walrus Documentation

Release 0.9.2

Charles Leifer

Oct 06, 2022

Contents

walrus

redis toolkit for python

Lightweight Python utilities for working with Redis.

The purpose of walrus is to make working with Redis in Python a little easier. Rather than ask you to learn a new library, walrus subclasses and extends the popular `redis-py` client, allowing it to be used as a drop-in replacement. In addition to all the features in `redis-py`, walrus adds support for some newer commands, including full support for streams and consumer groups.

walrus consists of:

- pythonic container classes for the Redis data-types.
- support for stream APIs, plus regular and blocking `zpop` variants.
- autocomplete
- bloom filter
- cache
- full-text search
- graph store
- rate limiting
- locks
- **experimental** active-record models (secondary indexes, full-text search, composable query filters, etc)
- more? more!

My hope is that walrus saves you time developing your application by providing useful Redis-specific components. If you have an idea for a new feature, please don't hesitate to tell me about it.

CHAPTER 1

Table of contents

Contents:

1.1 Installing and Testing

Most users will want to simply install the latest version, hosted on PyPI:

```
pip install walrus
```

1.1.1 Installing with git

The project is hosted at https://github.com/coleifer/walrus and can be installed using git:

```
git clone https://github.com/coleifer/walrus.git
cd walrus
python setup.py install
```

Note: On some systems you may need to use `sudo python setup.py install` to install walrus system-wide.

1.1.2 Running tests

You can test your installation by running the test suite. Requires a running Redis server.

```
python runtests.py
```

1.2 Getting Started

The purpose of walrus is to make working with Redis in Python a little easier by wrapping rich objects in Pythonic containers.

Let's see how this works by using `walrus` in the Python interactive shell. Make sure you have redis installed and running locally.

1.2.1 Introducing walrus

To begin using walrus, we'll start by importing it and creating a *Database* instance. The `Database` object is a thin wrapper over the redis-py `Redis` class, so any methods available on `Redis` will also be available on the walrus `Database` object.

```
>>> from walrus import *
>>> db = Database(host='localhost', port=6379, db=0)
```

If you like fun names, you can also use `Walrus` instead:

```
>>> from walrus import *
>>> db = Walrus(host='localhost', port=6379, db=0)
```

1.3 Containers

At the most basic level, Redis acts like an in-memory Python dictionary:

```
>>> db['walrus'] = 'tusk'
>>> print db['walrus']
tusk

>>> db['not-here']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/charles/pypath/redis/client.py", line 817, in __getitem__
    raise KeyError(name)
KeyError: 'not-here'

>>> db.get('not-here') is None
True
```

Redis also supports several primitive data-types:

- *Hash*: dictionary
- *List*: linked list
- *Set*
- *ZSet*: a sorted set
- *HyperLogLog*: probabilistic data-structure for cardinality estimation.
- *Array*: like a Python list (custom data type implemented on top of `Hash` using lua scripts).
- *BitField*: a bitmap that supports random access.
- *BloomFilter*: probabilistic data-structure for testing set membership.

- For stream types (*Stream*: and *ConsumerGroup*) see the streams documentation.

Let's see how to use these types.

1.3.1 Hashes

The *Hash* acts like a Python `dict`.

```
>>> h = db.Hash('charlie')
>>> h.update(name='Charlie', favorite_cat='Huey')
<Hash "charlie": {'name': 'Charlie', 'favorite_cat': 'Huey'}>
```

We can use common Python interfaces like iteration, len, contains, etc.

```
>>> print h['name']
Charlie

>>> for key, value in h:
...     print key, '=>', value
name => Charlie
favorite_cat => Huey

>>> del h['favorite_cat']
>>> h['age'] = 31
>>> print h
<Hash "charlie": {'age': '31', 'name': 'Charlie'}>

>>> 'name' in h
True
>>> len(h)
2
```

1.3.2 Lists

The *List* acts like a Python `list`.

```
>>> l = db.List('names')
>>> l.extend(['charlie', 'huey', 'mickey', 'zaizee'])
4L
>>> print l[:2]
['charlie', 'huey']
>>> print l[-2:]
['mickey', 'zaizee']
>>> l.pop()
'zaizee'
>>> l.prepend('scout')
4L
>>> len(l)
4
```

1.3.3 Sets

The *Set* acts like a Python `set`.

```
>>> s1 = db.Set('s1')
>>> s2 = db.Set('s2')
>>> s1.add(*range(5))
5
>>> s2.add(*range(3, 8))
5

>>> s1 | s2
{'0', '1', '2', '3', '4', '5', '6', '7'}
>>> s1 & s2
{'3', '4'}
>>> s1 - s2
{'0', '1', '2'}

>>> s1 -= s2
>>> s1.members()
{'0', '1', '2'}

>>> len(s1)
3
```

1.3.4 Sorted Sets (ZSet)

The *ZSet* acts a bit like a sorted dictionary, where the values are the scores used for sorting the keys.

```
>>> z1 = db.ZSet('z1')
>>> z1.add({'charlie': 31, 'huey': 3, 'mickey': 6, 'zaizee': 2.5})
4
>>> z1['huey'] = 3.5
```

Sorted sets provide a number of complex slicing and indexing options when retrieving values. You can slice by key or rank, and optionally include scores in the return value.

```
>>> z1[:'mickey']  # Who is younger than Mickey?
['zaizee', 'huey']

>>> z1[-2:]  # Who are the two oldest people?
['mickey', 'charlie']

>>> z1[-2:, True]  # Who are the two oldest, and what are their ages?
[('mickey', 6.0), ('charlie', 31.0)]
```

There are quite a few methods for working with sorted sets, so if you're curious then check out the *ZSet* API documentation.

1.3.5 HyperLogLog

The *HyperLogLog* provides an estimation of the number of distinct elements in a collection.

```
>>> hl = db.HyperLogLog('hl')
>>> hl.add(*range(100))
>>> len(hl)
100
>>> hl.add(*range(1, 100, 2))
```

(continues on next page)

(continued from previous page)

```
>>> hl.add(*range(1, 100, 3))
>>> len(hl)
102
```

1.3.6 Arrays

The *Array* type is implemented using lua scripts. Unlike *List* which is implemented as a linked-list, the Array is built on top of a Redis hash and has better run-times for certain operations (indexing, for instance). Like *List*, *Array* acts like a Python list.

```
>>> a = db.Array('arr')
>>> a.extend(['foo', 'bar', 'baz', 'nugget'])
>>> a[-1] = 'nize'
>>> list(a)
['foo', 'bar', 'baz', 'nize']
>>> a.pop(2)
'baz'
```

1.3.7 BitField

The *BitField* type acts as a bitmap that supports random access read, write and increment operations. Operations use a format string (e.g. "u8" for unsigned 8bit integer 0-255, "i4" for signed integer -8-7).

```
>>> bf = db.bit_field('bf')
>>> resp = (bf
...          .set('u8', 8, 255)
...          .get('u8', 0)  # 00000000
...          .get('u4', 8)  # 1111
...          .get('u4', 12)  # 1111
...          .get('u4', 13)  # 111? -> 1110
...          .execute())
...
[0, 0, 15, 15, 14]

>>> resp = (bf
...          .set('u8', 4, 1)  # 00ff -> 001f (returns old val, 0x0f).
...          .get('u16', 0)  # 001f (00011111)
...          .set('u16', 0, 0))  # 001f -> 0000
...
>>> for item in resp:  # bitfield responses are iterable!
...     print(item)
...
15
31
31

>>> resp = (bf
...          .incrby('u8', 8, 254)  # 0000 0000 1111 1110
...          .get('u16', 0)
...          .incrby('u8', 8, 2, 'FAIL')  # increment 254 -> 256? overflow!
...          .incrby('u8', 8, 1)  # increment 254 -> 255. success!
...          .incrby('u8', 8, 1)  # 255->256? overflow, will fail.
...          .get('u16', 0))
```

(continues on next page)

