aioredis Documentation

Release 0.3.1

Alexey Popravka

May 09, 2017

Contents

asyncio (**PEP 3156**) Redis client library.

The library is intended to provide simple and clear interface to Redis based on *asyncio*.

CHAPTER 1

Features

hiredis parser	Yes
Pure-python parser	TBD
Low-level & High-level APIs	Yes
Connections Pool	Yes
Pipelining support	Yes
Pub/Sub support	Yes
Redis Cluster support	WIP
Trollius (python 2.7)	No
Tested python versions	3.3, 3.4, 3.5
Tested for Redis server	2.6, 2.8, 3.0, 3.2
Support for dev Redis server	through low-level API

CHAPTER 2

Installation

The easiest way to install aioredis is by using the package on PyPi:

```
pip install aioredis
```

CHAPTER 3

Requirements

- Python 3.3 and *asyncio* or Python 3.4+
- *hiredis*

CHAPTER 4

Contribute

- Issue Tracker: https://github.com/aio-libs/aioredis/issues
- Source Code: https://github.com/aio-libs/aioredis

Feel free to file an issue or make pull request if you find any bugs or have some suggestions for library improvement.

CHAPTER 5

License

The aioredis is offered under MIT license.

CHAPTER 6

Contents

Getting started

Commands Pipelining

Commands pipelining is built-in.

Every command is sent to transport at-once (ofcourse if no `TypeError`/`ValueError` was raised)

When you making a call with `await` / `yield from` you will be waiting result, and then gather results.

Simple example show both cases (`get source code`):

```
# No pipelining;
async def wait_each_command():
    val = await redis.get('foo')     # wait until `val` is available
    cnt = await redis.incr('bar')    # wait until `cnt` is available
    return val, cnt

# Sending multiple commands and then gathering results
async def pipelined():
    fut1 = redis.get('foo')      # issue command and return future
    fut2 = redis.incr('bar')     # issue command and return future
    # block until results are available
    val, cnt = await asyncio.gather(fut1, fut2)
    return val, cnt
```

Note: For convenience *aioredis* provides *pipeline()* method allowing to execute bulk of commands as one (`get source code`):

```
# Explicit pipeline
async def explicit_pipeline():
```

```
    pipe = redis.pipeline()
    fut1 = pipe.get('foo')
    fut2 = pipe.incr('bar')
    result = await pipe.execute()
    val, cnt = await asyncio.gather(fut1, fut2)
    assert result == [val, cnt]
    return val, cnt
```

Multi/Exec transactions

aioredis provides several ways for executing transactions:

- when using raw connection you can issue `Multi`/`Exec` commands manually;
- when using *aioredis.Redis* instance you can use *multi_exec()* transaction pipeline.

multi_exec() method creates and returns new *MultiExec* object which is used for buffering commands and then executing them inside MULTI/EXEC block.

Here is a simple example (`get source code`):

```
1 async def transaction():
2     tr = redis.multi_exec()
3     future1 = tr.set('foo', '123')
4     future2 = tr.set('bar', '321')
5     result = await tr.execute()
6     assert result == await asyncio.gather(future1, future2)
7     return result
```

As you can notice `await` is **only** used at line 5 with `tr.execute` and **not with** `tr.set(...)` calls.

Warning: It is very important not to `await` buffered command (ie `tr.set('foo', '123')`) as it will block forever.

The following code will block forever:

```
tr = redis.multi_exec()
await tr.incr('foo')    # that's all. we've stuck!
```

Pub/Sub mode

aioredis provides support for Redis Publish/Subscribe messaging.

To switch connection to subscribe mode you must execute `subscribe` command by yield'ing from *subscribe()* it returns a list of *Channel* objects representing subscribed channels.

As soon as connection is switched to subscribed mode the channel will receive and store messages (the `Channel` object is basically a wrapper around `asyncio.Queue`). To read messages from channel you need to use *get()* or *get_json()* coroutines.

Note: In Pub/Sub mode redis connection can only receive messages or issue (P)SUBSCRIBE / (P)UNSUBSCRIBE commands.

Pub/Sub example (get source code):

```
sub = await aioredis.create_redis(
     ('localhost', 6379))

ch1, ch2 = await sub.subscribe('channel:1', 'channel:2')
assert isinstance(ch1, aioredis.Channel)
assert isinstance(ch2, aioredis.Channel)

async def async_reader(channel):
    while await channel.wait_message():
        msg = await channel.get(encoding='utf-8')
        # ... process message ...
        print("message in {}: {}".format(channel.name, msg))

tsk1 = asyncio.ensure_future(async_reader(ch1))

# Or alternatively:

async def async_reader2(channel):
    while True:
        msg = await channel.get(encoding='utf-8')
        if msg is None:
            break
        # ... process message ...
        print("message in {}: {}".format(channel.name, msg))

tsk2 = asyncio.ensure_future(async_reader2(ch2))
```

Pub/Sub example (get source code):

```
async def reader(channel):
    while (await channel.wait_message()):
        msg = await channel.get(encoding='utf-8')
        # ... process message ...
        print("message in {}: {}".format(channel.name, msg))

        if msg == STOPWORD:
            return

with await pool as redis:
    channel, = await redis.subscribe('channel:1')
    await reader(channel)  # wait for reader to complete
    await redis.unsubscribe('channel:1')

# Explicit redis usage
redis = await pool.acquire()
try:
    channel, = await redis.subscribe('channel:1')
    await reader(channel)  # wait for reader to complete
    await redis.unsubscribe('channel:1')
finally:
    pool.release(redis)
```

Python 3.5 `async with` / `async for` support

aioredis is compatible with **PEP 492**.

`Pool` can be used with async with (`get source code`):

```
pool = await aioredis.create_pool(
    ('localhost', 6379))
async with pool.get() as conn:
    value = await conn.get('my-key')
    print('raw value:', value)
```

It also can be used with `await`:

```
pool = await aioredis.create_pool(
    ('localhost', 6379))
# This is exactly the same as:
#   with (yield from pool) as conn:
with await pool as conn:
    value = await conn.get('my-key')
    print('raw value:', value)
```

New `scan`-family commands added with support of async for (`get source code`):

```
redis = await aioredis.create_redis(
    ('localhost', 6379))

async for key in redis.iscan(match='something*'):
    print('Matched:', key)

async for name, val in redis.ihscan(key, match='something*'):
    print('Matched:', name, '->', val)

async for val in redis.isscan(key, match='something*'):
    print('Matched:', val)

async for val, score in redis.izscan(key, match='something*'):
    print('Matched:', val, ':', score)
```

SSL/TLS support

Though Redis server does not support data encryption it is still possible to setup Redis server behind SSL proxy. For such cases *aioredis* library support secure connections through `asyncio` SSL support. See BaseEventLoop.create_connection for details.

`aioredis` — API Reference

Connection

Redis Connection is the core function of the library. Connection instances can be used as is or through *pool* or *high-level API*.

Connection usage is as simple as:

```
import asyncio
import aioredis

async def connect_tcp():
    conn = await aioredis.create_connection(
        ('localhost', 6379))
    val = await conn.execute('GET', 'my-key')

async def connect_unixsocket():
    conn = await aioredis.create_connection(
        '/path/to/redis/socket')
    val = await conn.execute('GET', 'my-key')

asyncio.get_event_loop().run_until_complete(connect_tcp())
asyncio.get_event_loop().run_until_complete(connect_unixsocket())
```

coroutine aioredis.**create_connection**(*address*, ***, *db=0*, *password=None*, *ssl=None*, *encoding=None*, *loop=None*)

Creates Redis connection.

Parameters

- **address** (*tuple or str*) – An address where to connect. Can be a (host, port) tuple or unix domain socket path string.
- **db** (*int*) – Redis database index to switch to when connected.
- **password** (*str or None*) – Password to use if redis server instance requires authorization.
- **ssl** (ssl.SSLContext or True or None) – SSL context that is passed through to asyncio.BaseEventLoop.create_connection().
- **encoding** (*str or None*) – Codec to use for response decoding.
- **loop** (EventLoop) – An optional *event loop* instance (uses asyncio.get_event_loop() if not specified).

Returns *RedisConnection* instance.

class aioredis.**RedisConnection**

Bases: *abc.AbcConnection*

Redis connection interface.

address

Redis server address; either IP-port tuple or unix socket str (*read-only*). IP is either IPv4 or IPv6 depending on resolved host part in initial address.

New in version v0.2.8.

db

Current database index (*read-only*).

encoding

Current codec for response decoding (*read-only*).

closed

Set to True if connection is closed (*read-only*).

in_transaction

Set to True when MULTI command was issued (*read-only*).

pubsub_channels
Read-only dict with subscribed channels. Keys are bytes, values are *Channel* instances.

pubsub_patterns
Read-only dict with subscribed patterns. Keys are bytes, values are *Channel* instances.

in_pubsub
Indicates that connection is in PUB/SUB mode. Provides the number of subscribed channels. *Read-only*.

execute(*command*, **args*, *encoding=_NOTSET*)
Execute Redis command.

The method is **not a coroutine** itself but instead it writes to underlying transport and returns a asyncio.Future waiting for result.

Parameters

- **command** (*str, bytes, bytearray*) – Command to execute
- **encoding** (*str or None*) – Keyword-only argument for overriding response decoding. By default will use connection-wide encoding. May be set to None to skip response decoding.

Raises

- **TypeError** – When any of arguments is None or can not be encoded as bytes.
- ***aioredis.ReplyError*** – For redis error replies.
- ***aioredis.ProtocolError*** – When response can not be decoded and/or connection is broken.

