redis-py Documentation

Release 2.10.5

Andy McCurdy, Mahdi Yusuf

August 17, 2016

CHAPTER 1

Redis

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

Provides backwards compatibility with older versions of redis-py that changed arguments to some commands to be more Pythonic, sane, or by accident.

`lrem`(*name*, *value*, *num=0*)

Remove the first `num` occurrences of elements equal to `value` from the list stored at `name`.

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

`pipeline`(*transaction=True*, *shard_hint=None*)

Return a new pipeline object that can queue multiple commands for later execution. `transaction` indicates whether all commands should be executed atomically. Apart from making a group of operations atomic, pipelines are useful for reducing the back-and-forth overhead between the client and server.

`setex`(*name*, *value*, *time*)

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

`zadd`(*name*, **args*, ***kwargs*)

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

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

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

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

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

CHAPTER 2

StrictRedis

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

Implementation of the Redis protocol.

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

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

`append`(*key*, *value*)

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

`bgrewriteaof`()

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

`bgsave`()

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

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

Returns the count of set bits in the value of `key`. Optional `start` and `end` paramaters indicate which bytes to consider

`bitop`(*operation*, *dest*, **keys*)

Perform a bitwise operation using `operation` between `keys` and store the result in `dest`.

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

Return the position of the first bit set to 1 or 0 in a string. `start` and `end` difines search range. The range is interpreted as a range of bytes and not a range of bits, so start=0 and end=2 means to look at the first three bytes.

`blpop`(*keys*, *timeout=0*)

LPOP a value off of the first non-empty list named in the `keys` list.

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

If timeout is 0, then block indefinitely.

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

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

If timeout is 0, then block indefinitely.

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

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

`client_getname`()
Returns the current connection name

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

`client_list`()
Returns a list of currently connected clients

`client_setname`(*name*)
Sets the current connection name

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

`config_resetstat`()
Reset runtime statistics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

flushdb()
Delete all keys in the current database

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

For example:

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

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

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

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

3.The `db` argument to this function.

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

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

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

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

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

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

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

The units must be one o fthe following : m, km mi, ft. By default

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  `match` allows for filtering the keys by pattern

  `count` allows for hint the minimum number of returns

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

  `match` allows for filtering the keys by pattern

  `count` allows for hint the minimum number of returns

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`lock_class` forces the specified lock implementation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ping()
Ping the Redis server

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

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

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

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

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

randomkey()
Returns the name of a random key

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

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

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

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

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

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

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

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

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

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

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

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

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

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

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

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

script_flush()
Flush all scripts from the script cache

script_kill()
Kill the currently executing Lua script

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Returns the length of the new string.

shutdown ()
Shutdown the server

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

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

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

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

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

slowlog_len ()
Get the number of items in the slowlog

slowlog_reset ()
Remove all items in the slowlog

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

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

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

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

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

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

`desc` allows for reversing the sort

`alpha` allows for sorting lexicographically rather than numerically

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

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

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

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

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

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

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

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

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

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

score_cast_func' a callable used to cast the score return value

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

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

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

Returns the number of elements removed.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

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

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

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

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

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

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

Connections

class redis.**Connection**(*host='localhost'*, *port=6379*, *db=0*, *password=None*, *socket_timeout=None*, *socket_connect_timeout=None*, *socket_keepalive=False*, *socket_keepalive_options=None*, *retry_on_timeout=False*, *encoding='utf-8'*, *encoding_errors='strict'*, *decode_responses=False*, *parser_class=<class 'redis.connection.PythonParser'>*, *socket_read_size=65536*)

Manages TCP communication to and from a Redis server

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

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

disconnect()
: Disconnects from the Redis server

encode(*value*)
: Return a bytestring representation of the value

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

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

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

read_response()
: Read the response from a previously sent command

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

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

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

Generic connection pool

disconnect()
: Disconnects all connections in the pool

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

For example:

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

Three URL schemes are supported: redis:// creates a normal TCP socket connection rediss:// creates a SSL wrapped TCP socket connection unix:// creates a Unix Domain Socket connection

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

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

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

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

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

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

`make_connection`()
Create a new connection

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

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

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

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

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

Use `max_connections` to increase / decrease the pool size:

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

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

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

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

`disconnect`()
Disconnects all connections in the pool.