(continued from previous page)

```
...
>>> resp.execute()
[254, 254, None, 255, None, 255]
```

BitField also supports slice notation, using bit-offsets. The return values are always unsigned integers:

```
>>> bf.set('u8', 0, 166).execute()  # 10100110
166

>>> bf[:8]  # Read first 8 bits as unsigned byte.
166

>>> bf[:4]  # 1010
10
>>> bf[4:8]  # 0110
6
>>> bf[2:6]  # 1001
9
>>> bf[6:10]  # 10?? -> 1000
8
>>> bf[8:16]  # ???????? -> 00000000
0

>>> bf[:8] = 89  # 01011001
>>> bf[:8]
89

>>> bf[:8] = 255  # 1111 1111
>>> bf[:4]  # 1111
15
>>> del bf[2:6]  # 1111 1111 -> 1100 0011
>>> bf[:8]  # 1100 0011
195
```

1.3.8 BloomFilter

A *BloomFilter* is a probabilistic data-structure used for answering the question: "is X a member of set S?" The bloom-filter may return a false positive, but it is impossible to receive a false negative (in other words, if the bloom-filter contains a value, it will never erroneously report that it does *not* contain such a value). The accuracy of the bloom-filter and the likelihood of a false positive can be reduced by increasing the size of the bloom-filter buffer. The default size is 64KB (or 524,288 bits).

```
>>> bf = db.bloom_filter('bf')  # Create a bloom-filter, stored in key "bf".

>>> data = ('foo', 'bar', 'baz', 'nugget', 'this is a test', 'testing')
>>> for item in data:
...     bf.add(item)  # Add the above items to the bloom-filter.
...

>>> for item in data:
...     assert item in bf  # Verify that all items are present.
...

>>> for item in data:
...     assert item.upper() not in bf  # FOO, BAR, etc, are *not* present.
```

(continues on next page)

(continued from previous page)

```
...     assert item.title() not in bf  # Foo, Bar, etc, are *not* present.
...
```

BloomFilter implements only two methods:

- *add()* - to add an item to the bloom-filter.
- *contains()* - test whether an item exists in the filter.

Note: Items cannot be removed from a bloom-filter.

Warning: Once a *BloomFilter* has been created and items have been added, you must not modify the size of the buffer.

1.4 Autocomplete

Provide suggestions based on partial string search. Walrus' autocomplete library is based on the implementation from redis-completion.

Note: The walrus implementation of autocomplete relies on the `HSCAN` command and therefore requires Redis >= 2.8.0.

1.4.1 Overview

The *Autocomplete* engine works by storing substrings and mapping them to user-defined data.

Features

- Perform searches using partial words or phrases.
- Store rich metadata along with substrings.
- Boosting.

1.4.2 Simple example

Walrus *Autocomplete* can be used to index words and phrases, and then make suggestions based on user searches.

To begin, call `Database.autocomplete()` to create an instance of the autocomplete index.

```
>>> database = Database()
>>> ac = database.autocomplete()
```

Phrases can be stored by calling *Autocomplete.store()*:

```
>>> phrases = [
...     'the walrus and the carpenter',
...     'walrus tusks',
```

(continues on next page)

(continued from previous page)

```
...     'the eye of the walrus']

>>> for phrase in phrases:
...     ac.store(phrase)
```

To search for results, use *Autocomplete.search()*.

```
>>> ac.search('wal')
['the walrus and the carpenter',
 'walrus tusks',
 'the eye of the walrus']

>>> ac.search('wal car')
['the walrus and the carpenter']
```

To boost a result, we can specify one or more *boosts* when searching:

```
>>> ac.search('wal', boosts={'walrus tusks': 2})
['walrus tusks',
 'the walrus and the carpenter',
 'the eye of the walrus']
```

To remove a phrase from the index, use *Autocomplete.remove()*:

```
>>> ac.remove('walrus tusks')
```

We can also check for the existence of a phrase in the index using *Autocomplete.exists()*:

```
>>> ac.exists('the walrus and the carpenter')
True

>>> ac.exists('walrus tusks')
False
```

1.4.3 Complete example

While walrus can work with just simple words and phrases, the *Autocomplete* index was really developed to be able to provide meaningful typeahead suggestions for sites containing rich content. To this end, the autocomplete search allows you to store arbitrary metadata in the index, which will then be returned when a search is performed.

```
>>> database = Database()
>>> ac = database.autocomplete()
```

Suppose we have a blog site and wish to add search for the entries. We'll use the blog entry's title for the search, and return, along with title, a thumbnail image and a link to the entry's detail page. That way when we display results we have all the information we need to display a nice-looking link:

```
>>> for blog_entry in Entry.select():
...     metadata = {
...         'image': blog_entry.get_primary_thumbnail(),
...         'title': blog_entry.title,
...         'url': url_for('entry_detail', entry_id=blog_entry.id)}
...
...     ac.store(
```

(continues on next page)

(continued from previous page)

```
...         obj_id=blog_entry.id,
...         title=blog_entry.title,
...         data=metadata,
...         obj_type='entry')
```

When we search we receive the metadata that was stored in the index:

```
>>> ac.search('walrus')
[{'image': '/images/walrus-logo.jpg',
  'title': 'Walrus: Lightweight Python utilities for working with Redis',
  'url': '/blog/walrus-lightweight-python-utilities-for-working-with-redis/'},
 {'image': '/images/walrus-tusk.jpg',
  'title': 'Building Autocomplete with Walrus',
  'url': '/blog/building-autocomplete-with-redis/'}]
```

Whenever an entry is created or updated, we will want to update the index. By keying off the entry's primary key and object type (*'entry'*), walrus will handle this correctly:

```
def save_entry(entry):
    entry.save_to_db()  # Save entry to relational database, etc.

    ac.store(
        obj_id=entry.id,
        title=entry.title,
        data={
            'image': entry.get_primary_thumbnail(),
            'title': entry.title,
            'url': url_for('entry_detail', entry_id=entry.id)},
        obj_type='entry')
```

Suppose we have a very popular blog entry that is frequently searched for. We can *boost* that entry's score by calling *boost_object()*:

```
>>> popular_entry = Entry.get(Entry.title == 'Some popular entry')
>>> ac.boost_object(
...     obj_id=popular_entry.id,
...     obj_type='entry',
...     multiplier=2.0)
```

To perform boosts on a one-off basis while searching, we can specify a dictionary mapping object IDs or types to a particular multiplier:

```
>>> ac.search(
...     'some phrase',
...     boosts={popular_entry.id: 2.0, unpopular_entry.id, 0.5})
...
[ list of matching entry's metadata ]
```

To remove an entry from the index, we just need to specify the object's id and type:

```
def delete_entry(entry):
    entry.delete_from_db()  # Remove from relational database, etc.

    ac.remove(
        obj_id=entry.id,
        obj_type='entry')
```

We can also check whether an entry exists in the index:

```
>>> entry = Entry.get(Entry.title == 'Building Autocomplete with Walrus')
>>> ac.exists(entry.id, 'entry')
True
```

1.4.4 Scoring

Walrus implements a scoring algorithm that considers the words and also their position relative to the entire phrase. Let's look at some simple searches. We'll index the following strings:

- "aa bb"
- "aa cc"
- "bb cc"
- "bb aa cc"
- "cc aa bb"

```
>>> phrases = ['aa bb', 'aa cc', 'bb cc', 'bb aa cc', 'cc aa bb']
>>> for phrase in phrases:
...     ac.store(phrase)
```

Note how when we search for *aa* that the results with *aa* towards the front of the string score higher:

```
>>> ac.search('aa')
['aa bb',
 'aa cc',
 'bb aa cc',
 'cc aa bb']
```

This is even more clear when we search for *bb* and *cc*:

```
>>> ac.search('bb')
['bb aa cc',
 'bb cc',
 'aa bb',
 'cc aa bb']

>>> ac.search('cc')
['cc aa bb',
 'aa cc',
 'bb cc',
 'bb aa cc']
```

As you can see, results are scored by the proximity of the match to the front of the string, then alphabetically.