Returns Returns bytes or int reply (or str if encoding was set)

execute_pubsub(*command*, **channels_or_patterns*)
Method to execute Pub/Sub commands. The method is not a coroutine itself but returns a asyncio.gather() coroutine. Method also accept *aioredis.Channel* instances as command arguments:

```
>>> ch1 = Channel('A', is_pattern=False, loop=loop)
>>> await conn.execute_pubsub('subscribe', ch1)
[[b'subscribe', b'A', 1]]
```

Changed in version v0.3: The method accept *Channel* instances.

Parameters

- **command** (*str, bytes, bytearray*) – One of the following Pub/Sub commands: subscribe, unsubscribe, psubscribe, punsubscribe.
- ***channels_or_patterns** – Channels or patterns to subscribe connection to or unsubscribe from. At least one channel/pattern is required.

Returns

Returns a list of subscribe/unsubscribe messages, ex:

```
>>> await conn.execute_pubsub('subscribe', 'A', 'B')
[[b'subscribe', b'A', 1], [b'subscribe', b'B', 2]]
```

close()
Closes connection.

Mark connection as closed and schedule cleanup procedure.

wait_closed()
Coroutine waiting for connection to get closed.

select(*db*)
Changes current db index to new one.

Parameters **db** (*int*) – New redis database index.

Raises

- **TypeError** – When db parameter is not int.
- **ValueError** – When db parameter is less then 0.

Return True Always returns True or raises exception.

auth(*password*)
Send AUTH command.

Parameters **password** (*str*) – Plain-text password

Return bool True if redis replied with 'OK'.

Connections Pool

The library provides connections pool. The basic usage is as follows:

```
import asyncio
import aioredis

async def sample_pool():
    pool = await aioredis.create_pool(('localhost', 6379))
    val = await pool.execute('get', 'my-key')
```

aioredis.**create_pool**(*address*, ***, *db=0*, *password=None*, *ssl=None*, *encoding=None*, *minsize=1*, *maxsize=10*, *commands_factory=_NOTSET*, *loop=None*)
A coroutine that instantiates a pool of *RedisConnection*.

By default it creates pool of *Redis* instances, but it is also possible to create plain connections pool by passing `lambda conn: conn` as *commands_factory*.

Changed in version v0.2.7: `minsize` default value changed from 10 to 1.

Changed in version v0.2.8: Disallow arbitrary RedisPool maxsize.

Deprecated since version v0.2.9: *commands_factory* argument is deprecated and will be removed in *v0.3*.

Parameters

- **address** (*tuple or str*) – An address where to connect. Can be a (host, port) tuple or unix domain socket path string.
- **db** (*int*) – Redis database index to switch to when connected.
- **password** (*str or None*) – Password to use if redis server instance requires authorization.
- **ssl** (ssl.SSLContext or True or None) – SSL context that is passed through to asyncio.BaseEventLoop.create_connection().
- **encoding** (*str or None*) – Codec to use for response decoding.

- **minsize** (*int*) – Minimum number of free connection to create in pool. `1` by default.
- **maxsize** (*int*) – Maximum number of connection to keep in pool. `10` by default. Must be greater then `0`. `None` is disallowed.
- **commands_factory** (*callable*) – A factory to be passed to `create_redis` call. *Redis* by default. **Deprecated** since v0.2.8
- **loop** (EventLoop) – An optional *event loop* instance (uses `asyncio.get_event_loop()` if not specified).

Returns *RedisPool* instance.

class aioredis.**RedisPool**

Redis connections pool.

minsize

A minimum size of the pool (*read-only*).

maxsize

A maximum size of the pool (*read-only*).

size

Current pool size — number of free and used connections (*read-only*).

freesize

Current number of free connections (*read-only*).

db

Currently selected db index (*read-only*).

encoding

Current codec for response decoding (*read-only*).

closed

`True` if pool is closed.

New in version v0.2.8.

coroutine clear()

Closes and removes all free connections in the pool.

coroutine select(*db*)

Changes db index for all free connections in the pool.

Parameters **db** (*int*) – New database index.

coroutine acquire()

Acquires a connection from *free pool*. Creates new connection if needed.

Raises ***aioredis.PoolClosedError*** – if pool is already closed

release(*conn*)

Returns used connection back into pool.

When returned connection has db index that differs from one in pool the connection will be dropped. When queue of free connections is full the connection will be dropped.

Note: This method is **not a coroutine**.

Parameters **conn** (`aioredis.RedisConnection`) – A RedisConnection instance.

close()
Close all free and in-progress connections and mark pool as closed.

New in version v0.2.8.

coroutine wait_closed()
Wait until pool gets closed (when all connections are closed).

New in version v0.2.8.

Pub/Sub Channel object

Channel object is a wrapper around queue for storing received pub/sub messages.

class aioredis.**Channel**(*name*, *is_pattern*, *loop=None*)
Bases: *abc.AbcChannel*

Object representing Pub/Sub messages queue. It's basically a wrapper around asyncio.Queue.

name
Holds encoded channel/pattern name.

is_pattern
Set to True for pattern channels.

is_active
Set to True if there are messages in queue and connection is still subscribed to this channel.

coroutine get(*, *encoding=None*, *decoder=None*)
Coroutine that waits for and returns a message.

Return value is message received or None signifying that channel has been unsubscribed and no more messages will be received.

Parameters

- **encoding** (*str*) – If not None used to decode resulting bytes message.
- **decoder** (*callable*) – If specified used to decode message, ex. json.loads()

Raises ***aioredis.ChannelClosedError*** – If channel is unsubscribed and has no more messages.

get_json(*, *encoding="utf-8"*)
Shortcut to get(encoding="utf-8", decoder=json.loads)

coroutine wait_message()
Waits for message to become available in channel.

Main idea is to use it in loops:

```
>>> ch = redis.channels['channel:1']
>>> while await ch.wait_message():
...     msg = await ch.get()
```

coroutine async-for iter()
Same as *get()* method but it is a native coroutine.

Usage example:

```
>>> async for msg in ch.iter():
...     print(msg)
```

New in version 0.2.5: Available for Python 3.5 only

Exceptions

exception aioredis.**RedisError**
Base exception class for aioredis exceptions.

exception aioredis.**ProtocolError**
Raised when protocol error occurs. When this type of exception is raised connection must be considered broken and must be closed.

exception aioredis.**ReplyError**
Raised for Redis *error replies*.

exception aioredis.**ConnectionClosedError**
Raised if connection to server was lost/closed.

exception aioredis.**PipelineError**
Raised from *pipeline()* if any pipelined command raised error.

exception aioredis.**MultiExecError**
Same as *PipelineError* but raised when executing multi_exec block.

exception aioredis.**WatchVariableError**
Raised if watched variable changed (EXEC returns None). Subclass of *MultiExecError*.

exception aioredis.**ChannelClosedError**
Raised from *aioredis.Channel.get()* when Pub/Sub channel is unsubscribed and messages queue is empty.

exception aioredis.**PoolClosedError**
Raised from *aioredis.RedisPool.acquire()* when pool is already closed.

Exceptions Hierarchy

```
Exception
   RedisError
      ProtocolError
      ReplyError
         PipelineError
            MultiExecError
               WatchVariableError
      ChannelClosedError
      ConnectionClosedError
      PoolClosedError
```

Commands Interface

The library provides high-level API implementing simple interface to Redis commands.

coroutine aioredis.**create_redis**(*address*, ***, *db=0*, *password=None*, *ssl=None*, *encoding=None*, *commands_factory=Redis*, *loop=None*)

This coroutine creates high-level Redis interface instance.

Parameters

- **address** (*tuple or str*) – An address where to connect. Can be a (host, port) tuple or unix domain socket path string.
- **db** (*int*) – Redis database index to switch to when connected.
- **password** (*str or None*) – Password to use if redis server instance requires authorization.
- **ssl** (ssl.SSLContext or True or None) – SSL context that is passed through to asyncio.BaseEventLoop.create_connection().
- **encoding** (*str or None*) – Codec to use for response decoding.
- **commands_factory** (*callable*) – A factory accepting single parameter – *RedisConnection* instance and returning an object providing high-level interface to Redis. *Redis* by default.
- **loop** (EventLoop) – An optional *event loop* instance (uses asyncio.get_event_loop() if not specified).

coroutine aioredis.**create_reconnecting_redis**(*address*, ***, *db=0*, *password=None*, *ssl=None*, *encoding=None*, *commands_factory=Redis*, *loop=None*)

Like *create_redis()* this coroutine creates high-level Redis interface instance that may reconnect to redis server between requests. Accepts same arguments as *create_redis()*.

The reconnect process is done at most once, at the start of the request. So if your request is broken in the middle of sending or receiving reply, it will not be repeated but an exception is raised.

Note: There are two important differences between *create_redis()* and *create_reconnecting_redis()*:

1. The *create_reconnecting_redis()* does not establish connection "right now", it defers connection establishing to the first request.
2. Methods of *Redis()* factory returned do not buffer commands until you *yield from* it. I.e. they are real coroutines not the functions returning future. It may impact your pipelining.

---

class aioredis.**Redis**(*connection*)

High-level Redis commands interface.

For details see *mixins* reference.

aioredis.Redis — Commands Mixins Reference

This section contains reference for mixins implementing Redis commands.

Descriptions are taken from docstrings so may not contain proper markup.

class aioredis.**Redis**(*connection*)
High-level Redis interface.

Gathers in one place Redis commands implemented in mixins.

