminates when the cursor returned by the server is `0`.

For more information on *scan()*, visit the Redis docs on scan.

Note: Time complexity: `O(1)` for every call. `O(N)` for a complete iteration, including enough command calls for the cursor to return back to `0`. `N` is the number of elements inside the collection.

Parameters

- **key** (`str`, `bytes`) – The key to scan
- **cursor** (*int*) – The server specified cursor value or `0`
- **pattern** (`str`, `bytes`) – An optional pattern to apply for key matching
- **count** (*int*) – An optional amount of work to perform in the scan

Return type int, list

Returns A tuple containing the cursor and the list of set items

Raises *RedisError*

strlen(*key*)

Returns the length of the string value stored at key. An error is returned when key holds a non-string value

New in version 0.2.0.

Note: Time complexity: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to set

Returns The length of the string at key, or 0 when key does not exist

Return type int

Raises *RedisError*

sunion(**keys*)

Returns the members of the set resulting from the union of all the given sets.

For example:

```
key1 = {a,b,c,d}
key2 = {c}
key3 = {a,c,e}
SUNION key1 key2 key3 = {a,b,c,d,e}
```

Note: Time complexity: `O(N)` where `N` is the total number of elements in all given sets.

Keys that do not exist are considered to be empty sets.

Parameters **keys** (`str`, `bytes`) – Two or more set keys as positional arguments

Return type list

Raises *RedisError*

sunionstore(*destination*, **keys*)

This command is equal to *sunion()*, but instead of returning the resulting set, it is stored in destination.

If destination already exists, it is overwritten.

Note: Time complexity: `O(N)` where `N` is the total number of elements in all given sets.

Parameters

- **destination** (`str`, `bytes`) – The set to store the union into
- **keys** (`str`, `bytes`) – One or more set keys as positional arguments

Return type int

Raises *RedisError*

time()
Retrieve the current time from the redis server.

Return type float

Raises *RedisError*

ttl(*key*)
Returns the remaining time to live of a key that has a timeout. This introspection capability allows a Redis client to check how many seconds a given key will continue to be part of the dataset.

Note: Time complexity: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to get the TTL for

Return type int

Raises *RedisError*

type(*key*)
Returns the string representation of the type of the value stored at key. The different types that can be returned are: `string`, `list`, `set`, `zset`, and `hash`.

Note: Time complexity: `O(1)`

Parameters **key** (`str`, `bytes`) – The key to get the type for

Return type bytes

Raises *RedisError*

wait(*num_slaves*, *timeout=0*)
his command blocks the current client until all the previous write commands are successfully transferred and acknowledged by at least the specified number of slaves. If the timeout, specified in milliseconds, is reached, the command returns even if the specified number of slaves were not yet reached.

The command will always return the number of slaves that acknowledged the write commands sent before the *wait()* command, both in the case where the specified number of slaves are reached, or when the timeout is reached.

Note: Time complexity: `O(1)`

Parameters

- **num_slaves** (*int*) – Number of slaves to acknowledge previous writes
- **timeout** (*int*) – Timeout in milliseconds

Return type int

Raises *RedisError*

zadd(*key*, **members*, ***kwargs*)
Adds all the specified members with the specified scores to the sorted set stored at key. It is possible to

specify multiple score / member pairs. If a specified member is already a member of the sorted set, the score is updated and the element reinserted at the right position to ensure the correct ordering.

If key does not exist, a new sorted set with the specified members as sole members is created, like if the sorted set was empty. If the key exists but does not hold a sorted set, an error is returned.

The score values should be the string representation of a double precision floating point number. +inf and -inf values are valid values as well.

Members parameters

`members` could be either: - a single dict where keys correspond to scores and values to elements - multiple strings paired as score then element

```
yield client.zadd('myzset', {'1': 'one', '2': 'two'})
yield client.zadd('myzset', '1', 'one', '2', 'two')
```

ZADD options (Redis 3.0.2 or greater)

ZADD supports a list of options. Options are:

- `xx`: Only update elements that already exist. Never add elements.
- **`nx`: Don't update already existing elements. Always add new** elements.
- **`ch`: Modify the return value from the number of new elements** added, to the total number of elements changed (CH is an abbreviation of changed). Changed elements are new elements added and elements already existing for which the score was updated. So elements specified in the command having the same score as they had in the past are not counted. Note: normally the return value of `ZADD` only counts the number of new elements added.
- **`incr`: When this option is specified `ZADD` acts like** `zincrby()`. Only one score-element pair can be specified in this mode.

Note: **Time complexity**: `O(log(N))` for each item added, where `N` is the number of elements in the sorted set.

Parameters

- **`key`** (`str`, `bytes`) – The key of the sorted set
- **`members`** (`dict`, `str`, `bytes`) – Elements to add
- **`xx`** (*`bool`*) – Only update elements that already exist
- **`nx`** (*`bool`*) – Don't update already existing elements
- **`ch`** (*`bool`*) – Return the number of changed elements
- **`incr`** (*`bool`*) – Increment the score of an element

Return type int, `str`, `bytes`

Returns Number of elements changed, or the new score if incr is set

Raises *`RedisError`*

`zcard`(*key*)

Returns the set cardinality (number of elements) of the sorted set stored at key.