Boosting

To modify the score of certain words or phrases, we can apply *boosts* when searching. Boosts consist of a dictionary mapping identifiers to multipliers. Multipliers greater than 1 will move results to the top, while multipliers between 0 and 1 will push results to the bottom.

In this example, we'll take the 3rd result, *bb cc* and bring it to the top:

```
>>> ac.search('cc', boosts={'bb cc': 2})
['bb cc',
 'cc aa bb',
 'aa cc',
 'bb aa cc']
```

In this example, we'll take the best result, *cc aa bb*, and push it back a spot:

```
>>> ac.search('cc', boosts={'cc aa bb': .75})
['aa cc',
 'cc aa bb',
 'bb cc',
 'bb aa cc']
```

Persisting boosts

While boosts can be specified on a one-off basis while searching, we can also permanently store boosts that will be applied to *all* searches. To store a boost for a particular object or object type, call the *boost_object()* method:

```
>>> ac.boost_object(obj_id='bb cc', multiplier=2.0)
>>> ac.boost_object(obj_id='cc aa bb', multiplier=.75)
```

Now we can search and our boosts will automatically be in effect:

```
>>> ac.search('cc')
['bb cc',
 'aa cc',
 'cc aa bb',
 'bb aa cc']
```

1.4.5 ZRANGEBYLEX

Because I wanted to implement a slightly more complex scoring algorithm, I chose not to use the ZRANGEBYLEX command while implementing autocomplete. For very simple use-cases, though, ZRANGEBYLEX will certainly offer better performance. Depending on your application's needs, you may be able to get by just storing your words in a sorted set and calling ZRANGEBYLEX on that set.

1.5 Cache

Walrus provides a simple *Cache* implementation that makes use of Redis' key expiration feature. The cache can be used to set or retrieve values, and also provides a decorator (*Cache.cached()*) for wrapping function or methods.

1.5.1 Basic usage

You can *get()*, *set()* and *delete()* objects directly from the cache.

```
>>> from walrus import *
>>> db = Database()
>>> cache = db.cache()
```

(continues on next page)

(continued from previous page)

```
>>> cache.set('foo', 'bar', 10)  # Set foo=bar, expiring in 10s.
>>> cache.get('foo')
'bar'

>>> time.sleep(10)
>>> cache.get('foo') is None
True

>>> cache.set_many({'k1': 'v1', 'k2': 'v2'}, 300)
True
>>> cache.get_many(['k1', 'kx', 'k2'])
{'k1': 'v1', 'k2': 'v2'}
>>> cache.delete_many(['k1', 'kx', 'k2'])
2
```

1.5.2 Simple Decorator

The *Cache.cached()* decorator will *memoize* the return values from the wrapped function for the given arguments. One way to visualize this is by creating a function that returns the current time and wrap it with the decorator. The decorated function will run the first time it is called and the return value is stored in the cache. Subsequent calls will not execute the function, but will instead return the cached value from the previous call, until the cached value expires.

```
>>> @cache.cached(timeout=10)
... def get_time():
...     return datetime.datetime.now()

>>> print get_time()  # First call, return value cached.
2015-01-07 18:26:42.730638

>>> print get_time()  # Hits the cache.
2015-01-07 18:26:42.730638

>>> time.sleep(10)  # Wait for cache to expire then call again.
>>> print get_time()
2015-01-07 18:26:53.529011
```

If a decorated function accepts arguments, then values will be cached based on the arguments specified. In the example below we'll pass a garbage argument to the `get_time` function to show how the cache varies for different arguments:

```
>>> @cache.cached(timeout=60)
... def get_time(seed=None):
...     return datetime.datetime.now()

>>> print get_time()
2015-01-07 18:30:53.831977

>>> print get_time()
2015-01-07 18:30:53.831977

>>> print get_time('foo')
2015-01-07 18:30:56.614064

>>> print get_time('foo')
2015-01-07 18:30:56.614064
```

(continues on next page)

(continued from previous page)

```
>>> print get_time('bar')
2015-01-07 18:31:01.497050

>>> print get_time('foo')
2015-01-07 18:30:56.614064
```

To clear the cache, you can call the special `bust()` method on the decorated function:

```
>>> get_time.bust('foo')
>>> print get_time('foo')
2015-01-07 18:31:15.326435
```

1.5.3 Cached Property

Python supports dynamic instance attributes through the `property` decorator. A property looks like a normal instance attribute, but it's value is calculated at run-time. Walrus comes with a special decorator designed for implementing *cached properties*. Here is how you might use *cached_property()*:

```
>>> class Clock(object):
...     @cache.cached_property()
...     def now(self):
...         return datetime.datetime.now()

>>> print clock.now
2015-01-12 21:10:34.335755

>>> print clock.now
2015-01-12 21:10:34.335755
```

1.5.4 Cache Asynchronously

If you have a function that runs slowly and would like to be able to perform other operations while waiting for the return value, you might try the *asynchronous cache decorator*, *cache_async()*.

The *cache_async()* decorator will run the decorated function in a separate thread. The function therefore will return immediately, even though your code may be processing in the background. Calls to the decorated function will return a method on a synchronized queue object. When the value is calculated (or returned from the cache), it will be placed in the queue and you can retrieve it.

Let's see how this works. We'll add a call to `time.sleep` in the decorated function to simulate a function that takes a while to run, and we'll also print a message indicating that we're inside the function body.

```
>>> import time
>>> @cache.cache_async()
... def get_now(seed=None):
...     print 'About to sleep for 5 seconds.'
...     time.sleep(5)
...     return datetime.datetime.now()
```

The first time we call our function we will see the message indicating our function is sleeping, but the function will return immediately! The return value can be used to get the *actual* return value of the decorated function:

```
>>> result = get_now()
About to sleep for 5 seconds.
>>> result
<function _get_value at 0x7fe3a4685de8>
```

If we attempt to check the result immediately, there will be no value because the function is still sleeping. In this case a queue `Empty` exception is raised:

```
>>> result(block=False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/Queue.py", line 165, in get
    raise Empty
Queue.Empty
```

We can force our code to block until the result is ready, though:

```
>>> print result(block=True)
2015-01-12 21:28:25.266448
```

Now that the result has been calculated and cached, a subsequent call to `get_now()` will not execute the function body. We can tell because the function does not print *About to sleep for 5 seconds*.

```
>>> result = get_now()
>>> print result()
2015-01-12 21:28:25.266448
```

The result function can be called any number of times. It will always return the same value:

```
>>> print result()
2015-01-12 21:28:25.266448
```

Another trick is passing a timeout to the result function. Let's see what happens when we call `get_now()` using a different seed, then specify a timeout to block for the return value. Since we hard-coded a delay of 5 seconds, let's see what happens when we specify a timeout of 4 seconds:

```
>>> print get_now('foo')(timeout=4)
About to sleep for 5 seconds.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/charles/pypath/walrus/cache.py", line 160, in _get_value
    result = q.get(block=block, timeout=timeout)
  File "/usr/lib/python2.7/Queue.py", line 176, in get
    raise Empty
Queue.Empty
```

Now let's try with a timeout of 6 seconds (being sure to use a different seed so we trigger the 5 second delay):

```
>>> print get_now('bar')(timeout=6)
About to sleep for 5 seconds.
2015-01-12 21:46:49.060883
```

Since the function returns a value within the given timeout, the value is returned.

1.6 Full-text Search

Walrus comes with a standalone full-text search index that supports:

- Storing documents along with arbitrary metadata.
- Complex search using boolean/set operations and parentheses.
- Stop-word removal.
- Porter-stemming.
- Optional double-metaphone for phonetic search.