For commands details see: http://redis.io/commands/#connection

auth(*password*)
Authenticate to server.

This method wraps call to *aioredis.RedisConnection.auth()*

closed
True if connection is closed.

connection
aioredis.RedisConnection instance.

db
Currently selected db index.

echo(*message*, *, *encoding=<object object>*)
Echo the given string.

encoding
Current set codec or None.

in_transaction
Set to True when MULTI command was issued.

ping(*, *encoding=<object object>*)
Ping the server.

quit()
Close the connection.

select(*db*)
Change the selected database for the current connection.

This method wraps call to *aioredis.RedisConnection.select()*

Generic commands

class aioredis.commands.**GenericCommandsMixin**
Generic commands mixin.

For commands details see: http://redis.io/commands/#generic

delete(*key*, **keys*)
Delete a key.

dump(*key*)
Dump a key.

exists(*key*, **keys*)
Check if key(s) exists.

Changed in version v0.2.9: Accept multiple keys; **return** type **changed** from bool to int.

expire(*key*, *timeout*)
Set a timeout on key.

if timeout is float it will be multiplied by 1000 coerced to int and passed to *pexpire* method.

Otherwise raises TypeError if timeout argument is not int.

expireat(*key*, *timestamp*)
Set expire timestamp on key.

if timeout is float it will be multiplied by 1000 coerced to int and passed to *pexpire* method.

Otherwise raises TypeError if timestamp argument is not int.

iscan(*, *match=None*, *count=None*)
Incrementally iterate the keys space using async for.

Usage example:

```
>>> async for key in redis.iscan(match='something*'):
...     print('Matched:', key)
```

keys(*pattern*, *, *encoding=<object object>*)
Returns all keys matching pattern.

migrate(*host*, *port*, *key*, *dest_db*, *timeout*, *copy=False*, *replace=False*)
Atomically transfer a key from a Redis instance to another one.

move(*key*, *db*)
Move key from currently selected database to specified destination.

Raises

- **TypeError** – if db is not int
- **ValueError** – if db is less then 0

object_encoding(*key*)
Returns the kind of internal representation used in order to store the value associated with a key (OBJECT ENCODING).

object_idletime(*key*)
Returns the number of seconds since the object is not requested by read or write operations (OBJECT IDLETIME).

object_refcount(*key*)
Returns the number of references of the value associated with the specified key (OBJECT REFCOUNT).

persist(*key*)
Remove the existing timeout on key.

pexpire(*key*, *timeout*)
Set a milliseconds timeout on key.

Raises **TypeError** – if timeout is not int

pexpireat(*key*, *timestamp*)
Set expire timestamp on key, timestamp in milliseconds.

Raises **TypeError** – if timeout is not int

pttl(*key*)
Returns time-to-live for a key, in milliseconds.

Special return values (starting with Redis 2.8):

•command returns -2 if the key does not exist.

•command returns -1 if the key exists but has no associated expire.

randomkey (*, *encoding=<object object>*)
Return a random key from the currently selected database.

rename (*key*, *newkey*)
Renames key to newkey.

Raises **ValueError** – if key == newkey

renamenx (*key*, *newkey*)
Renames key to newkey only if newkey does not exist.

Raises **ValueError** – if key == newkey

restore (*key*, *ttl*, *value*)
Creates a key associated with a value that is obtained via DUMP.

scan (*cursor=0*, *match=None*, *count=None*)
Incrementally iterate the keys space.

Usage example:

```
>>> match = 'something*'
>>> cur = b'0'
>>> while cur:
...     cur, keys = await redis.scan(cur, match=match)
...     for key in keys:
...         print('Matched:', key)
```

sort (*key*, **get_patterns*, *by=None*, *offset=None*, *count=None*, *asc=None*, *alpha=False*, *store=None*)
Sort the elements in a list, set or sorted set.

ttl (*key*)
Returns time-to-live for a key, in seconds.

Special return values (starting with Redis 2.8): * command returns -2 if the key does not exist. * command returns -1 if the key exists but has no associated expire.

type (*key*)
Returns the string representation of the value's type stored at key.

Python 3.5 async/await support

class aioredis.commands.**GenericCommandsMixin**

async-for iscan (*, *match=None*, *count=None*)
Incrementally iterate the keys space using async for.

Usage example:

```
>>> async for key in redis.iscan(match='something*'):
...     print('Matched:', key)
```

See also *GenericCommandsMixin.scan()*.

Geo commands

class aioredis.commands.**GeoCommandsMixin**
Geo commands mixin.

For commands details see: http://redis.io/commands#geo

geoadd(*key*, *longitude*, *latitude*, *member*, **args*, ***kwargs*)
Add one or more geospatial items in the geospatial index represented using a sorted set.

geodist(*key*, *member1*, *member2*, *unit='m'*)
Returns the distance between two members of a geospatial index.

Return type list[float or None]

geohash(*key*, *member*, **args*, ***kwargs*)
Returns members of a geospatial index as standard geohash strings.

geopos(*key*, *member*, **args*, ***kwargs*)
Returns longitude and latitude of members of a geospatial index.

Return type list[*GeoPoint* or None]

georadius(*key*, *longitude*, *latitude*, *radius*, *unit='m'*, ***, *with_dist=False*, *with_hash=False*, *with_coord=False*, *count=None*, *sort=None*, *encoding=<object object>*)
Query a sorted set representing a geospatial index to fetch members matching a given maximum distance from a point.

Return value follows Redis convention:

•if none of `WITH*` flags are set – list of strings returned:

```
>>> await redis.georadius('Sicily', 15, 37, 200, 'km')
[b"Palermo", b"Catania"]
```

•if any flag (or all) is set – list of named tuples returned:

```
>>> await redis.georadius('Sicily', 15, 37, 200, 'km',
...                       with_dist=True)
[GeoMember(name=b"Palermo", dist=190.4424, hash=None, coord=None),
 GeoMember(name=b"Catania", dist=56.4413, hash=None, coord=None)]
```

Raises

- **TypeError** – radius is not float or int
- **TypeError** – count is not int
- **ValueError** – if unit not equal `m`, `km`, `mi` or `ft`
- **ValueError** – if sort not equal `ASC` or `DESC`

Return type list[str] or list[*GeoMember*]

georadiusbymember(*key*, *member*, *radius*, *unit='m'*, ***, *with_dist=False*, *with_hash=False*, *with_coord=False*, *count=None*, *sort=None*, *encoding=<object object>*)
Query a sorted set representing a geospatial index to fetch members matching a given maximum distance from a member.

Return value follows Redis convention:

•if none of `WITH*` flags are set – list of strings returned:

```
>>> await redis.georadiusbymember('Sicily', 'Palermo', 200, 'km')
[b"Palermo", b"Catania"]
```

•if any flag (or all) is set – list of named tuples returned:

```
>>> await redis.georadiusbymember('Sicily', 'Palermo', 200, 'km',
...                               with_dist=True)
[GeoMember(name=b"Palermo", dist=190.4424, hash=None, coord=None),
 GeoMember(name=b"Catania", dist=56.4413, hash=None, coord=None)]
```

Raises

- **TypeError** – radius is not float or int
- **TypeError** – count is not int
- **ValueError** – if unit not equal `m`, `km`, `mi` or `ft`
- **ValueError** – if sort not equal `ASC` or `DESC`

Return type list[str] or list[*GeoMember*]

class `aioredis.commands.`**GeoPoint**(*longitude*, *latitude*)

class `aioredis.commands.`**GeoMember**(*member*, *dist*, *hash*, *coord*)

Strings commands

class `aioredis.commands.`**StringCommandsMixin**

String commands mixin.

For commands details see: http://redis.io/commands/#string

append(*key*, *value*)

Append a value to key.

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

Count set bits in a string.

Raises **TypeError** – if only start or end specified.

bitop_and(*dest*, *key*, **keys*)

Perform bitwise AND operations between strings.

bitop_not(*dest*, *key*)

Perform bitwise NOT operations between strings.

bitop_or(*dest*, *key*, **keys*)

Perform bitwise OR operations between strings.

bitop_xor(*dest*, *key*, **keys*)

Perform bitwise XOR operations between strings.

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

Find first bit set or clear in a string.

Raises **ValueError** – if bit is not 0 or 1

decr(*key*)

Decrement the integer value of a key by one.

decrby(*key*, *decrement*)

Decrement the integer value of a key by the given number.

Raises **TypeError** – if decrement is not int

get(*key*, ***, *encoding=<object object>*)
Get the value of a key.

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

Raises

- **TypeError** – if offset is not int
- **ValueError** – if offset is less then 0

getrange(*key*, *start*, *end*, ***, *encoding=<object object>*)
Get a substring of the string stored at a key.

Raises **TypeError** – if start or end is not int

getset(*key*, *value*, ***, *encoding=<object object>*)
Set the string value of a key and return its old value.

incr(*key*)
Increment the integer value of a key by one.

incrby(*key*, *increment*)
Increment the integer value of a key by the given amount.

Raises **TypeError** – if increment is not int

incrbyfloat(*key*, *increment*)
Increment the float value of a key by the given amount.

Raises **TypeError** – if increment is not int

mget(*key*, **keys*, *encoding=<object object>*)
Get the values of all the given keys.

mset(*key*, *value*, **pairs*)
Set multiple keys to multiple values.

Raises **TypeError** – if len of pairs is not event number

msetnx(*key*, *value*, **pairs*)
Set multiple keys to multiple values, only if none of the keys exist.

Raises **TypeError** – if len of pairs is not event number

psetex(*key*, *milliseconds*, *value*)
Set the value and expiration in milliseconds of a key.