Note: **Time complexity**: `O(1)`

Parameters **key** (`str`, `bytes`) – The key of the set

Return type int

Raises *RedisError*

zrange(*key*, *start=0*, *stop=-1*, *with_scores=False*)

Returns the specified range of elements in the sorted set stored at key. The elements are considered to be ordered from the lowest to the highest score. Lexicographical order is used for elements with equal score.

See *tredis.Client.zrevrange()* when you need the elements ordered from highest to lowest score (and descending lexicographical order for elements with equal score).

Both start and stop are zero-based indexes, where `0` is the first element, `1` is the next element and so on. They can also be negative numbers indicating offsets from the end of the sorted set, with `-1` being the last element of the sorted set, `-2` the penultimate element and so on.

`start` and `stop` are inclusive ranges, so for example `ZRANGE myzset 0 1` will return both the first and the second element of the sorted set.

Out of range indexes will not produce an error. If start is larger than the largest index in the sorted set, or `start > stop`, an empty list is returned. If stop is larger than the end of the sorted set Redis will treat it like it is the last element of the sorted set.

It is possible to pass the `WITHSCORES` option in order to return the scores of the elements together with the elements. The returned list will contain `value1,score1,...,valueN,scoreN` instead of `value1,...,valueN`. Client libraries are free to return a more appropriate data type (suggestion: an array with (value, score) arrays/tuples).

Note: **Time complexity**: `O(log(N)+M)` with `N` being the number of elements in the sorted set and `M` the number of elements returned.

Parameters

- **key** (`str`, `bytes`) – The key of the sorted set
- **start** (*int*) – The starting index of the sorted set
- **stop** (*int*) – The ending index of the sorted set
- **with_scores** (*bool*) – Return the scores with the elements

Return type list

Raises *RedisError*

zrangebyscore(*key*, *min_score*, *max_score*, *with_scores=False*, *offset=0*, *count=0*)

Returns all the elements in the sorted set at key with a score between min and max (including elements with score equal to min or max). The elements are considered to be ordered from low to high scores.

The elements having the same score are returned in lexicographical order (this follows from a property of the sorted set implementation in Redis and does not involve further computation).

The optional `offset` and `count` arguments can be used to only get a range of the matching elements (similar to SELECT LIMIT offset, count in SQL). Keep in mind that if offset is large, the sorted set needs

to be traversed for offset elements before getting to the elements to return, which can add up to O(N) time complexity.

The optional with_scores argument makes the command return both the element and its score, instead of the element alone. This option is available since Redis 2.0.

Exclusive intervals and infinity

min_score and max_score can be -inf and +inf, so that you are not required to know the highest or lowest score in the sorted set to get all elements from or up to a certain score.

By default, the interval specified by min_score and max_score is closed (inclusive). It is possible to specify an open interval (exclusive) by prefixing the score with the character (. For example:

```
ZRANGEBYSCORE zset (1 5
```

Will return all elements with 1 < score <= 5 while:

```
ZRANGEBYSCORE zset (5 (10
```

Will return all the elements with 5 < score < 10 (5 and 10 excluded).

Note: **Time complexity**: O(log(N)+M) with N being the number of elements in the sorted set and M the number of elements being returned. If M is constant (e.g. always asking for the first 10 elements with count), you can consider it O(log(N)).

Parameters

- **key** (str, bytes) – The key of the sorted set
- **min_score** (str, bytes) – Lowest score definition
- **max_score** (str, bytes) – Highest score definition
- **with_scores** (*bool*) – Return elements and scores
- **offset** – The number of elements to skip
- **count** – The number of elements to return

Return type list

Raises *RedisError*

zrem(*key*, **members*)

Removes the specified members from the sorted set stored at key. Non existing members are ignored.

An error is returned when key exists and does not hold a sorted set.

Note: **Time complexity**: O(M*log(N)) with N being the number of elements in the sorted set and M the number of elements to be removed.

Parameters

- **key** (str, bytes) – The key of the sorted set
- **members** (str, bytes) – One or more member values to remove

Return type int

Raises *RedisError*

zremrangebyscore(*key*, *min_score*, *max_score*)
Removes all elements in the sorted set stored at key with a score between min and max.

Intervals are described in *zrangebyscore()*.

Returns the number of elements removed.

Note: **Time complexity**: `O(log(N)+M)` with `N` being the number of elements in the sorted set and M the number of elements removed by the operation.

Parameters

- **key** (`str`, `bytes`) – The key of the sorted set
- **min_score** (`str`, `bytes`) – Lowest score definition
- **max_score** (`str`, `bytes`) – Highest score definition

Return type int

Raises *RedisError*

zrevrange(*key*, *start=0*, *stop=-1*, *with_scores=False*)
Returns the specified range of elements in the sorted set stored at key. The elements are considered to be ordered from the highest to the lowest score. Descending lexicographical order is used for elements with equal score.