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

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

`make_connection`()
Make a fresh connection.

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

class `redis.`**`SSLConnection`**(*ssl_keyfile=None*, *ssl_certfile=None*, *ssl_cert_reqs=None*, *ssl_ca_certs=None*, ***kwargs*)

class `redis.`**`UnixDomainSocketConnection`**(*path=''*, *db=0*, *password=None*, *socket_timeout=None*, *encoding='utf-8'*, *encoding_errors='strict'*, *decode_responses=False*, *retry_on_timeout=False*, *parser_class=<class 'redis.connection.PythonParser'>*, *socket_read_size=65536*)

CHAPTER 4

Utils

`redis.`**`from_url`**`(`*url*, *db=None*, ***kwargs*`)`

Returns an active Redis client generated from the given database URL.

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

CHAPTER 5

Exceptions

class redis.**AuthenticationError**

class redis.**BusyLoadingError**

class redis.**ConnectionError**

class redis.**DataError**

class redis.**InvalidResponse**

class redis.**PubSubError**

class redis.**ReadOnlyError**

class redis.**RedisError**

class redis.**ResponseError**

class redis.**TimeoutError**

class redis.**WatchError**

CHAPTER 6

CHANGELOG

- **2.10.6 (in development)**
  - Socket timeouts when connecting to a server are now properly raised as TimeoutErrors.
- **2.10.5**
  - Allow URL encoded parameters in Redis URLs. Characters like a "/" can now be URL encoded and redis-py will correctly decode them. Thanks Paul Keene.
  - Added support for the WAIT command. Thanks https://github.com/eshizhan
  - Better shutdown support for the PubSub Worker Thread. It now properly cleans up the connection, unsubscribes from any channels and patterns previously subscribed to and consumes any waiting messages on the socket.
  - Added the ability to sleep for a brief period in the event of a WatchError occuring. Thanks Joshua Harlow.
  - Fixed a bug with pipeline error reporting when dealing with characters in error messages that could not be encoded to the connection's character set. Thanks Hendrik Muhs.
  - Fixed a bug in Sentinel connections that would inadvertantly connect to the master when the connection pool resets. Thanks https://github.com/df3n5
  - Better timeout support in Pubsub get_message. Thanks Andy Isaacson.
  - Fixed a bug with the HiredisParser that would cause the parser to get stuck in an endless loop if a specific number of bytes were delivered from the socket. This fix also increases performance of parsing large responses from the Redis server.
  - Added support for ZREVRANGEBYLEX.
  - ConnectionErrors are now raised if Redis refuses a connection due to the maxclients limit being exceeded. Thanks Roman Karpovich.
  - max_connections can now be set when instantiating client instances. Thanks Ohad Perry.
- **2.10.4** (skipped due to a PyPI snafu)
- **2.10.3**
  - Fixed a bug with the bytearray support introduced in 2.10.2. Thanks Josh Owen.
- **2.10.2**
  - Added support for Hiredis's new bytearray support. Thanks https://github.com/tzickel

  - POSSIBLE BACKWARDS INCOMPATBLE CHANGE: Fixed a possible race condition when multiple threads share the same Lock instance with a timeout. Lock tokens are now stored in thread local storage by default. If you have code that acquires a lock in one thread and passes that lock instance to another thread to release it, you need to disable thread local storage. Refer to the doc strings on the Lock class about the thread_local argument information.
  - Fixed a regression in from_url where "charset" and "errors" weren't valid options. "encoding" and "encoding_errors" are still accepted and preferred.
  - The "charset" and "errors" options have been deprecated. Passing either to StrictRedis.__init__ or from_url will still work but will also emit a DeprecationWarning. Instead use the "encoding" and "encoding_errors" options.
  - Fixed a compatability bug with Python 3 when the server closes a connection.
  - Added BITPOS command. Thanks https://github.com/jettify.
  - Fixed a bug when attempting to send large values to Redis in a Pipeline.