Raises **TypeError** – if milliseconds is not int

set(*key*, *value*, ***, *expire=0*, *pexpire=0*, *exist=None*)
Set the string value of a key.

Raises **TypeError** – if expire or pexpire is not int

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

Raises

- **TypeError** – if offset is not int
- **ValueError** – if offset is less then 0 or value is not 0 or 1

setex(*key*, *seconds*, *value*)
Set the value and expiration of a key.

If seconds is float it will be multiplied by 1000 coerced to int and passed to *psetex* method.

Raises **TypeError** – if seconds is neither int nor float

setnx(*key*, *value*)
Set the value of a key, only if the key does not exist.

setrange(*key*, *offset*, *value*)
Overwrite part of a string at key starting at the specified offset.

Raises

- **TypeError** – if offset is not int
- **ValueError** – if offset less then 0

strlen(*key*)
Get the length of the value stored in a key.

Hash commands

class aioredis.commands.**HashCommandsMixin**
Hash commands mixin.

For commands details see: http://redis.io/commands#hash

hdel(*key*, *field*, **fields*)
Delete one or more hash fields.

hexists(*key*, *field*)
Determine if hash field exists.

hget(*key*, *field*, *, *encoding=<object object>*)
Get the value of a hash field.

hgetall(*key*, *, *encoding=<object object>*)
Get all the fields and values in a hash.

hincrby(*key*, *field*, *increment=1*)
Increment the integer value of a hash field by the given number.

hincrbyfloat(*key*, *field*, *increment=1.0*)
Increment the float value of a hash field by the given number.

hkeys(*key*, *, *encoding=<object object>*)
Get all the fields in a hash.

hlen(*key*)
Get the number of fields in a hash.

hmget(*key*, *field*, **fields*, *encoding=<object object>*)
Get the values of all the given fields.

hmset(*key*, *field*, *value*, **pairs*)
Set multiple hash fields to multiple values.

hmset_dict(*key*, **args*, ***kwargs*)
Set multiple hash fields to multiple values.

dict can be passed as first positional argument:

```
>>> await redis.hmset_dict(
...     'key', {'field1': 'value1', 'field2': 'value2'})
```

or keyword arguments can be used:

```
>>> await redis.hmset_dict(
...     'key', field1='value1', field2='value2')
```

or dict argument can be mixed with kwargs:

```
>>> await redis.hmset_dict(
...     'key', {'field1': 'value1'}, field2='value2')
```

Note: `dict` and `kwargs` not get mixed into single dictionary, if both specified and both have same key(s) – `kwargs` will win:

```
>>> await redis.hmset_dict('key', {'foo': 'bar'}, foo='baz')
>>> await redis.hget('key', 'foo', encoding='utf-8')
'baz'
```

---

hscan (*key*, *cursor=0*, *match=None*, *count=None*)
Incrementally iterate hash fields and associated values.

hset (*key*, *field*, *value*)
Set the string value of a hash field.

hsetnx (*key*, *field*, *value*)
Set the value of a hash field, only if the field does not exist.

hstrlen (*key*, *field*)
Get the length of the value of a hash field.

hvals (*key*, ***, *encoding=<object object>*)
Get all the values in a hash.

ihscan (*key*, ***, *match=None*, *count=None*)
Incrementally iterate sorted set items using async for.

Usage example:

```
>>> async for name, val in redis.ihscan(key, match='something*'):
...     print('Matched:', name, '->', val)
```

Python 3.5 async/await support

class `aioredis.commands.`**HashCommandsMixin**

async-for ihscan (*key*, ***, *match=None*, *count=None*)
Incrementally iterate sorted set items using async for.

Usage example:

```
>>> async for name, val in redis.ihscan(key, match='something*'):
...     print('Matched:', name, '->', val)
```

See also *HashCommandsMixin.hscan()*.

List commands

class aioredis.commands.**ListCommandsMixin**

List commands mixin.

For commands details see: http://redis.io/commands#list

blpop(*key*, **keys*, *timeout=0*, *encoding=<object object>*)

Remove and get the first element in a list, or block until one is available.

Raises

- **TypeError** – if timeout is not int
- **ValueError** – if timeout is less then 0

brpop(*key*, **keys*, *timeout=0*, *encoding=<object object>*)

Remove and get the last element in a list, or block until one is available.

Raises

- **TypeError** – if timeout is not int
- **ValueError** – if timeout is less then 0

brpoplpush(*sourcekey*, *destkey*, *timeout=0*, *encoding=<object object>*)

Remove and get the last element in a list, or block until one is available.

Raises

- **TypeError** – if timeout is not int
- **ValueError** – if timeout is less then 0

lindex(*key*, *index*, ***, *encoding=<object object>*)

Get an element from a list by its index.

Raises **TypeError** – if index is not int

linsert(*key*, *pivot*, *value*, *before=False*)

Inserts value in the list stored at key either before or after the reference value pivot.

llen(*key*)

Returns the length of the list stored at key.

lpop(*key*, ***, *encoding=<object object>*)

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

lpush(*key*, *value*, **values*)

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

lpushx(*key*, *value*)

Inserts value at the head of the list stored at key, only if key already exists and holds a list.

lrange(*key*, *start*, *stop*, ***, *encoding=<object object>*)

Returns the specified elements of the list stored at key.

Raises **TypeError** – if start or stop is not int

lrem(*key*, *count*, *value*)

Removes the first count occurrences of elements equal to value from the list stored at key.

Raises **TypeError** – if count is not int

lset(*key*, *index*, *value*)
Sets the list element at index to value.

Raises **TypeError** – if index is not int

ltrim(*key*, *start*, *stop*)
Trim an existing list so that it will contain only the specified range of elements specified.

Raises **TypeError** – if start or stop is not int

rpop(*key*, *, *encoding=<object object>*)
Removes and returns the last element of the list stored at key.

rpoplpush(*sourcekey*, *destkey*, *, *encoding=<object object>*)
Atomically returns and removes the last element (tail) of the list stored at source, and pushes the element at the first element (head) of the list stored at destination.

rpush(*key*, *value*, **values*)
Insert all the specified values at the tail of the list stored at key.

rpushx(*key*, *value*)
Inserts value at the tail of the list stored at key, only if key already exists and holds a list.

Set commands

class aioredis.commands.**SetCommandsMixin**
Set commands mixin.

For commands details see: http://redis.io/commands#set

isscan(*key*, *, *match=None*, *count=None*)
Incrementally iterate set elements using async for.

Usage example:

```
>>> async for val in redis.isscan(key, match='something*'):
...     print('Matched:', val)
```

sadd(*key*, *member*, **members*)
Add one or more members to a set.

scard(*key*)
Get the number of members in a set.

sdiff(*key*, **keys*)
Subtract multiple sets.

sdiffstore(*destkey*, *key*, **keys*)
Subtract multiple sets and store the resulting set in a key.

sinter(*key*, **keys*)
Intersect multiple sets.

sinterstore(*destkey*, *key*, **keys*)
Intersect multiple sets and store the resulting set in a key.

sismember(*key*, *member*)
Determine if a given value is a member of a set.

smembers(*key*, *, *encoding=<object object>*)
Get all the members in a set.

smove (*sourcekey*, *destkey*, *member*)
Move a member from one set to another.

spop (*key*, *, *encoding=<object object>*)
Remove and return a random member from a set.

srandmember (*key*, *count=None*, *, *encoding=<object object>*)
Get one or multiple random members from a set.

srem (*key*, *member*, **members*)
Remove one or more members from a set.

sscan (*key*, *cursor=0*, *match=None*, *count=None*)
Incrementally iterate Set elements.

sunion (*key*, **keys*)
Add multiple sets.

sunionstore (*destkey*, *key*, **keys*)
Add multiple sets and store the resulting set in a key.

Python 3.5 async/await support

class aioredis.commands.**SetCommandsMixin**

async-for isscan (*key*, *, *match=None*, *count=None*)
Incrementally iterate set elements using async for.

Usage example:

```
>>> async for val in redis.isscan(key, match='something*'):
...     print('Matched:', val)
```

See also *SetCommandsMixin.sscan()*.

Sorted Set commands

class aioredis.commands.**SortedSetCommandsMixin**
Sorted Sets commands mixin.

For commands details see: http://redis.io/commands/#sorted_set

izscan (*key*, *, *match=None*, *count=None*)
Incrementally iterate sorted set items using async for.

Usage example:

```
>>> async for val, score in redis.izscan(key, match='something*'):
...     print('Matched:', val, ':', score)
```

zadd (*key*, *score*, *member*, **pairs*)
Add one or more members to a sorted set or update its score.

Raises

- **TypeError** – score not int or float
- **TypeError** – length of pairs is not even number

zcard(*key*)
Get the number of members in a sorted set.

zcount(*key*, *min=-inf*, *max=inf*, ***, *exclude=None*)
Count the members in a sorted set with scores within the given values.

Raises

- **TypeError** – min or max is not float or int
- **ValueError** – if min grater then max

zincrby(*key*, *increment*, *member*)
Increment the score of a member in a sorted set.

Raises **TypeError** – increment is not float or int

zinterstore(*destkey*, *key*, **keys*, *with_weights=False*, *aggregate=None*)
Intersect multiple sorted sets and store result in a new key.

Parameters **with_weights** (*bool*) – when set to true each key must be a tuple in form of (key, weight)

zlexcount(*key*, *min=b'-'*, *max=b'+'*, *include_min=True*, *include_max=True*)
Count the number of members in a sorted set between a given lexicographical range.