To create a full-text index, use:

- `Database.Index()`
- `Index`

Example:

```
from walrus import Database

db = Database()
search_index = db.Index('app-search')

# Phonetic search.
phonetic_index = db.Index('phonetic-search', metaphone=True)
```

1.6.1 Storing data

Use the `Index.add()` method to add documents to the search index:

```
# Specify the document's unique ID and the content to be indexed.
search_index.add('doc-1', 'this is the content of document 1')

# Besides the document ID and content, we can also store metadata, which is
# not searchable, but is returned along with the document content when a
# search is performed.
search_index.add('doc-2', 'another document', title='Another', status='1')
```

To update a document, use either the `Index.update()` or `Index.replace()` methods. The former will update existing metadata while the latter clears any pre-existing metadata before saving.

```
# Update doc-1's content and metadata.
search_index.update('doc-1', 'this is the new content', title='Doc 1')

# Overwrite doc-2...the "status" metadata value set earlier will be lost.
search_index.replace('doc-2', 'another document', title='Another doc')
```

To remove a document use `Index.remove()`:

```
search_index.remove('doc-1')  # Removed from index and removed metadata.
```

1.6.2 Searching

Use the `Index.search()` method to perform searches. The search query can include set operations (e.g. *AND*, *OR*) and use parentheses to indicate operation precedence.

```
for document in search_index.search('python AND flask'):
    # Print the "title" that was stored as metadata. The "content" field
    # contains the original content of the document as it was indexed.
    print(document['title'], document['content'])
```

Phonetic search, using `metaphone`, is tolerant of typos:

```
for document in phonetic_index.search('flasck AND pythonn'):
    print(document['title'], document['content'])
```

For more information, see the `Index` API documentation.

1.7 Graph

The walrus `graph` module provides a lightweight hexastore implementation. The *`Graph`* class uses Redis *`ZSet`* objects to store collections of `subject - predicate - object` triples. These relationships can then be queried in a very flexible manner.

Note: The hexastore logic is expecting UTF-8 encoded values. If you are using Python 2.X unicode text, you are responsible for encoding prior to storing/querying with those values.

For example, we might store things like:

- charlie – friends – huey
- charlie – lives – Kansas
- huey – lives – Kansas

We might wish to ask questions of our data-store like "which of charlie's friends live in Kansas?" To do this, we will store every permutation of the S-P-O triples, then we can efficiently query using the parts of the relationship we know beforehand.

- query the "object" portion of the "charlie – friends" subject/predicate.
- for each object returned, turn it into the subject of a second query whose predicate is "lives" and whose object is "Kansas".

So we would return the subjects that satisfy the following expression:

```
("charlie -- friends") -- lives -- Kansas
```

Let's go through this simple example to illustrate how the *`Graph`* class works.

```
from walrus import Database

# Begin by instantiating a `Graph` object.
db = Database()
graph = db.graph('people')

# Store my friends.
```

(continues on next page)

(continued from previous page)

```
# "charlie" is subject, "friends" is predicate, "huey" is object.
graph.store('charlie', 'friends', 'huey')

# Can also store multiple relationships at once.
graph.store_many((
    ('charlie', 'friends', 'zaizee'),
    ('charlie', 'friends', 'nuggie')))

# Store where people live.
graph.store_many((
    ('huey', 'lives', 'Kansas'),
    ('zaizee', 'lives', 'Missouri'),
    ('nuggie', 'lives', 'Kansas'),
    ('mickey', 'lives', 'Kansas')))

# We are now ready to search. We'll use a variable (X) to indicate
# the value we're interested in.
X = graph.v.X  # Create a variable placeholder.

# In the first clause we indicate we are searching for my friends.
# In the second clause, we only want those friends who also live
# in Kansas.
results = graph.search(
    {'s': 'charlie', 'p': 'friends', 'o': X},
    {'s': X, 'p': 'lives', 'o': 'Kansas'})

print(results)

# Prints: {'X': {'huey', 'nuggie'}}
```

In the above example, the result value is a dictionary of variable values that satisfy the search expressions. The *search()* method is quite powerful!

1.7.1 An even simpler example

Let's say we wish only to retrieve a list of Charlie's friends. In this case we do not need to use a variable. We can use the simpler *query()* method. This method optionally takes a subject, predicate and/or object and, using the provided data, returns all objects that "match" the given pieces.

So to find Charlie's friends, we would write:

```
query = graph.query(s='charlie', p='friends')
for result in query:
    print(result['o'])  # Print the object for the corresponding S/P.
```

1.8 Rate Limit

Walrus provides a simple *RateLimit* implementation that makes use of Redis' *List* object to store a series of event timestamps.

As the rate-limiter logs events, it maintains a fixed-size list of timestamps. When the list of timestamps is at max capacity, Walrus will look at the difference between the oldest timestamp and the present time to determine if a new event can be logged.

Example with a rate limiter that allows 2 events every 10 seconds.

- Log event from IP 192.168.1.2
- List for key `192.168.1.2` now contains `['14:42:27.04521']` (these are actually unix timestamps, but are shown as times for readability).
- Five seconds later log another event from the same IP.
- List for `192.168.1.2` now contains `['14:42:32.08293', '14:42:27.04521']`
- Two seconds later attempt another event from the same IP. Since the list is "at capacity", and the time difference between the oldest event and the newest is less than 10 seconds, the event will not be logged and the event will be rate-limited.

1.8.1 Basic usage

You can *limit()* to log an event and check whether it should be rate-limited:

```
>>> from walrus import *
>>> db = Database()
>>> rate_limit = db.rate_limit('mylimit', limit=2, per=60)  # 2 events per minute.

>>> rate_limit.limit('user-1')
False
>>> rate_limit.limit('user-1')
False
>>> rate_limit.limit('user-1')  # Slow down, user-1!
True

>>> rate_limit.limit('user-2')  # User 2 has not performed any events yet.
False
```

1.8.2 Decorator

The *RateLimit.rate_limited()* decorator can be used to restrict calls to a function or method. The decorator accepts a `key_function` parameter which instructs it how to uniquely identify the source of the function call. For example, on a web-site, you might want the key function to be derived from the requesting user's IP address.

```
rate_limit = walrus.rate_limit('login-limiter', limit=3, per=60)

@app.route('/login/', methods=['GET', 'POST'])
@rate_limit.rate_limited(lambda: request.remote_addr)
def login():
    # Accept user login, etc.
    pass
```

Note: The *rate_limited()* decorator will raise a `RateLimitException` when an attempt to call the decorated function would exceed the allowed number of events. In your application you can catch these and perform the appropriate action.

If no key function is supplied, then Walrus will simply take the hash of all the arguments the function was called with and treat that as the key. Except for very simple functions, this is probably not what you want, so take care to ensure your `key_function` works as you expect.

1.9 Streams

Redis streams is a new data-type available in Redis 5.0 which provides a persistent, append-only log. Redis streams are a complex topic, so I strongly recommend reading the streams introduction.

I like to think of streams as having two modes of operation:

- standalone-mode: streams act much like every other data-structure
- consumer-groups: streams become stateful, with state such as "which messages were read?", "who read what?", etc are tracked within Redis.

Stream objects in walrus can be used standalone or within the context of a *ConsumerGroup*.