- **2.10.1**
  - Fixed a bug where Sentinel connections to a server that's no longer a master and receives a READONLY error will disconnect and reconnect to the master.
- **2.10.0**
  - Discontinuted support for Python 2.5. Upgrade. You'll be happier.
  - The HiRedis parser will now properly raise ConnectionErrors.
  - Completely refactored PubSub support. Fixes all known PubSub bugs and adds a bunch of new features. Docs can be found in the README under the new "Publish / Subscribe" section.
  - Added the new HyperLogLog commanads (PFADD, PFCOUNT, PFMERGE). Thanks Pepijn de Vos and Vincent Ohprecio.
  - Updated TTL and PTTL commands with Redis 2.8+ semantics. Thanks Markus Kaiserswerth.
  - *SCAN commands now return a long (int on Python3) cursor value rather than the string representation. This might be slightly backwards incompatible in code using *SCAN commands loops such as "while cursor != '0':".
  - Added extra *SCAN commands that return iterators instead of the normal [cursor, data] type. Use scan_iter, hscan_iter, sscan_iter, and zscan_iter for iterators. Thanks Mathieu Longtin.
  - Added support for SLOWLOG commands. Thanks Rick van Hattem.
  - Added lexicographical commands ZRANGEBYLEX, ZREMRANGEBYLEX, and ZLEXCOUNT for sorted sets.
  - Connection objects now support an optional argument, socket_read_size, indicating how much data to read during each socket.recv() call. After benchmarking, increased the default size to 64k, which dramatically improves performance when fetching large values, such as many results in a pipeline or a large (>1MB) string value.
  - Improved the pack_command and send_packed_command functions to increase performance when sending large (>1MB) values.
  - Sentinel Connections to master servers now detect when a READONLY error is encountered and disconnect themselves and all other active connections to the same master so that the new master can be discovered.
  - Fixed Sentinel state parsing on Python 3.

- Added support for SENTINEL MONITOR, SENTINEL REMOVE, and SENTINEL SET commands. Thanks Greg Murphy.
- INFO ouput that doesn't follow the "key:value" format will now be appended to a key named "__raw__" in the INFO dictionary. Thanks Pedro Larroy.
- The "vagrant" directory contains a complete vagrant environment for redis-py developers. The environment runs a Redis master, a Redis slave, and 3 Sentinels. Future iterations of the test sutie will incorporate more integration style tests, ensuring things like failover happen correctly.
- It's now possible to create connection pool instances from a URL. StrictRedis.from_url() now uses this feature to create a connection pool instance and use that when creating a new client instance. Thanks https://github.com/chillipino
- When creating client instances or connection pool instances from an URL, it's now possible to pass additional options to the connection pool with querystring arguments.
- Fixed a bug where some encodings (like utf-16) were unusable on Python 3 as command names and literals would get encoded.
- Added an SSLConnection class that allows for secure connections through stunnel or other means. Construct and SSL connection with the sll=True option on client classes, using the rediss:// scheme from an URL, or by passing the SSLConnection class to a connection pool's connection_class argument. Thanks https://github.com/oranagra.
- Added a socket_connect_timeout option to control how long to wait while establishing a TCP connection before timing out. This lets the client fail fast when attempting to connect to a downed server while keeping a more lenient timeout for all other socket operations.
- Added TCP Keep-alive support by passing use the socket_keepalive=True option. Finer grain control can be achieved using the socket_keepalive_options option which expects a dictionary with any of the keys (socket.TCP_KEEPIDLE, socket.TCP_KEEPCNT, socket.TCP_KEEPINTVL) and integers for values. Thanks Yossi Gottlieb.
- Added a *retry_on_timeout* option that controls how socket.timeout errors are handled. By default it is set to False and will cause the client to raise a TimeoutError anytime a socket.timeout is encountered. If *retry_on_timeout* is set to True, the client will retry a command that timed out once like other 'socket.error's.
- Completely refactored the Lock system. There is now a LuaLock class that's used when the Redis server is capable of running Lua scripts along with a fallback class for Redis servers < 2.6. The new locks fix several subtle race consider that the old lock could face. In additional, a new method, "extend" is available on lock instances that all a lock owner to extend the amount of time they have the lock for. Thanks to Eli Finkelshteyn and https://github.com/chillipino for contributions.