Raises

- **TypeError** – if min is not bytes
- **TypeError** – if max is not bytes

zrange(*key*, *start=0*, *stop=-1*, *withscores=False*)
Return a range of members in a sorted set, by index.

Raises

- **TypeError** – if start is not int
- **TypeError** – if stop is not int

zrangebylex(*key*, *min=b'-'*, *max=b'+'*, *include_min=True*, *include_max=True*, *offset=None*, *count=None*)
Return a range of members in a sorted set, by lexicographical range.

Raises

- **TypeError** – if min is not bytes
- **TypeError** – if max is not bytes
- **TypeError** – if both offset and count are not specified
- **TypeError** – if offset is not bytes
- **TypeError** – if count is not bytes

zrangebyscore(*key*, *min=-inf*, *max=inf*, *withscores=False*, *offset=None*, *count=None*, ***, *exclude=None*)
Return a range of members in a sorted set, by score.

Raises

- **TypeError** – if min or max is not float or int
- **TypeError** – if both offset and count are not specified
- **TypeError** – if offset is not int

- **TypeError** – if count is not int

zrank(*key*, *member*)
Determine the index of a member in a sorted set.

zrem(*key*, *member*, **members*)
Remove one or more members from a sorted set.

zremrangebylex(*key*, *min=b'-'*, *max=b'+'*, *include_min=True*, *include_max=True*)
Remove all members in a sorted set between the given lexicographical range.

Raises

- **TypeError** – if min is not bytes
- **TypeError** – if max is not bytes

zremrangebyrank(*key*, *start*, *stop*)
Remove all members in a sorted set within the given indexes.

Raises

- **TypeError** – if start is not int
- **TypeError** – if stop is not int

zremrangebyscore(*key*, *min=-inf*, *max=inf*, ***, *exclude=None*)
Remove all members in a sorted set within the given scores.

Raises **TypeError** – if min or max is not int or float

zrevrange(*key*, *start*, *stop*, *withscores=False*)
Return a range of members in a sorted set, by index, with scores ordered from high to low.

Raises **TypeError** – if start or stop is not int

zrevrangebyscore(*key*, *max=inf*, *min=-inf*, ***, *exclude=None*, *withscores=False*, *offset=None*, *count=None*)
Return a range of members in a sorted set, by score, with scores ordered from high to low.

Raises

- **TypeError** – if min or max is not float or int
- **TypeError** – if both offset and count are not specified
- **TypeError** – if offset is not int
- **TypeError** – if count is not int

zrevrank(*key*, *member*)
Determine the index of a member in a sorted set, with scores ordered from high to low.

zscan(*key*, *cursor=0*, *match=None*, *count=None*)
Incrementally iterate sorted sets elements and associated scores.

zscore(*key*, *member*)
Get the score associated with the given member in a sorted set.

zunionstore(*destkey*, *key*, **keys*, *with_weights=False*, *aggregate=None*)
Add multiple sorted sets and store result in a new key.

Python 3.5 async/await support

class aioredis.commands.**SortedSetCommandsMixin**

async-for izscan(*key*, ***, *match=None*, *count=None*)
Incrementally iterate sorted set items using async for.

Usage example:

```
>>> async for val, score in redis.izscan(key, match='something*'):
...     print('Matched:', val, ':', score)
```

See also *SortedSetCommandsMixin.zscan()*.

Server commands

class aioredis.commands.**ServerCommandsMixin**
Server commands mixin.

For commands details see: http://redis.io/commands/#server

bgrewriteaof()
Asynchronously rewrite the append-only file.

bgsave()
Asynchronously save the dataset to disk.

client_getname(*encoding=<object object>*)
Get the current connection name.

client_kill()
Kill the connection of a client.

> **Warning:** Not Implemented

client_list()
Get the list of client connections.

Returns list of ClientInfo named tuples.

client_pause(*timeout*)
Stop processing commands from clients for *timeout* milliseconds.

Raises

- **TypeError** – if timeout is not int
- **ValueError** – if timeout is less then 0

client_setname(*name*)
Set the current connection name.

config_get(*parameter='*'*)
Get the value of a configuration parameter(s).

If called without argument will return all parameters.

Raises **TypeError** – if parameter is not string

config_resetstat()
Reset the stats returned by INFO.

config_rewrite()
Rewrite the configuration file with the in memory configuration.

config_set(*parameter*, *value*)
Set a configuration parameter to the given value.

dbsize()
Return the number of keys in the selected database.

debug_object(*key*)
Get debugging information about a key.

debug_segfault(*key*)
Make the server crash.

flushall()
Remove all keys from all databases.

flushdb()
Remove all keys from the current database.

info(*section='default'*)
Get information and statistics about the server.

If called without argument will return default set of sections. For available sections, see http://redis.io/commands/INFO

Raises **ValueError** – if section is invalid

lastsave()
Get the UNIX time stamp of the last successful save to disk.

monitor()
Listen for all requests received by the server in real time.

> **Warning:** Will not be implemented for now.

role()
Return the role of the server instance.

Returns named tuples describing role of the instance. For fields information see http://redis.io/commands/role#output-format

save()
Synchronously save the dataset to disk.

shutdown(*save=None*)
Synchronously save the dataset to disk and then shut down the server.

slaveof(*host=<object object>*, *port=None*)
Make the server a slave of another instance, or promote it as master.

Calling `slaveof(None)` will send `SLAVEOF NO ONE`.

Changed in version v0.2.6: `slaveof()` form deprecated in favour of explicit `slaveof(None)`.

slowlog_get(*length=None*)
Returns the Redis slow queries log.

slowlog_len(*length=None*)
Returns length of Redis slow queries log.

slowlog_reset()
Resets Redis slow queries log.

sync()
Redis-server internal command used for replication.

time()
Return current server time.

HyperLogLog commands

class aioredis.commands.**HyperLogLogCommandsMixin**
HyperLogLog commands mixin.

For commands details see: http://redis.io/commands#hyperloglog

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

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

pfmerge(*destkey*, *sourcekey*, **sourcekeys*)
Merge N different HyperLogLogs into a single one.

Transaction commands

class aioredis.commands.**TransactionsCommandsMixin**
Transaction commands mixin.

For commands details see: http://redis.io/commands/#transactions

Transactions HOWTO:

```
>>> tr = redis.multi_exec()
>>> result_future1 = tr.incr('foo')
>>> result_future2 = tr.incr('bar')
>>> try:
...     result = await tr.execute()
... except MultiExecError:
...     pass    # check what happened
>>> result1 = await result_future1
>>> result2 = await result_future2
>>> assert result == [result1, result2]
```

multi_exec()
Returns MULTI/EXEC pipeline wrapper.

Usage:

```
>>> tr = redis.multi_exec()
>>> fut1 = tr.incr('foo')   # NO `await` as it will block forever!
>>> fut2 = tr.incr('bar')
>>> result = await tr.execute()
>>> result
[1, 1]
```

```
>>> await asyncio.gather(fut1, fut2)
[1, 1]
```

pipeline()
Returns *Pipeline* object to execute bulk of commands.

It is provided for convenience. Commands can be pipelined without it.

Example:

```
>>> pipe = redis.pipeline()
>>> fut1 = pipe.incr('foo') # NO `await` as it will block forever!
>>> fut2 = pipe.incr('bar')
>>> result = await pipe.execute()
>>> result
[1, 1]
>>> await asyncio.gather(fut1, fut2)
[1, 1]
>>> #
>>> # The same can be done without pipeline:
>>> #
>>> fut1 = redis.incr('foo')    # the 'INCRY foo' command already sent
>>> fut2 = redis.incr('bar')
>>> await asyncio.gather(fut1, fut2)
[2, 2]
```

unwatch()
Forget about all watched keys.

watch(*key*, **keys*)
Watch the given keys to determine execution of the MULTI/EXEC block.

class aioredis.commands.**Pipeline**(*connection*, *commands_factory=lambda conn: conn*, ***, *loop=None*)
Commands pipeline.

Buffers commands for execution in bulk.

This class implements *__getattr__* method allowing to call methods on instance created with `commands_factory`.

Parameters

- **connection** (aioredis.RedisConnection) – Redis connection
- **commands_factory** (*callable*) – Commands factory to get methods from.
- **loop** (EventLoop) – An optional *event loop* instance (uses `asyncio.get_event_loop()` if not specified).

coroutine execute(***, *return_exceptions=False*)
Executes all buffered commands and returns result.

Any exception that is raised by any command is caught and raised later when processing results.

If `return_exceptions` is set to `True` then all collected errors are returned in resulting list otherwise single *aioredis.PipelineError* exception is raised (containing all collected errors).

Parameters **return_exceptions** (*bool*) – Raise or return exceptions.

Raises ***aioredis.PipelineError*** – Raised when any command caused error.

class aioredis.commands.**MultiExec**(*connection*, *commands_factory=lambda conn: conn*, ***, *loop=None*)

Bases: *Pipeline*.

Multi/Exec pipeline wrapper.

See *Pipeline* for parameters description.

coroutine execute(***, *return_exceptions=False*)

Executes all buffered commands and returns result.

see *Pipeline.execute()* for details.

Parameters **return_exceptions** (*bool*) – Raise or return exceptions.

Raises

- ***aioredis.MultiExecError*** – Raised instead of *aioredis.PipelineError*
- ***aioredis.WatchVariableError*** – If watched variable is changed

Scripting commands

class aioredis.commands.**ScriptingCommandsMixin**

Set commands mixin.