Apart from the reversed ordering, *zrevrange()* is similar to *zrange()* .

Note: **Time complexity**: `O(log(N)+M)` with `N` being the number of elements in the sorted set and `M` the number of elements returned.

Parameters

- **key** (`str`, `bytes`) – The key of the sorted set
- **start** (*int*) – The starting index of the sorted set
- **stop** (*int*) – The ending index of the sorted set
- **with_scores** (*bool*) – Return the scores with the elements

Return type list

Raises *RedisError*

zscore(*key*, *member*)
Returns the score of member in the sorted set at key. If member does not exist in the sorted set, or key does not exist None is returned.

Note: **Time complexity**: `O(1)`

Parameters

- **key** (`str`, `bytes`) – The key of the set to check for membership in

- **member** (str, bytes) – The value to check for set membership with

Return type str or None

Raises *RedisError*

2.2 Exceptions

class tredis.exceptions.**TRedisException**
Raised as a top-level exception class for all exceptions raised by *RedisClient*.

class tredis.exceptions.**ConnectError**
Raised when *RedisClient* can not connect to the specified Redis server.

class tredis.exceptions.**ConnectionError**
Raised when *RedisClient* has had its connection to the Redis server interrupted unexpectedly.

class tredis.exceptions.**InvalidClusterCommand**
Raised when a method is invoked that is not able to be used when acting as a client for a Redis cluster.

class tredis.exceptions.**AuthError**
Raised when *auth()* is invoked and the Redis server returns an error.

class tredis.exceptions.**RedisError**
Raised when the Redis server returns a error to *RedisClient*. The string representation of this class will contain the error response from the Redis server, if one is sent.

2.3 Supported Commands

The following table summarizes the number of commands supported by category:

Category	Count	Version Added
Cluster	2 of 20	0.7.0
Connection	5 of 5	0.1.0
Geo	0 of 6	—
Hashes	13 of 15	0.4.0
HyperLogLog	3 of 3	0.2.0
Keys	22 of 22	0.1.0
Lists	9 of 17	0.8.0
Pub/Sub	0 of 6	•
Scripting	6 of 6	0.3.0
Server	7 of 30	0.1.0+
Sets	15 of 15	0.1.0
Sorted Sets	8 of 21	0.4.0+
Strings	23 of 23	0.2.0
Transactions	0 of 5	—

2.4 Example

The following examples expect a pre-existing asynchronous application:

Listing 1: A simple set and get of a key from Redis

```
import logging
import pprint

from tornado import gen, ioloop
import tredis


@gen.engine
def run():
    client = tredis.Client([{"host": "127.0.0.1", "port": 6379, "db": 0}],
                           auto_connect=False)
    yield client.connect()
    yield client.set("foo", "bar")
    value = yield client.get("foo")
    pprint.pprint(value)
    ioloop.IOLoop.current().stop()

if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    io_loop = ioloop.IOLoop.current()
    io_loop.add_callback(run)
    io_loop.start()
```

2.5 Version History

- 0.8.0 - released *2018-07-20*
  - Add List commands (9 of 17) (#7 - dave-shawley)
  - Add *zcard()* (#8 - ibnpaul)
  - Add *zscore()* (#8 - ibnpaul)
  - Documentation fixes (#6 - Zephor5)
- 0.7.0 - released *2017-02-03*
  - Add *zrange()*
  - Add *zrevrange()*
- 0.7.0 - released *2017-02-02*
  - Add support for Redis Clusters in the new *Client* class
  - Add *cluster_info()* and *cluster_nodes()*
- 0.6.0 - released *2017-01-27*
  - Add *zrem()* to the Sorted Sets commands
  - Locate master and reconnect when a `READONLY` response is received
  - Add *time()* command
- 0.5.0 - released *2016-11-08*
  - Add Hash commands (13 of 15)

  - Add Sorted Sets commands (3 of 21)
- 0.4.0 - released *2016-01-25*
  - Add `info` command
- 0.3.0 - released *2016-01-18*
  - Remove broken pipelining implementation
  - Add scripting commands
- 0.2.1 - released *2015-11-23*
  - Add hiredis to the requirements
- 0.2.0 - released *2015-11-23*
  - Add per-command execution locking, preventing errors with concurrency in command processing - Clean up connection logic to simplify connecting to exist within the command execution lock instead of maintaining its own event
  - Add all missing methods in the strings category
  - Add hyperloglog methods
  - Add support for mixins to extend core tredis.RedisClient methods in future versions
  - Significant updates to docstrings
- 0.1.0 - released *2015-11-20*
  - initial version

CHAPTER 3

Issues

Please report any issues to the Github repo at https://github.com/gmr/tredis/issues

CHAPTER 4

Source

TRedis source is available on Github at https://github.com/gmr/tredis

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

I

K

L

M

O

P

Q

R

S

T

W

Z