1.9.1 Standalone streams

In standalone mode, streams behave much like every other data-structure in Redis. By this, I mean that they act as a dumb container: you append items, you read them, you delete them – everything happens explicitly. Streams support the following operations:

- Add a new item (XADD) - *Stream.add()*
- Read a range of items (XRANGE) - *Stream.range()*
- Read new messages, optionally blocking (XREAD) - *Stream.read()*
- Delete one or more items (XDEL) - *Stream.delete()*
- Get the length of the stream (XLEN) - `Stream.length()`
- Trim the length to a given size (XTRIM) - *Stream.trim()*
- Set the maximum allowable ID (XSETID) - *Stream.set_id()*

To get started with streams, we'll create a *Database* instance and use it to instantiate a *Stream*:

```
from walrus import Database  # A subclass of the redis-py Redis client.

db = Database()
stream = db.Stream('stream-a')  # Create a new stream instance.
```

When adding data to a stream, Redis can automatically provide you with a unique timestamp-based identifier, which is almost always what you want. When a new message is added, the message id is returned:

```
msgid = stream.add({'message': 'hello streams'})
print(msgid)

# Prints something like:
# b'1539008591844-0'
```

Message ids generated by Redis consist of a millisecond timestamp along with a sequence number (for ordering messages that arrived on the same millisecond). Let's *add()* a couple more items:

```
msgid2 = stream.add({'message': 'message 2'})
msgid3 = stream.add({'message': 'message 3'})
```

Ranges of records can be read using either the *range()* method, or using Python's slice notation. The message ids provided as the range endpoints are inclusive when using the range API:

```
# Get messages 2 and newer:
messages = stream[msgid2:]

# messages contains:
[(b'1539008914283-0', {b'message': b'message 2'}),
 (b'1539008918230-0', {b'message': b'message 3'})]

# We can use the "step" parameter to limit the number of records returned.
messages = stream[msgid::2]

# messages contains the first two messages:
[(b'1539008903588-0', {b'message': b'hello, stream'}),
 (b'1539008914283-0', {b'message': b'message 2'})]

# Get all messages in stream:
messages = list(stream)
[(b'1539008903588-0', {b'message': b'hello, stream'}),
 (b'1539008914283-0', {b'message': b'message 2'}),
 (b'1539008918230-0', {b'message': b'message 3'})]
```

The size of streams can be managed by deleting messages by id, or by "trimming" the stream, which removes the oldest messages. The desired size is specified when issuing a *trim()* operation, though, due to the internal implementation of the stream data-structures, the size is considered approximate by default.

```
# Adding and deleting a message:
msgid4 = stream.add({'message': 'delete me'})
del stream[msgid4]

# How many items are in the stream?
print(len(stream))  # Prints 3.
```

To see how trimming works, let's create another stream and fill it with 1000 items, then request it to be trimmed to 10 items:

```
# Add 1000 items to "stream-2".
stream2 = db.Stream('stream-2')
for i in range(1000):
    stream2.add({'data': 'message-%s' % i})

# Trim stream-2 to (approximately) 10 most-recent messages.
nremoved = stream2.trim(10)
print(nremoved)
# 909
print(len(stream2))
# 91

# To trim to an exact number, specify `approximate=False`:
stream2.trim(10, approximate=False)  # Returns 81.
print(len(stream2))
# 10
```

The previous examples show how to *add()*, read a *range()* of messages, *delete()* messages, and manage the size using the *trim()* method. When processing a continuous stream of events, though, it may be desirable to **block** until messages are added. For this we can use the *read()* API, which supports blocking until messages become available.

```
# By default, calling `stream.read()` returns all messages in the stream:
stream.read()

# Returns:
[(b'1539008903588-0', {b'message': b'hello, stream'}),
 (b'1539008914283-0', {b'message': b'message 2'}),
 (b'1539008918230-0', {b'message': b'message 3'})]
```

We can pass a message id to *read()*, and unlike the slicing operations, this id is considered the "last-read message" and acts as an **exclusive** lower-bound:

```
# Read any messages newer than msgid2.
stream.read(last_id=msgid2)

# Returns:
[(b'1539008918230-0', {b'message': b'message 3'})]

# This returns None since there are no messages newer than msgid3.
stream.read(last_id=msgid3)
```

We can make *read()* blocking by specifying a special id, `"$"`, and a `block` in milliseconds. To block forever, you can use `block=0`.

```
# This will block for 2 seconds, after which an empty list is returned
# (provided no messages are added while waiting).
stream.read(block=2000, last_id='$')
```

While its possible to build consumers using these APIs, the client is still responsible for keeping track of the last-read message ID and coming up with semantics for retrying failed messages, etc. In the next section, we'll see how consumer groups can greatly simplify building a stream processing pipeline.

1.9.2 Consumer groups

In consumer-group mode, streams retain the behaviors of standalone mode, adding functionality which makes them *stateful*. What state is tracked?

- Read any unseen messages (`XREAD`) - `ConsumerGroupStream.read()`
- List messages that were read, but not acknowledged (`XPENDING`) - `ConsumerGroupStream.pending()`
- Acknowledge one or more pending messages (`XACK`) - `ConsumerGroupStream.ack()`
- Claim one or more pending messages for re-processing (`XCLAIM`) - `ConsumerGroupStream.claim()`

ConsumerGroup objects provide the building-blocks for robust message processing pipelines or task queues. Ordinarily this type of stuff would be implemented by the client – having it in Redis means that we have a single, unified interface (rather than implementation-specific, with all the bugs that likely entails). Furthermore, consumer group state is tracked by the RDB and replicated.

```
# Consumer groups require that a stream exist before the group can be
# created, so we have to add an empty message.
stream_keys = ['stream-a', 'stream-b', 'stream-c']
for stream in stream_keys:
    db.xadd(stream, {'data': ''})

# Create a consumer-group for streams a, b, and c. We will mark all
# messages as having been processed, so only messages added after the
```

(continues on next page)

(continued from previous page)

```
# creation of the consumer-group will be read.
cg = db.consumer_group('cg-abc', stream_keys)
cg.create()  # Create the consumer group.
cg.set_id('$')
```

To read from all the streams in a consumer group, we can use the `ConsumerGroupStream.read()` method. Since we marked all messages as read and have not added anything new since creating the consumer group, the return value is an empty list:

```
resp = cg.read()

# Returns an empty list:
[]
```

For convenience, walrus exposes the individual streams within a consumer group as attributes on the *ConsumerGroup* instance. Let's add some messages to streams *a*, *b*, and *c*:

```
cg.stream_a.add({'message': 'new a'})
cg.stream_b.add({'message': 'new for b'})
for i in range(10):
    cg.stream_c.add({'message': 'c-%s' % i})
```

Now let's try reading from the consumer group again. We'll pass `count=1` so that we read no more than one message from each stream in the group:

```
# Read up to one message from each stream in the group.
cg.read(count=1)

# Returns:
[('stream-a', [(b'1539023088125-0', {b'message': b'new a'})]),
 ('stream-b', [(b'1539023088125-0', {b'message': b'new for b'})]),
 ('stream-c', [(b'1539023088126-0', {b'message': b'c-0'})])]
```

We've now read all the unread messages from streams *a* and *b*, but stream *c* still has messages. Calling `read()` again will give us the next unread message from stream *c*:

```
# Read up to 1 message from each stream in the group. Since
# we already read everything in streams a and b, we will only
# get the next unread message in stream c.
cg.read(count=1)

# Returns:
[('stream-c', [(b'1539023088126-1', {b'message': b'c-1'})])]
```

When using consumer groups, messages that are read need to be **acknowledged**. Let's look at the **pending** (read but unacknowledged) messages from stream *a* using the `pending()` method, which returns a list of metadata about each unacknowledged message:

```
# We read one message from stream a, so we should see one pending message.
cg.stream_a.pending()

# Returns a list of:
# [message id, consumer name, message age, delivery count]
[[b'1539023088125-0', b'cg-abc.c1', 22238, 1]]
```

To acknowledge receipt of a message and remove it from the pending list, use the `ack()` method on the consumer group stream:

```
# View the pending message list for stream a.
pending_list = cg.stream_a.pending()
msg_id = pending_list[0]['message_id']

# Acknowledge the message.
cg.stream_a.ack(msg_id)

# Returns number of pending messages successfully acknowledged:
1
```

Consumer groups have the concept of individual **consumers**. These might be workers in a process pool, for example. Note that the `pending()` method returned the consumer name as `"cg-abc.c1"`. Walrus uses the consumer group name + `".c1"` as the name for the default consumer name. To create another consumer within a given group, we can use the `consumer()` method:

```
# Create a second consumer within the consumer group.
cg2 = cg.consumer('cg-abc.c2')
```