For commands details see: http://redis.io/commands#scripting

eval(*script*, *keys=[]*, *args=[]*)

Execute a Lua script server side.

evalsha(*digest*, *keys=[]*, *args=[]*)

Execute a Lua script server side by its SHA1 digest.

script_exists(*digest*, **digests*)

Check existence of scripts in the script cache.

script_flush()

Remove all the scripts from the script cache.

script_kill()

Kill the script currently in execution.

script_load(*script*)

Load the specified Lua script into the script cache.

Server commands

class aioredis.commands.**ServerCommandsMixin**

Server commands mixin.

For commands details see: http://redis.io/commands/#server

bgrewriteaof()

Asynchronously rewrite the append-only file.

bgsave()

Asynchronously save the dataset to disk.

client_getname(*encoding=<object object>*)

Get the current connection name.

client_kill()
Kill the connection of a client.

> **Warning:** Not Implemented

client_list()
Get the list of client connections.

Returns list of ClientInfo named tuples.

client_pause(*timeout*)
Stop processing commands from clients for *timeout* milliseconds.

Raises

- **TypeError** – if timeout is not int
- **ValueError** – if timeout is less then 0

client_setname(*name*)
Set the current connection name.

config_get(*parameter='*'*)
Get the value of a configuration parameter(s).

If called without argument will return all parameters.

Raises **TypeError** – if parameter is not string

config_resetstat()
Reset the stats returned by INFO.

config_rewrite()
Rewrite the configuration file with the in memory configuration.

config_set(*parameter*, *value*)
Set a configuration parameter to the given value.

dbsize()
Return the number of keys in the selected database.

debug_object(*key*)
Get debugging information about a key.

debug_segfault(*key*)
Make the server crash.

flushall()
Remove all keys from all databases.

flushdb()
Remove all keys from the current database.

info(*section='default'*)
Get information and statistics about the server.

If called without argument will return default set of sections. For available sections, see http://redis.io/commands/INFO

Raises **ValueError** – if section is invalid

lastsave()
Get the UNIX time stamp of the last successful save to disk.

monitor()
Listen for all requests received by the server in real time.

> **Warning:** Will not be implemented for now.

role()
Return the role of the server instance.

Returns named tuples describing role of the instance. For fields information see http://redis.io/commands/role#output-format

save()
Synchronously save the dataset to disk.

shutdown(*save=None*)
Synchronously save the dataset to disk and then shut down the server.

slaveof(*host=<object object>*, *port=None*)
Make the server a slave of another instance, or promote it as master.

Calling `slaveof(None)` will send `SLAVEOF NO ONE`.

Changed in version v0.2.6: `slaveof()` form deprecated in favour of explicit `slaveof(None)`.

slowlog_get(*length=None*)
Returns the Redis slow queries log.

slowlog_len(*length=None*)
Returns length of Redis slow queries log.

slowlog_reset()
Resets Redis slow queries log.

sync()
Redis-server internal command used for replication.

time()
Return current server time.

Pub/Sub commands

Also see *aioredis.Channel*.

class aioredis.commands.**PubSubCommandsMixin**
Pub/Sub commands mixin.

For commands details see: http://redis.io/commands/#pubsub

channels
Returns read-only channels dict.

See *pubsub_channels*

in_pubsub
Indicates that connection is in PUB/SUB mode.

Provides the number of subscribed channels.

patterns
Returns read-only patterns dict.

See *pubsub_patterns*

psubscribe(*pattern*, **patterns*)
Switch connection to Pub/Sub mode and subscribe to specified patterns.

Arguments can be instances of *Channel*.

Returns asyncio.gather() coroutine which when done will return a list of subscribed *Channel* objects with is_pattern property set to True.

publish(*channel*, *message*)
Post a message to channel.

publish_json(*channel*, *obj*)
Post a JSON-encoded message to channel.

pubsub_channels(*pattern=None*)
Lists the currently active channels.

pubsub_numpat()
Returns the number of subscriptions to patterns.

pubsub_numsub(**channels*)
Returns the number of subscribers for the specified channels.

punsubscribe(*pattern*, **patterns*)
Unsubscribe from specific patterns.

Arguments can be instances of *Channel*.

subscribe(*channel*, **channels*)
Switch connection to Pub/Sub mode and subscribe to specified channels.

Arguments can be instances of *Channel*.

Returns asyncio.gather() coroutine which when done will return a list of *Channel* objects.

unsubscribe(*channel*, **channels*)
Unsubscribe from specific channels.

Arguments can be instances of *Channel*.

Cluster commands

Warning: Current release (0.3.1) of the library **does not support** Redis Cluster in a full manner. It provides only several API methods which may be changed in future.

aioredis.abc — Interfaces Reference

This module defines several abstract classes that must be used when implementing custom connection managers or other features.

class aioredis.abc.**AbcConnection**
Bases: abc.ABC

Abstract connection interface.

address
Connection address.

close()
Perform connection(s) close and resources cleanup.

closed
Flag indicating if connection is closing or already closed.

db
Current selected DB index.

encoding
Current set connection codec.

execute(*command*, **args*, ***kwargs*)
Execute redis command.

execute_pubsub(*command*, **args*, ***kwargs*)
Execute Redis (p)subscribe/(p)unsubscribe commands.

in_pubsub
Returns number of subscribed channels.

Can be tested as bool indicating Pub/Sub mode state.

pubsub_channels
Read-only channels dict.

pubsub_patterns
Read-only patterns dict.

coroutine wait_closed()
Coroutine waiting until all resources are closed/released/cleaned up.

class aioredis.abc.**AbcPool**
Bases: *aioredis.abc.AbcConnection*

Abstract connections pool interface.

Inherited from AbcConnection so both have common interface for executing Redis commands.

coroutine acquire()
Acquires connection from pool.

address
Connection address or None.

get_connection()
Gets free connection from pool in a sync way.

If no connection available — returns None.

release(*conn*)
Releases connection to pool.

Parameters **conn** (AbcConnection) – Owned connection to be released.

class aioredis.abc.**AbcChannel**
Bases: abc.ABC

Abstract Pub/Sub Channel interface.

close()
Marks Channel as closed, no more messages will be sent to it.

Called by RedisConnection when channel is unsubscribed or connection is closed.

coroutine get()
Wait and return new message.

Will raise ChannelClosedError if channel is not active.

is_active
Flag indicating that channel has unreceived messages and not marked as closed.

is_pattern
Boolean flag indicating if channel is pattern channel.

name
Encoded channel name or pattern.

put_nowait(*data*)
Send data to channel.

Called by RedisConnection when new message received. For pattern subscriptions data will be a tuple of channel name and message itself.

aioredis.pubsub module

Module provides a Pub/Sub listener interface implementing multi-producers, single-consumer queue pattern.

class aioredis.pubsub.**Receiver**(*loop=None*)
Multi-producers, single-consumer Pub/Sub queue.

Can be used in cases where a single consumer task must read messages from several different channels (where pattern subscriptions may not work well or channels can be added/removed dynamically).

Example use case:

```
>>> from aioredis.pubsub import Receiver
>>> from aioredis.abc import AbcChannel
>>> mpsc = Receiver(loop=loop)
>>> async def reader(mpsc):
...     async for channel, msg in mpsc.iter():
...         assert isinstance(channel, AbcChannel)
...         print("Got {!r} in channel {!r}".format(msg, channel))
>>> asyncio.ensure_future(reader(mpsc))
>>> await redis.subscribe(mpsc.channel('channel:1'),
...                       mpsc.channel('channel:3'))
...                       mpsc.channel('channel:5'))
>>> await redis.psubscribe(mpsc.pattern('hello'))
>>> # publishing 'Hello world' into 'hello-channel'
>>> # will print this message:
Got b'Hello world' in channel b'hello-channel'
>>> # when all is done:
>>> await redis.unsubscribe('channel:1', 'channel:3', 'channel:5')
>>> await redis.punsubscribe('hello')
>>> mpsc.stop()
>>> # any message received after stop() will be ignored.
```

To do: few words regarding exclusive channel usage.

channel(*name*)
Create a channel.

Returns _Sender object implementing *AbcChannel*.

channels
Read-only channels dict.

coroutine get(*, *encoding=None*, *decoder=None*)
Wait for and return pub/sub message from one of channels.

Return value is either:

- tuple of two elements: channel & message;
- tuple of three elements: pattern channel, (target channel & message);
- or None in case Receiver is stopped.

Raises *aioredis.ChannelClosedError* – If listener is stopped and all messages have been received.

is_active
Returns True if listener has any active subscription.

iter(*, *encoding=None*, *decoder=None*)
Returns async iterator.

Usage example:

```
>>> async for ch, msg in mpsc.iter():
...     print(ch, msg)
```

pattern(*pattern*)
Create a pattern channel.

Returns `_Sender` object implementing *AbcChannel*.

patterns
Read-only patterns dict.

stop()
Stop receiving messages.

All new messages after this call will be ignored, so you must call unsubscribe before stopping this listener.

coroutine wait_message()
Blocks until new message appear.

class aioredis.pubsub.**_Sender**(*receiver*, *name*, *is_pattern*, *, *loop*)
Write-Only Channel.

Does not allow direct `.get()` calls.

Bases: *aioredis.abc.AbcChannel*

Not to be used directly, returned by *Receiver.channel()* or *Receiver.pattern()* calls.

Examples of aioredis usage

Below is a list of examples from aioredis/examples (see for more).

Every example is a correct python program that can be executed.