- **2.9.1**
  - IPv6 support. Thanks https://github.com/amashinchi
- **2.9.0**
  - Performance improvement for packing commands when using the PythonParser. Thanks Guillaume Viot.
  - Executing an empty pipeline transaction no longer sends MULTI/EXEC to the server. Thanks EliFinkelshteyn.
  - Errors when authenticating (incorrect password) and selecting a database now close the socket.
  - Full Sentinel support thanks to Vitja Makarov. Thanks!
  - Better repr support for client and connection pool instances. Thanks Mark Roberts.

  - Error messages that the server sends to the client are now included in the client error message. Thanks Sangjin Lim.
  - Added the SCAN, SSCAN, HSCAN, and ZSCAN commands. Thanks Jingchao Hu.
  - ResponseErrors generated by pipeline execution provide addition context including the position of the command in the pipeline and the actual command text generated the error.
  - ConnectionPools now play nicer in threaded environments that fork. Thanks Christian Joergensen.

- **2.8.0**
  - redis-py should play better with gevent when a gevent Timeout is raised. Thanks leifkb.
  - Added SENTINEL command. Thanks Anna Janackova.
  - Fixed a bug where pipelines could potentially corrupt a connection if the MULTI command generated a ResponseError. Thanks EliFinkelshteyn for the report.
  - Connections now call socket.shutdown() prior to socket.close() to ensure communication ends immediately per the note at http://docs.python.org/2/library/socket.html#socket.socket.close Thanks to David Martin for pointing this out.
  - Lock checks are now based on floats rather than ints. Thanks Vitja Makarov.
- **2.7.6**
  - Added CONFIG RESETSTAT command. Thanks Yossi Gottlieb.
  - Fixed a bug introduced in 2.7.3 that caused issues with script objects and pipelines. Thanks Carpentier Pierre-Francois.
  - Converted redis-py's test suite to use the awesome py.test library.
  - Fixed a bug introduced in 2.7.5 that prevented a ConnectionError from being raised when the Redis server is LOADING data.
  - Added a BusyLoadingError exception that's raised when the Redis server is starting up and not accepting commands yet. BusyLoadingError subclasses ConnectionError, which this state previously returned. Thanks Yossi Gottlieb.
- **2.7.5**
  - DEL, HDEL and ZREM commands now return the numbers of keys deleted instead of just True/False.
  - from_url now supports URIs with a port number. Thanks Aaron Westendorf.
- **2.7.4**
  - Added missing INCRBY method. Thanks Krzysztof Dorosz.
  - SET now accepts the EX, PX, NX and XX options from Redis 2.6.12. These options will generate errors if these options are used when connected to a Redis server < 2.6.12. Thanks George Yoshida.
- **2.7.3**
  - Fixed a bug with BRPOPLPUSH and lists with empty strings.
  - All empty except: clauses have been replaced to only catch Exception subclasses. This prevents a KeyboardInterrupt from triggering exception handlers. Thanks Lucian Branescu Mihaila.
  - All exceptions that are the result of redis server errors now share a command Exception subclass, ServerError. Thanks Matt Robenolt.
  - Prevent DISCARD from being called if MULTI wasn't also called. Thanks Pete Aykroyd.

  - SREM now returns an integer indicating the number of items removed from the set. Thanks http://github.com/ronniekk.
  - Fixed a bug with BGSAVE and BGREWRITEAOF response callbacks with Python3. Thanks Nathan Wan.
  - Added CLIENT GETNAME and CLIENT SETNAME commands. Thanks http://github.com/bitterb.
  - It's now possible to use len() on a pipeline instance to determine the number of commands that will be executed. Thanks Jon Parise.
  - Fixed a bug in INFO's parse routine with floating point numbers. Thanks Ali Onur Uyar.
  - Fixed a bug with BITCOUNT to allow *start* and *end* to both be zero. Thanks Tim Bart.
  - The transaction() method now accepts a boolean keyword argument, value_from_callable. By default, or if False is passes, the transaction() method will return the value of the pipelines execution. Otherwise, it will return whatever func() returns.
  - Python3 compatibility fix ensuring we're not already bytes(). Thanks Salimane Adjao Moustapha.
  - Added PSETEX. Thanks YAMAMOTO Takashi.
  - Added a BlockingConnectionPool to limit the number of connections that can be created. Thanks James Arthur.
  - SORT now accepts a *groups* option that if specified, will return tuples of n-length, where n is the number of keys specified in the GET argument. This allows for convenient row-based iteration. Thanks Ionu Arării.