Creating a new consumer within a consumer group does not affect the state of the group itself. Calling `read()` using our new consumer will pick up from the last-read message, as you would expect:

```
# Read from our consumer group using the new consumer. Recall
# that we read all the messages from streams a and b, and the
# first two messages in stream c.
cg2.read(count=1)

# Returns:
[('stream-c', [(b'1539023088126-2', {b'message': b'c-2'})])]
```

If we look at the pending message status for stream *c*, we will see that the first and second messages were read by the consumer *"cg-abc.c1"* and the third message was read by our new consumer, *"cg-abc.c2"*:

```
# What messages have been read, but were not acknowledged, from stream c?
cg.stream_c.pending()

# Returns list of [message id, consumer, message age, delivery count]:
[{'message_id': b'1539023088126-0', 'consumer': b'cg-abc.c1',
  'time_since_delivered': 51329, 'times_delivered': 1}],
 {'message_id': b'1539023088126-1', 'consumer': b'cg-abc.c1',
  'time_since_delivered': 43772, 'times_delivered': 1},
 {'message_id': b'1539023088126-2', 'consumer': b'cg-abc.c2',
  'time_since_delivered': 5966, 'times_delivered': 1}]
```

Consumers can `claim()` pending messages, which transfers ownership of the message and returns a list of (message id, data) tuples to the caller:

```
# Unpack the pending messages into a couple variables.
mc1, mc2, mc3 = cg.stream_c.pending()

# Claim the first message for consumer 2:
cg2.stream_c.claim(mc1['message_id'])

# Returns a list of (message id, data) tuples for the claimed messages:
[(b'1539023088126-0', {b'message': b'c-0'})]
```

Re-inspecting the pending messages for stream *c*, we can see that the consumer for the first message has changed and the message age has been reset:

```
# What messages are pending in stream c?
cg.stream_c.pending()

# Returns:
[{'message_id': b'1539023088126-0', 'consumer': b'cg-abc.c2',
  'time_since_delivered': 2168, 'times_delivered': 1},
 {'message_id': b'1539023088126-1', 'consumer': b'cg-abc.c1',
  'time_since_delivered': 47141, 'times_delivered': 1},
 {'message_id': b'1539023088126-2', 'consumer': b'cg-abc.c2',
  'time_since_delivered': 9335, 'times_delivered': 1}]
```

The individual streams within the consumer group support a number of useful APIs:

- `consumer_group.stream.ack(*id_list)` - acknowledge one or more messages read from the given stream.
- `consumer_group.stream.add(data, id='*', maxlen=None, approximate=True)` - add a new message to the stream. The `maxlen` parameter can be used to keep the stream from growing without bounds. If given, the `approximate` flag indicates whether the stream maxlen should be approximate or exact.
- `consumer_group.stream.claim(*id_list)` - claim one or more pending messages.
- `consumer_group.stream.delete(*id_list)` - delete one or more messages by ID.
- `consumer_group.stream.pending(start='-', stop='+', count=1000)` - get the list of unacknowledged messages in the stream. The `start` and `stop` parameters can be message ids, while the `count` parameter can be used to limit the number of results returned.
- `consumer_group.stream.read(count=None, block=None)` - monitor the stream for new messages within the context of the consumer group. This method can be made to block by specifying a `block` (or `0` to block forever).
- `consumer_group.stream.set_id(id='$')` - set the id of the last-read message for the consumer group. Use the special id `"$"` to indicate all messages have been read, or `"0-0"` to mark all messages as unread.
- `consumer_group.stream.trim(count, approximate=True)` - trim the stream to the given size.

1.9.3 TimeSeries

Redis automatically uses the millisecond timestamp plus a sequence number to uniquely identify messages added to a stream. This makes streams a natural fit for time-series data. To simplify working with streams as time-series in Python, you can use the special *TimeSeries* helper class, which acts just like the *ConsumerGroup* from the previous section with the exception that it can translate between Python `datetime` objects and message ids automatically.

To get started, we'll create a *TimeSeries* instance, specifying the stream keys, just like we did with *ConsumerGroup*:

```
# Create a time-series consumer group named "demo-ts" for the
# streams s1 and s2.
ts = db.time_series('demo-ts', ['s1', 's2'])

# Add dummy data and create the consumer group.
db.xadd('s1', {'': ''}, id='0-1')
db.xadd('s2', {'': ''}, id='0-1')
ts.create()
ts.set_id('$')  # Do not read the dummy items.
```

Let's add some messages to the time-series, one for each day between January 1st and 10th, 2018:

```
from datetime import datetime, timedelta

date = datetime(2018, 1, 1)
for i in range(10):
    ts.s1.add({'message': 's1-%s' % date}, id=date)
    date += timedelta(days=1)
```

We can read messages from the stream using the familiar slicing API. For example, to read 3 messages starting at January 2nd, 2018:

```
ts.s1[datetime(2018, 1, 2)::3]

# Returns messages for Jan 2nd - 4th:
[<Message s1 1514872800000-0: {'message': 's1-2018-01-02 00:00:00'}>,
 <Message s1 1514959200000-0: {'message': 's1-2018-01-03 00:00:00'}>,
 <Message s1 1515045600000-0: {'message': 's1-2018-01-04 00:00:00'}>]
```

Note that the values returned are `Message` objects. Message objects provide some convenience functions, such as extracting timestamp and sequence values from stream message ids:

```
for message in ts.s1[datetime(2018, 1, 1)::3]:
    print(message.stream, message.timestamp, message.sequence, message.data)

# Prints:
s1 2018-01-01 00:00:00 0 {'message': 's1-2018-01-01 00:00:00'}
s1 2018-01-02 00:00:00 0 {'message': 's1-2018-01-02 00:00:00'}
s1 2018-01-03 00:00:00 0 {'message': 's1-2018-01-03 00:00:00'}
```

Let's add some messages to stream "s2" as well:

```
date = datetime(2018, 1, 1)
for i in range(5):
    ts.s2.add({'message': 's2-%s' % date}, id=date)
    date += timedelta(days=1)
```

One difference between *TimeSeries* and *ConsumerGroup* is what happens when reading from multiple streams. ConsumerGroup returns a dictionary keyed by stream, along with a corresponding list of messages read from each stream. TimeSeries, however, returns a flat list of Message objects:

```
# Read up to 2 messages from each stream (s1 and s2):
messages = ts.read(count=2)

# "messages" is a list of messages from both streams:
[<Message s1 1514786400000-0: {'message': 's1-2018-01-01 00:00:00'}>,
 <Message s2 1514786400000-0: {'message': 's2-2018-01-01 00:00:00'}>,
 <Message s1 1514872800000-0: {'message': 's1-2018-01-02 00:00:00'}>,
 <Message s2 1514872800000-0: {'message': 's2-2018-01-02 00:00:00'}>]
```

When inspecting pending messages within a *TimeSeries* the message ids are unpacked into (datetime, seq) 2-tuples:

```
ts.s1.pending()

# Returns:
[((datetime.datetime(2018, 1, 1, 0, 0), 0), 'events-ts.c', 1578, 1),
 ((datetime.datetime(2018, 1, 2, 0, 0), 0), 'events-ts.c', 1578, 1)]
```

(continues on next page)

(continued from previous page)

```
# Acknowledge the pending messages:
for msgts_seq, _, _, _ in ts.s1.pending():
    ts.s1.ack(msgts_seq)
```

We can set the last-read message id using a datetime:

```
ts.s1.set_id(datetime(2018, 1, 1))

# Next read will be 2018-01-02, ...
ts.s1.read(count=2)

# Returns:
[<Message s1 1514872800000-0: {'message': 's1-2018-01-02 00:00:00'}>,
 <Message s1 1514959200000-0: {'message': 's1-2018-01-03 00:00:00'}>]
```

As with *ConsumerGroup*, the *TimeSeries* helper provides stream-specific APIs for claiming unacknowledged messages, creating additional consumers, etc.