Python 3.5 examples

Low-level connection usage example

get source code

```
import asyncio
import aioredis


def main():
    loop = asyncio.get_event_loop()

    async def go():
        conn = await aioredis.create_connection(
            ('localhost', 6379), encoding='utf-8')

        ok = await conn.execute('set', 'my-key', 'some value')
        assert ok == 'OK', ok

        str_value = await conn.execute('get', 'my-key')
        raw_value = await conn.execute('get', 'my-key', encoding=None)
        assert str_value == 'some value'
        assert raw_value == b'some value'

        print('str value:', str_value)
        print('raw value:', raw_value)

        # optionally close connection
        conn.close()
    loop.run_until_complete(go())


if __name__ == '__main__':
    main()
```

Connections pool example

get source code

```
import asyncio
import aioredis


def main():
    loop = asyncio.get_event_loop()

    async def go():
        pool = await aioredis.create_pool(
            ('localhost', 6379),
            minsize=5, maxsize=10)
        with await pool as redis:    # high-level redis API instance
            await redis.set('my-key', 'value')
            val = await redis.get('my-key')
        print('raw value:', val)
        pool.close()
        await pool.wait_closed()    # closing all open connections
```

```
    loop.run_until_complete(go())


if __name__ == '__main__':
    main()
```

Commands example

get source code

```
import asyncio
import aioredis


def main():
    loop = asyncio.get_event_loop()

    async def go():
        redis = await aioredis.create_redis(
            ('localhost', 6379))
        await redis.set('my-key', 'value')
        val = await redis.get('my-key')
        print(val)

        # gracefully closing underlying connection
        redis.close()
        await redis.wait_closed()
    loop.run_until_complete(go())


if __name__ == '__main__':
    main()
```

Transaction example

get source code

```
import asyncio
import aioredis


def main():
    loop = asyncio.get_event_loop()

    async def go():
        redis = await aioredis.create_redis(
            ('localhost', 6379))
        await redis.delete('foo', 'bar')
        tr = redis.multi_exec()
        fut1 = tr.incr('foo')
        fut2 = tr.incr('bar')
        res = await tr.execute()
        res2 = await asyncio.gather(fut1, fut2)
        print(res)
```

```
        assert res == res2

        redis.close()
        await redis.wait_closed()

    loop.run_until_complete(go())


if __name__ == '__main__':
    main()
```

Pub/Sub example

get source code

```
import asyncio
import aioredis


def main():
    loop = asyncio.get_event_loop()

    async def reader(ch):
        while (await ch.wait_message()):
            msg = await ch.get_json()
            print("Got Message:", msg)

    async def go():
        pub = await aioredis.create_redis(
            ('localhost', 6379))
        sub = await aioredis.create_redis(
            ('localhost', 6379))
        res = await sub.subscribe('chan:1')
        ch1 = res[0]

        tsk = asyncio.ensure_future(reader(ch1))

        res = await pub.publish_json('chan:1', ["Hello", "world"])
        assert res == 1

        await sub.unsubscribe('chan:1')
        await tsk
        sub.close()
        pub.close()

    loop.run_until_complete(go())


if __name__ == '__main__':
    main()
```

Scan command example

get source code

```
import asyncio
import aioredis


def main():
    """Scan command example."""
    loop = asyncio.get_event_loop()

    async def go():
        redis = await aioredis.create_redis(
            ('localhost', 6379))

        await redis.mset('key:1', 'value1', 'key:2', 'value2')
        cur = b'0'  # set initial cursor to 0
        while cur:
            cur, keys = await redis.scan(cur, match='key:*')
            print("Iteration results:", keys)

        redis.close()
        await redis.wait_closed()
    loop.run_until_complete(go())


if __name__ == '__main__':
    import os
    if 'redis_version:2.6' not in os.environ.get('REDIS_VERSION', ''):
        main()
```

Python 3.4 examples (using `yield from`)

Located in aioredis/examples/py34

Low-level connection usage example

get source code

```
import asyncio
import aioredis


def main():
    loop = asyncio.get_event_loop()

    @asyncio.coroutine
    def go():
        conn = yield from aioredis.create_connection(
            ('localhost', 6379), encoding='utf-8')

        ok = yield from conn.execute('set', 'my-key', 'some value')
        assert ok == 'OK', ok

        str_value = yield from conn.execute('get', 'my-key')
        raw_value = yield from conn.execute('get', 'my-key', encoding=None)
        assert str_value == 'some value'
        assert raw_value == b'some value'
```

```
        print('str value:', str_value)
        print('raw value:', raw_value)

        # optionally close connection
        conn.close()
    loop.run_until_complete(go())


if __name__ == '__main__':
    main()
```

Connections pool example

get source code

```
import asyncio
import aioredis


def main():
    loop = asyncio.get_event_loop()

    @asyncio.coroutine
    def go():
        pool = yield from aioredis.create_pool(
            ('localhost', 6379),
            minsize=5, maxsize=10)
        with (yield from pool) as redis:    # high-level redis API instance
            yield from redis.set('my-key', 'value')
            val = yield from redis.get('my-key')
        print('raw value:', val)
        pool.close()
        yield from pool.wait_closed()    # closing all open connections

    loop.run_until_complete(go())


if __name__ == '__main__':
    main()
```

Commands example

get source code

```
import asyncio
import aioredis


def main():
    loop = asyncio.get_event_loop()

    @asyncio.coroutine
    def go():
        redis = yield from aioredis.create_redis(
```

```
            ('localhost', 6379))
        yield from redis.set('my-key', 'value')
        val = yield from redis.get('my-key')
        print(val)

        # optinally closing underlying connection
        redis.close()
    loop.run_until_complete(go())


if __name__ == '__main__':
    main()
```

Transaction example

get source code

```
import asyncio
import aioredis


def main():
    loop = asyncio.get_event_loop()

    @asyncio.coroutine
    def go():
        redis = yield from aioredis.create_redis(
            ('localhost', 6379))
        yield from redis.delete('foo', 'bar')
        tr = redis.multi_exec()
        fut1 = tr.incr('foo')
        fut2 = tr.incr('bar')
        res = yield from tr.execute()
        res2 = yield from asyncio.gather(fut1, fut2)
        print(res)
        assert res == res2
        redis.close()
        yield from redis.wait_closed()

    loop.run_until_complete(go())


if __name__ == '__main__':
    main()
```

Pub/Sub example

get source code

```
import asyncio
import aioredis


def main():
    loop = asyncio.get_event_loop()
```

```
    @asyncio.coroutine
    def reader(ch):
        while (yield from ch.wait_message()):
            msg = yield from ch.get_json()
            print("Got Message:", msg)

    @asyncio.coroutine
    def go():
        pub = yield from aioredis.create_redis(
            ('localhost', 6379))
        sub = yield from aioredis.create_redis(
            ('localhost', 6379))
        res = yield from sub.subscribe('chan:1')
        ch1 = res[0]

        tsk = asyncio.async(reader(ch1))

        res = yield from pub.publish_json('chan:1', ["Hello", "world"])
        assert res == 1

        yield from sub.unsubscribe('chan:1')
        yield from tsk
        sub.close()
        pub.close()

    loop.run_until_complete(go())


if __name__ == '__main__':
    main()
```

Scan command example

get source code

```
import asyncio
import aioredis


def main():
    """Scan command example."""
    loop = asyncio.get_event_loop()

    @asyncio.coroutine
    def go():
        redis = yield from aioredis.create_redis(
            ('localhost', 6379))

        yield from redis.mset('key:1', 'value1', 'key:2', 'value2')
        cur = b'0'  # set initial cursor to 0
        while cur:
            cur, keys = yield from redis.scan(cur, match='key:*')
            print("Iteration results:", keys)

        redis.close()
        yield from redis.wait_closed()
```

```
    loop.run_until_complete(go())


if __name__ == '__main__':
    import os
    if 'redis_version:2.6' not in os.environ.get('REDIS_VERSION', ''):
        main()
```

Contributing

To start contributing you must read all the following.

First you must fork/clone repo from github:

```
$ git clone git@github.com:aio-libs/aioredis.git
```

Next, you should install all python dependencies, it is as easy as running single command:

```
$ make devel
```

this command will install:

- `sphinx` for building documentation;
- `pytest` for running tests;
- `flake8` for code linting;
- and few other packages.

Code style

Code **must** be pep8 compliant.

You can check it with following command:

```
$ make flake
```

Running tests

You can run tests in any of the following ways:

```
# will run tests in a verbose mode
$ make test
# or
$ py.test

# will run tests with coverage report
$ make cov
# or
$ py.test --cov
```

SSL tests

Running SSL tests requires following additional programs to be installed:

- `openssl` – to generate test key and certificate;
- `socat` – to make SSL proxy;

To install these on Ubuntu and generate test key & certificate run:

```
$ sudo apt-get install socat openssl
$ make certificate
```

Different Redis server versions

To run tests against different redises use `--redis-server` command line option:

```
$ py.test --redis-server=/path/to/custom/redis-server
```

UVLoop

To run tests with *uvloop*:

```
$ pip install uvloop
$ py.test --uvloop
```

Note: Until Python 3.5.2 EventLoop has no `create_future` method so aioredis won't benefit from uvloop's futures.

Writing tests

aioredis uses *pytest* tool.

Tests are located under `/tests` directory.