- **2.7.2**
  - Parse errors are now *always* raised on multi/exec pipelines, regardless of the *raise_on_error* flag. See https://groups.google.com/forum/?hl=en&fromgroups=#!topic/redis-db/VUiEFT8U8U0 for more info.

- **2.7.1**
  - Packaged tests with source code

- **2.7.0**
  - Added BITOP and BITCOUNT commands. Thanks Mark Tozzi.
  - Added the TIME command. Thanks Jason Knight.
  - Added support for LUA scripting. Thanks to Angus Peart, Drew Smathers, Issac Kelly, Louis-Philippe Perron, Sean Bleier, Jeffrey Kaditz, and Dvir Volk for various patches and contributions to this feature.
  - Changed the default error handling in pipelines. By default, the first error in a pipeline will now be raised. A new parameter to the pipeline's execute, *raise_on_error*, can be set to False to keep the old behavior of embeedding the exception instances in the result.
  - Fixed a bug with pipelines where parse errors won't corrupt the socket.
  - Added the optional *number* argument to SRANDMEMBER for use with Redis 2.6+ servers.
  - Added PEXPIRE/PEXPIREAT/PTTL commands. Thanks Luper Rouch.
  - Added INCRBYFLOAT/HINCRBYFLOAT commands. Thanks Nikita Uvarov.
  - High precision floating point values won't lose their precision when being sent to the Redis server. Thanks Jason Oster and Oleg Pudeyev.
  - Added CLIENT LIST/CLIENT KILL commands

- **2.6.2**

  - *from_url* is now available as a classmethod on client classes. Thanks Jon Parise for the patch.
  - Fixed several encoding errors resulting from the Python 3.x support.
- **2.6.1**
  - Python 3.x support! Big thanks to Alex Grönholm.
  - Fixed a bug in the PythonParser's read_response that could hide an error from the client (#251).
- **2.6.0**
  - Changed (p)subscribe and (p)unsubscribe to no longer return messages indicating the channel was subscribed/unsubscribed to. These messages are available in the listen() loop instead. This is to prevent the following scenario:
    * Client A is subscribed to "foo"
    * Client B publishes message to "foo"
    * Client A subscribes to channel "bar" at the same time.

    Prior to this change, the subscribe() call would return the published messages on "foo" rather than the subscription confirmation to "bar".
  - Added support for GETRANGE, thanks Jean-Philippe Caruana
  - A new setting "decode_responses" specifies whether return values from Redis commands get decoded automatically using the client's charset value. Thanks to Frankie Dintino for the patch.
- **2.4.13**
  - redis.from_url() can take an URL representing a Redis connection string and return a client object. Thanks Kenneth Reitz for the patch.
- **2.4.12**
  - ConnectionPool is now fork-safe. Thanks Josiah Carson for the patch.
- **2.4.11**
  - AuthenticationError will now be correctly raised if an invalid password is supplied.
  - If Hiredis is unavailable, the HiredisParser will raise a RedisError if selected manually.
  - Made the INFO command more tolerant of Redis changes formatting. Fix for #217.
- **2.4.10**
  - Buffer reads from socket in the PythonParser. Fix for a Windows-specific bug (#205).
  - Added the OBJECT and DEBUG OBJECT commands.
  - Added __del__ methods for classes that hold on to resources that need to be cleaned up. This should prevent resource leakage when these objects leave scope due to misuse or unhandled exceptions. Thanks David Wolever for the suggestion.
  - Added the ECHO command for completeness.
  - Fixed a bug where attempting to subscribe to a PubSub channel of a Redis server that's down would blow out the stack. Fixes #179 and #195. Thanks Ovidiu Predescu for the test case.
  - StrictRedis's TTL command now returns a -1 when querying a key with no expiration. The Redis class continues to return None.
  - ZADD and SADD now return integer values indicating the number of items added. Thanks Homer Strong.