1.9.4 Learning more

For more information, the following links may be helpful:

- Redis streams introduction.
- Example multi-process task queue using walrus and streams.
- API docs for *Stream*, *ConsumerGroup*, ConsumerGroupStream and *TimeSeries*.

1.10 Models

Warning: Walrus models should **not** be considered production-grade code and I strongly advise against anyone actually using it for anything other than experimenting or for inspiration/learning.

My advice: just use hashes for your structured data. If you need ad-hoc queries, then use a relational database.

Walrus provides a lightweight *Model* class for storing structured data and executing queries using secondary indexes.

```
>>> from walrus import *
>>> db = Database()
```

Let's create a simple data model to store some users.

```
>>> class User(Model):
...     __database__ = db
...     name = TextField(primary_key=True)
...     dob = DateField(index=True)
```

Note: As of 0.4.0, the Model.database attribute has been renamed to Model.__database__. Similarly, Model.namespace is now Model.__namespace__.

1.10.1 Creating, Updating and Deleting

To add objects to a collection, you can use *Model.create()*:

```
>>> User.create(name='Charlie', dob=datetime.date(1983, 1, 1))
<User: Charlie>

>>> names_dobs = [
...     ('Huey', datetime.date(2011, 6, 1)),
...     ('Zaizee', datetime.date(2012, 5, 1)),
...     ('Mickey', datetime.date(2007, 8, 1)),

>>> for name, dob in names_dobs:
...     User.create(name=name, dob=dob)
```

We can retrieve objects by primary key (name in this case). Objects can be modified or deleted after they have been created.

```
>>> zaizee = User.load('Zaizee')  # Get object by primary key.
>>> zaizee.name
'Zaizee'
>>> zaizee.dob
datetime.date(2012, 5, 1)

>>> zaizee.dob = datetime.date(2012, 4, 1)
>>> zaizee.save()

>>> nobody = User.create(name='nobody', dob=datetime.date(1990, 1, 1))
>>> nobody.delete()
```

1.10.2 Retrieving all records in a collection

We can retrieve all objects in the collection by calling *Model.all()*, which returns an iterator that successively yields model instances:

```
>>> for user in User.all():
...     print user.name
Huey
Zaizee
Charlie
Mickey
```

Note: The objects from *all()* are returned in an undefined order. This is because the index containing all primary keys is implemented as an unordered *Set*.

1.10.3 Sorting records

To get the objects in order, we can use *Model.query()*:

```
>>> for user in User.query(order_by=User.name):
...     print user.name
Charlie
```

(continues on next page)

(continued from previous page)

```
Huey
Mickey
Zaizee

>>> for user in User.query(order_by=User.dob.desc()):
...     print user.dob
2012-04-01
2011-06-01
2007-08-01
1983-01-01
```

1.10.4 Filtering records

Walrus supports basic filtering. The filtering options available vary by field type, so that *TextField*, *UUIDField* and similar non-scalar types support only equality and inequality tests. Scalar values, on the other hand, like integers, floats or dates, support range operations.

> **Warning:** You must specify `index=True` to be able to use a field for filtering.

Let's see how this works by filtering on name and dob. The *query()* method returns zero or more objects, while the *get()* method requires that there be exactly one result:

```
>>> for user in User.query(User.dob <= datetime.date(2009, 1, 1)):
...     print user.dob
2007-08-01
1983-01-01

>>> charlie = User.get(User.name == 'Charlie')
>>> User.get(User.name = 'missing')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/charles/pypath/walrus.py", line 1662, in get
    raise ValueError('Got %s results, expected 1.' % len(result))
ValueError: Got 0 results, expected 1.
```

We can combine multiple filters using bitwise *and* and *or*:

```
>>> low = datetime.date(2006, 1, 1)
>>> high = datetime.date(2012, 1, 1)
>>> query = User.query(
...     (User.dob >= low) &
...     (User.dob <= high))

>>> for user in query:
...     print user.dob

2011-06-01
2007-08-01

>>> query = User.query(User.dob.between(low, high))  # Equivalent to above.
>>> for user in query:
...     print user.dob
```

(continues on next page)

(continued from previous page)

```
2011-06-01
2007-08-01

>>> query = User.query(
...     (User.dob <= low) |
...     (User.dob >= high))

>>> for user in query:
...     print user.dob
2012-04-01
1983-01-01
```

You can combine filters with ordering:

```
>>> expr = (User.name == 'Charlie') | (User.name == 'Zaizee')
>>> for user in User.query(expr, order_by=User.name):
...     print user.name
Charlie
Zaizee

>>> for user in User.query(User.name != 'Charlie', order_by=User.name.desc()):
...     print user.name
Zaizee
Mickey
Huey
```

1.10.5 Container Fields

Up until now the fields we've used have been simple key/value pairs that are stored directly in the hash of model data. In this section we'll look at a group of special fields that correspond to Redis container types.

Let's create a model for storing personal notes. The notes will have a text field for the content and a timestamp, and as an interesting flourish we'll add a *SetField* to store a collection of tags.

```
class Note(Model):
    __database__ = db
    text = TextField()
    timestamp = DateTimeField(
        default=datetime.datetime.now,
        index=True)
    tags = SetField()
```

Note: Container fields cannot be used as a secondary index, nor can they be used as the primary key for a model. Finally, they do not accept a default value.

Warning: Due to the implementation, it is necessary that the model instance have a primary key value before you can access the container field. This is because the key identifying the container field needs to be associated with the instance, and the way we do that is with the primary key.

Here is how we might use the new note model:

```
>>> note = Note.create(content='my first note')
>>> note.tags
<Set "note:container.tags.note:id.3": 0 items>
>>> note.tags.add('testing', 'walrus')

>>> Note.load(note._id).tags
<Set "note:container.tags.note:id.3": 0 items>
```

In addition to *SetField*, there is also *HashField*, *ListField*, *ZSetField*.

1.10.6 Full-text search

I've added a really (really) simple full-text search index type. Here is how to use it:

```
>>> class Note(Model):
...     __database__ = db
...     content = TextField(fts=True)  # Note the "fts=True".
```

When a field contains an full-text index, then the index will be populated when new objects are added to the database:

```
>>> Note.create(content='this is a test of walrus FTS.')
>>> Note.create(content='favorite food is walrus-mix.')
>>> Note.create(content='do not forget to take the walrus for a walk.')
```

Use *TextField.search()* to create a search expression, which is then passed to the *Model.query()* method:

```
>>> for note in Note.query(Note.content.search('walrus')):
...     print note.content
do not forget to take the walrus for a walk.
this is a test of walrus FTS.
favorite food is walrus-mix.

>>> for note in Note.query(Note.content.search('walk walrus')):
...     print note.content
do not forget to take the walrus for a walk.

>>> for note in Note.query(Note.content.search('walrus mix')):
...     print note.content
favorite food is walrus-mix.
```

We can also specify complex queries using `AND` and `OR` conjunctions:

```
>>> for note in Note.query(Note.content.search('walrus AND (mix OR fts)')):
...     print note.content
this is a test of walrus FTS.
favorite food is walrus-mix.

>>> query = '(test OR food OR walk) AND walrus AND (favorite OR forget)'
>>> for note in Note.query(Note.content.search(query)):
...     print note.content
do not forget to take the walrus for a walk.
favorite food is walrus-mix.
```

Features

- Automatic removal of stop-words
- Porter stemmer on by default
- Optional double-metaphone implementation
- Default conjunction is *AND*, but there is also support for *OR*.

Limitations

- Partial strings are not matched.
- Very naive scoring function.
- Quoted multi-word matches do not work.

1.10.7 Need more power?

walrus' querying capabilities are extremely basic. If you want more sophisticated querying, check out StdNet. StdNet makes extensive use of Lua scripts to provide some really neat querying/filtering options.

1.11 API Documentation

class walrus.**Database**(*Redis*)
Redis-py client with some extras.

Array(*key*)
Create a *Array* instance wrapping the given key.

Hash(*key*)
Create a *Hash* instance wrapping the given key.