Pure Python 3.5 tests (ie the ones using `async`/`await` syntax) must be prefixed with `py35_`, for instance see:

```
tests/py35_generic_commands_tests.py
tests/py35_pool_test.py
```

Fixtures

There is a number of fixtures that can be used to write tests:

`loop`
Current event loop used for test. This is a function-scope fixture. Using this fixture will always create new event loop and set global one to None.

```
def test_with_loop(loop):
    @asyncio.coroutine
    def do_something():
```

```
        pass
    loop.run_until_complete(do_something())
```

unused_port()
Finds and returns free TCP port.

```
def test_bind(unused_port):
    port = unused_port()
    assert 1024 < port <= 65535
```

coroutine create_connection(**args*, ***kw*)
Wrapper around *aioredis.create_connection()*. Only difference is that it registers connection to be closed after test case, so you should not be worried about unclosed connections.

coroutine create_redis(**args*, ***kw*)
Wrapper around *aioredis.create_redis()*.

coroutine create_pool(**args*, ***kw*)
Wrapper around *aioredis.create_pool()*.

redis
Redis client instance.

pool
RedisPool instance.

server
Redis server instance info. Namedtuple with following properties:

name server instance name.

port Bind port.

unixsocket Bind unixsocket path.

version Redis server version tuple.

serverB
Second predefined Redis server instance info.

start_server(*name*)
Start Redis server instance. Redis instances are cached by name.

Returns server info tuple, see *server*.

Return type tuple

ssl_proxy(*unsecure_port*)
Start SSL proxy.

Parameters **unsecure_port** (*int*) – Redis server instance port

Returns secure_port and ssl_context pair

Return type tuple

Helpers

aioredis also updates *pytest*'s namespace with several helpers.

pytest.**redis_version**(**version*, *reason*)
Marks test with minimum redis version to run.

Example:

```
@pytest.redis_version(3, 2, 0, reason="HSTRLEN new in redis 3.2.0")
def test_hstrlen(redis):
    pass
```

pytest.**logs**(*logger*, *level=None*)
Adopted version of `unittest.TestCase.assertEqual()`, see it for details.

Example:

```
def test_logs(create_connection, server):
    with pytest.logs('aioredis', 'DEBUG') as cm:
        conn yield from create_connection(server.tcp_address)
    assert cm.output[0].startswith(
      'DEBUG:aioredis:Creating tcp connection')
```

pytest.**assert_almost_equal**(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Adopted version of `unittest.TestCase.assertAlmostEqual()`.

pytest.**raises_regex**(*exc_type*, *message*)
Adopted version of `unittest.TestCase.assertRaisesRegex()`.

Releases

Recent

0.3.1 (2017-05-09)

FIX:

- Fix pubsub Receiver missing iter() method (see #203);

0.3.0 (2017-01-11)

NEW:

- Pub/Sub connection commands accept `Channel` instances (see #168);
- Implement new Pub/Sub MPSC (multi-producers, single-consumer) Queue – `aioredis.pubsub.Receiver` (see #176);
- Add `aioredis.abc` module providing abstract base classes defining interface for basic lib components; (see #176);
- Implement Geo commands support (see #177 and #179);

FIX:

- Minor tests fixes;

MISC:

- Update examples and docs to use `async`/`await` syntax also keeping `yield from` examples for history (see #173);

- Reflow Travis CI configuration; add Python 3.6 section (see #170);
- Add AppVeyor integration to run tests on Windows (see #180);
- Update multiple development requirements;

0.2.9 (2016-10-24)

NEW:

- Allow multiple keys in `EXISTS` command (see #156 and #157);

FIX:

- Close RedisPool when connection to Redis failed (see #136);
- Add simple `INFO` command argument validation (see #140);
- Remove invalid uses of `next()`

MISC:

- Update devel.rst docs; update Pub/Sub Channel docs (cross-refs);
- Update MANIFEST.in to include docs, examples and tests in source bundle;

0.2.8 (2016-07-22)

NEW:

- Add `hmset_dict` command (see #130);
- Add `RedisConnection.address` property;
- RedisPool `minsize`/`maxsize` must not be `None`;
- Implement `close()`/`wait_closed()`/`closed` interface for pool (see #128);

FIX:

- Add test for `hstrlen`;
- Test fixes

MISC:

- Enable Redis 3.2.0 on Travis;
- Add spell checking when building docs (see #132);
- Documentation updated;

0.2.7 (2016-05-27)

- `create_pool()` minsize default value changed to 1;
- Fixed cancellation of wait_closed (see #118);
- Fixed `time()` convertion to float (see #126);
- Fixed `hmset()` method to return bool instead of `b'OK'` (see #126);
- Fixed multi/exec + watch issue (changed watch variable was causing `tr.execute()` to fail) (see #121);

- Replace `asyncio.Future` uses with utility method (get ready to Python 3.5.2 `loop.create_future()`);
- Tests switched from unittest to pytest (see #126);
- Documentation updates;

0.2.6 (2016-03-30)

- Fixed Multi/Exec transactions cancellation issue (see #110 and #114);
- Fixed Pub/Sub subscribe concurrency issue (see #113 and #115);
- Add SSL/TLS support (see #116);
- `aioredis.ConnectionClosedError` raised in `execute_pubsub` as well (see #108);
- `Redis.slaveof()` method signature changed: now to disable replication one should call `redis.slaveof(None)` instead of `redis.slaveof()`;
- More tests added;

0.2.5 (2016-03-02)

- Close all Pub/Sub channels on connection close (see #88);
- Add `iter()` method to `aioredis.Channel` allowing to use it with `async for` (see #89);
- Inline code samples in docs made runnable and downloadable (see #92);
- Python 3.5 examples converted to use `async`/`await` syntax (see #93);
- Fix Multi/Exec to honor encoding parameter (see #94 and #97);
- Add debug message in `create_connection` (see #90);
- Replace `asyncio.async` calls with wrapper that respects asyncio version (see #101);
- Use NODELAY option for TCP sockets (see #105);
- New `aioredis.ConnectionClosedError` exception added. Raised if connection to Redis server is lost (see #108 and #109);
- Fix RedisPool to close and drop connection in subscribe mode on release;
- Fix `aioredis.util.decode` to recursively decode list responses;
- More examples added and docs updated;
- Add google groups link to README;
- Bump year in LICENSE and docs;

0.2.4 (2015-10-13)

- Python 3.5 `async` support:
  - New scan commands API (`iscan`, `izscan`, `ihscan`);
  - Pool made awaitable (allowing `with await pool: ...` and `async with pool.get() as conn:` constructs);
- Fixed dropping closed connections from free pool (see #83);

- Docs updated;

0.2.3 (2015-08-14)

- Redis cluster support work in progress;
- Fixed pool issue causing pool growth over max size & `acquire` call hangs (see #71);
- `info` server command result parsing implemented;
- Fixed behavior of util functions (see #70);
- `hstrlen` command added;
- Few fixes in examples;
- Few fixes in documentation;

0.2.2 (2015-07-07)

- Decoding data with `encoding` parameter now takes into account list (array) replies (see #68);
- `encoding` parameter added to following commands:
  - generic commands: keys, randomkey;
  - hash commands: hgetall, hkeys, hmget, hvals;
  - list commands: blpop, brpop, brpoplpush, lindex, lpop, lrange, rpop, rpoplpush;
  - set commands: smembers, spop, srandmember;
  - string commands: getrange, getset, mget;
- Backward incompatibility:

  `ltrim` command now returns bool value instead of 'OK';
- Tests updated;

0.2.1 (2015-07-06)

- Logging added (aioredis.log module);
- Fixed issue with `wait_message` in pub/sub (see #66);

0.2.0 (2015-06-04)

- Pub/Sub support added;
- Fix in `zrevrangebyscore` command (see #62);
- Fixes/tests/docs;

Historical

0.1.5 (2014-12-09)

- AutoConnector added;
- wait_closed method added for clean connections shutdown;
- `zscore` command fixed;
- Test fixes;

0.1.4 (2014-09-22)

- Dropped following Redis methods – `Redis.multi()`, `Redis.exec()`, `Redis.discard()`;
- `Redis.multi_exec` hack'ish property removed;
- `Redis.multi_exec()` method added;
- High-level commands implemented:
  - generic commands (tests);
  - transactions commands (api stabilization).
- Backward incompatibilities:
  - Following sorted set commands' API changed:

    `zcount`, `zrangebyscore`, `zremrangebyscore`, `zrevrangebyscore`;
  - set string command' API changed;

0.1.3 (2014-08-08)

- RedisConnection.execute refactored to support commands pipelining (see #33);
- Several fixes;
- WIP on transactions and commands interface;
- High-level commands implemented and tested:
  - hash commands;
  - hyperloglog commands;
  - set commands;
  - scripting commands;
  - string commands;
  - list commands;

0.1.2 (2014-07-31)

- `create_connection`, `create_pool`, `create_redis` functions updated: db and password arguments made keyword-only (see #26);
- Fixed transaction handling (see #32);

- Response decoding (see #16);

0.1.1 (2014-07-07)

- Transactions support (in connection, high-level commands have some issues);
- Docs & tests updated.

0.1.0 (2014-06-24)

- Initial release;
- RedisConnection implemented;
- RedisPool implemented;
- Docs for RedisConnection & RedisPool;
- WIP on high-level API.

Glossary

asyncio Reference implementation of **PEP 3156**

See https://pypi.python.org/pypi/asyncio

error replies Redis server replies that start with - (minus) char. Usually starts with `-ERR`.

hiredis Python extension that wraps protocol parsing code in hiredis.

See https://pypi.python.org/pypi/hiredis

pytest A mature full-featured Python testing tool. See http://pytest.org/latest/

uvloop Is an ultra fast implementation of asyncio event loop on top of libuv. See https://github.com/MagicStack/uvloop

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T