  - Renamed the base client class to StrictRedis, replacing ZADD and LREM in favor of their official argument order. The Redis class is now a subclass of StrictRedis, implementing the legacy redis-py implementations of ZADD and LREM. Docs have been updated to suggesting the use of StrictRedis.
  - SETEX in StrictRedis is now compliant with official Redis SETEX command. the name, value, time implementation moved to "Redis" for backwards compatability.

- **2.4.9**
  - Removed socket retry logic in Connection. This is the responsbility of the caller to determine if the command is safe and can be retried. Thanks David Wolver.
  - Added some extra guards around various types of exceptions being raised when sending or parsing data. Thanks David Wolver and Denis Bilenko.
- **2.4.8**
  - Imported with_statement from __future__ for Python 2.5 compatability.
- **2.4.7**
  - Fixed a bug where some connections were not getting released back to the connection pool after pipeline execution.
  - Pipelines can now be used as context managers. This is the preferred way of use to ensure that connections get cleaned up properly. Thanks David Wolever.
  - Added a convenience method called transaction() on the base Redis class. This method eliminates much of the boilerplate used when using pipelines to watch Redis keys. See the documentation for details on usage.
- **2.4.6**
  - Variadic arguments for SADD, SREM, ZREN, HDEL, LPUSH, and RPUSH. Thanks Raphaël Vinot.
  - (CRITICAL) Fixed an error in the Hiredis parser that occasionally caused the socket connection to become corrupted and unusable. This became noticeable once connection pools started to be used.
  - ZRANGE, ZREVRANGE, ZRANGEBYSCORE, and ZREVRANGEBYSCORE now take an additional optional argument, score_cast_func, which is a callable used to cast the score value in the return type. The default is float.
  - Removed the PUBLISH method from the PubSub class. Connections that are [P]SUBSCRIBEd cannot issue PUBLISH commands, so it doesn't make sense to have it here.
  - Pipelines now contain WATCH and UNWATCH. Calling WATCH or UNWATCH from the base client class will result in a deprecation warning. After WATCHing one or more keys, the pipeline will be placed in immediate execution mode until UNWATCH or MULTI are called. Refer to the new pipeline docs in the README for more information. Thanks to David Wolever and Randall Leeds for greatly helping with this.
- **2.4.5**
  - The PythonParser now works better when reading zero length strings.
- **2.4.4**
  - Fixed a typo introduced in 2.4.3
- **2.4.3**
  - Fixed a bug in the UnixDomainSocketConnection caused when trying to form an error message after a socket error.
- **2.4.2**

  – Fixed a bug in pipeline that caused an exception while trying to reconnect after a connection timeout.

- **2.4.1**
  – Fixed a bug in the PythonParser if disconnect is called before connect.
- **2.4.0**
  – WARNING: 2.4 contains several backwards incompatible changes.

  – Completely refactored Connection objects. Moved much of the Redis protocol packing for requests here, and eliminated the nasty dependencies it had on the client to do AUTH and SELECT commands on connect.

  – Connection objects now have a parser attribute. Parsers are responsible for reading data Redis sends. Two parsers ship with redis-py: a PythonParser and the HiRedis parser. redis-py will automatically use the HiRedis parser if you have the Python hiredis module installed, otherwise it will fall back to the PythonParser. You can force or the other, or even an external one by passing the *parser_class* argument to ConnectionPool.

  – Added a UnixDomainSocketConnection for users wanting to talk to the Redis instance running on a local machine only. You can use this connection by passing it to the *connection_class* argument of the ConnectionPool.

  – Connections no longer derive from threading.local. See threading.local note below.

  – ConnectionPool has been comletely refactored. The ConnectionPool now maintains a list of connections. The redis-py client only hangs on to a ConnectionPool instance, calling get_connection() anytime it needs to send a command. When get_connection() is called, the command name and any keys involved in the command are passed as arguments. Subclasses of ConnectionPool could use this information to identify the shard the keys belong to and return a connection to it. ConnectionPool also implements disconnect() to force all connections in the pool to disconnect from the Redis server.

  – redis-py no longer support the SELECT command. You can still connect to a specific database by specifing it when instantiating a client instance or by creating a connection pool. If you need to talk to multiplate databases within your application, you should use a separate client instance for each database you want to talk to.

  – Completely refactored Publish/Subscribe support. The subscribe and listen commands are no longer available on the redis-py Client class. Instead, the *pubsub* method returns an instance of the PubSub class which contains all publish/subscribe support. Note, you can still PUBLISH from the redis-py client class if you desire.

  – Removed support for all previously deprecated commands or options.

  – redis-py no longer uses threading.local in any way. Since the Client class no longer holds on to a connection, it’s no longer needed. You can now pass client instances between threads, and commands run on those threads will retrieve an available connection from the pool, use it and release it. It should now be trivial to use redis-py with eventlet or greenlet.

  – ZADD now accepts pairs of value=score keyword arguments. This should help resolve the long standing #72. The older value and score arguments have been deprecated in favor of the keyword argument style.

  – Client instances now get their own copy of RESPONSE_CALLBACKS. The new set_response_callback method adds a user defined callback to the instance.

  – Support Jython, fixing #97. Thanks to Adam Vandenberg for the patch.

  – Using __getitem__ now properly raises a KeyError when the key is not found. Thanks Ionu Arării for the patch.

  – Newer Redis versions return a LOADING message for some commands while the database is loading from disk during server start. This could cause problems with SELECT. We now force a socket disconnection prior to raising a ResponseError so subsuquent connections have to reconnect and re-select the appropriate database. Thanks to Benjamin Anderson for finding this and fixing.

- **2.2.4**
  - WARNING: Potential backwards incompatible change - Changed order of parameters of ZREVRANGEBYSCORE to match those of the actual Redis command. This is only backwards-incompatible if you were passing max and min via keyword args. If passing by normal args, nothing in user code should have to change. Thanks Stéphane Angel for the fix.
  - Fixed INFO to properly parse the Redis data correctly for both 2.2.x and 2.3+. Thanks Stéphane Angel for the fix.
  - Lock objects now store their timeout value as a float. This allows floats to be used as timeout values. No changes to existing code required.
  - WATCH now supports multiple keys. Thanks Rich Schumacher.
  - Broke out some code that was Python 2.4 incompatible. redis-py should now be useable on 2.4, but this hasn't actually been tested. Thanks Dan Colish for the patch.
  - Optimized some code using izip and islice. Should have a pretty good speed up on larger data sets. Thanks Dan Colish.
  - Better error handling when submitting an empty mapping to HMSET. Thanks Dan Colish.
  - Subscription status is now reset after every (re)connection.
- **2.2.3**
  - Added support for Hiredis. To use, simply "pip install hiredis" or "easy_install hiredis". Thanks for Pieter Noordhuis for the hiredis-py bindings and the patch to redis-py.
  - The connection class is chosen based on whether hiredis is installed or not. To force the use of the PythonConnection, simply create your own ConnectionPool instance with the connection_class argument assigned to to PythonConnection class.
  - Added missing command ZREVRANGEBYSCORE. Thanks Jay Baird for the patch.
  - The INFO command should be parsed correctly on 2.2.x server versions and is backwards compatible with older versions. Thanks Brett Hoerner.
- **2.2.2**
  - Fixed a bug in ZREVRANK where retriving the rank of a value not in the zset would raise an error.
  - Fixed a bug in Connection.send where the errno import was getting overwritten by a local variable.
  - Fixed a bug in SLAVEOF when promoting an existing slave to a master.
  - Reverted change of download URL back to redis-VERSION.tar.gz. 2.2.1's change of this actually broke Pypi for Pip installs. Sorry!
- **2.2.1**
  - Changed archive name to redis-py-VERSION.tar.gz to not conflict with the Redis server archive.
- **2.2.0**
  - Implemented SLAVEOF
  - Implemented CONFIG as config_get and config_set
  - Implemented GETBIT/SETBIT

- Implemented BRPOPLPUSH
- Implemented STRLEN
- Implemented PERSIST
- Implemented SETRANGE

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index