HyperLogLog(*key*)
Create a *HyperLogLog* instance wrapping the given key.

Index(*name*, ***options*)
Create a *Index* full-text search index with the given name and options.

List(*key*)
Create a *List* instance wrapping the given key.

Set(*key*)
Create a *Set* instance wrapping the given key.

Stream(*key*)
Create a *Stream* instance wrapping the given key.

ZSet(*key*)
Create a *ZSet* instance wrapping the given key.

__init__(**args*, ***kwargs*)

Parameters

- **args** – Arbitrary positional arguments to pass to the base Redis instance.
- **kwargs** – Arbitrary keyword arguments to pass to the base Redis instance.

- **script_dir** (*str*) – Path to directory containing walrus scripts. Use "script_dir=False" to disable loading any scripts.

__iter__()
Iterate over the keys of the selected database.

bit_field(*key*)
Container for working with the Redis BITFIELD command.

Returns a *BitField* instance.

bloom_filter(*key*, *size=65536*)
Create a *BloomFilter* container type.

Bloom-filters are probabilistic data-structures that are used to answer the question: "is X a member of set S?" It is possible to receive a false positive, but impossible to receive a false negative (in other words, if the bloom filter contains a value, it will never erroneously report that it does *not* contain such a value). The accuracy of the bloom-filter and the likelihood of a false positive can be reduced by increasing the size of the bloomfilter. The default size is 64KB (or 524,288 bits).

cache(*name='cache'*, *default_timeout=3600*)
Create a *Cache* instance.

Parameters

- **name** (*str*) – The name used to prefix keys used to store cached data.
- **default_timeout** (*int*) – The default key expiry.

Returns A *Cache* instance.

cas(*key*, *value*, *new_value*)
Perform an atomic compare-and-set on the value in "key", using a prefix match on the provided value.

consumer_group(*group*, *keys*, *consumer=None*)
Create a named *ConsumerGroup* instance for the given key(s).

Parameters

- **group** – name of consumer group
- **keys** – stream identifier(s) to monitor. May be a single stream key, a list of stream keys, or a key-to-minimum id mapping. The minimum id for each stream should be considered an exclusive lower-bound. The '$' value can also be used to only read values added *after* our command started blocking.
- **consumer** – name for consumer within group

Returns a *ConsumerGroup* instance

counter(*name*)
Create a *Counter* instance.

Parameters **name** (*str*) – The name used to store the counter's value.

Returns A *Counter* instance.

get_key(*key*)
Return a rich object for the given key. For instance, if a hash key is requested, then a *Hash* will be returned.

Note: only works for Hash, List, Set and ZSet.

Parameters **key** (*str*) – Key to retrieve.

Returns A hash, set, list, zset or array.

get_temp_key()
Generate a temporary random key using UUID4.

graph(*name*, **args*, ***kwargs*)
Creates a *Graph* instance.

Parameters **name** (*str*) – The namespace for the graph metadata.

Returns a *Graph* instance.

listener(*channels=None*, *patterns=None*, *is_async=False*)
Decorator for wrapping functions used to listen for Redis pub-sub messages.

The listener will listen until the decorated function raises a `StopIteration` exception.

Parameters

- **channels** (*list*) – Channels to listen on.
- **patterns** (*list*) – Patterns to match.
- **is_async** (*bool*) – Whether to start the listener in a separate thread.

lock(*name*, *ttl=None*, *lock_id=None*)
Create a named *Lock* instance. The lock implements an API similar to the standard library's `threading.Lock`, and can also be used as a context manager or decorator.

Parameters

- **name** (*str*) – The name of the lock.
- **ttl** (*int*) – The time-to-live for the lock in milliseconds (optional). If the ttl is `None` then the lock will not expire.
- **lock_id** (*str*) – Optional identifier for the lock instance.

rate_limit(*name*, *limit=5*, *per=60*, *debug=False*)
Rate limit implementation. Allows up to *limit* of events every *per* seconds.

See rate-limit for more information.

run_script(*script_name*, *keys=None*, *args=None*)
Execute a walrus script with the given arguments.

Parameters

- **script_name** – The base name of the script to execute.
- **keys** (*list*) – Keys referenced by the script.
- **args** (*list*) – Arguments passed in to the script.

Returns Return value of script.

Note: Redis scripts require two parameters, `keys` and `args`, which are referenced in lua as `KEYS` and `ARGV`.

---

search(*pattern*)
Search the keyspace of the selected database using the given search pattern.

Parameters **pattern** (*str*) – Search pattern using wildcards.

Returns Iterator that yields matching keys.

stream_log(*callback*, *connection_id='monitor'*)
Stream Redis activity one line at a time to the given callback.

Parameters **callback** – A function that accepts a single argument, the Redis command.

time_series(*group*, *keys*, *consumer=None*)
Create a named *TimeSeries* consumer-group for the given key(s). TimeSeries objects are almost identical to *ConsumerGroup* except they offer a higher level of abstraction and read/write message ids as datetimes.

Parameters

- **group** – name of consumer group
- **keys** – stream identifier(s) to monitor. May be a single stream key, a list of stream keys, or a key-to-minimum id mapping. The minimum id for each stream should be considered an exclusive lower-bound. The '$' value can also be used to only read values added *after* our command started blocking.
- **consumer** – name for consumer within group

Returns a *TimeSeries* instance

xsetid(*name*, *id*)
Set the last ID of the given stream.

Parameters

- **name** – stream identifier
- **id** – new value for last ID

1.11.1 Container types

class walrus.**Container**(*database*, *key*)
Base-class for rich Redis object wrappers.

clear()
Clear the contents of the container by deleting the key.

dump()
Dump the contents of the given key using Redis' native serialization format.

expire(*ttl=None*)
Expire the given key in the given number of seconds. If `ttl` is `None`, then any expiry will be cleared and key will be persisted.

pexpire(*ttl=None*)
Expire the given key in the given number of milliseconds. If `ttl` is `None`, then any expiry will be cleared and key will be persisted.

class walrus.**Hash**(*Container*)
Redis Hash object wrapper. Supports a dictionary-like interface with some modifications.

See Hash commands for more info.

__contains__(*key*)
Return a boolean valud indicating whether the given key exists.

__delitem__(*key*)
Delete the key from the hash.

__getitem__(*item*)
Retrieve the value at the given key. To retrieve multiple values at once, you can specify multiple keys as a tuple or list:

```
hsh = db.Hash('my-hash')
first, last = hsh['first_name', 'last_name']
```

__iter__()
Iterate over the items in the hash.

__len__()
Return the number of keys in the hash.

__setitem__(*key*, *value*)
Set the value of the given key.

as_dict(*decode=False*)
Return a dictionary containing all the key/value pairs in the hash.

incr(*key*, *incr_by=1*)
Increment the key by the given amount.

items(*lazy=False*)
Like Python's `dict.items()` but supports an optional parameter `lazy` which will return a generator rather than a list.

keys()
Return the keys of the hash.

search(*pattern*, *count=None*)
Search the keys of the given hash using the specified pattern.

Parameters

- **pattern** (*str*) – Pattern used to match keys.
- **count** (*int*) – Limit number of results returned.

Returns An iterator yielding matching key/value pairs.

update(*_Hash__data=None*, ***kwargs*)
Update the hash using the given dictionary or key/value pairs.

values()
Return the values stored in the hash.

class walrus.**List**(*Container*)
Redis List object wrapper. Supports a list-like interface.

See List commands for more info.

__delitem__(*item*)
By default Redis treats deletes as delete by value, as opposed to delete by index. If an integer is passed into the function, it will be treated as an index, otherwise it will be treated as a value.

If a slice is passed, then the list will be trimmed so that it *ONLY* contains the range specified by the slice start and stop. Note that this differs from the default behavior of Python's *list* type.

__getitem__(*item*)
Retrieve an item from the list by index. In addition to integer indexes, you can also pass a `slice`.

__iter__()
Iterate over the items in the list.

__len__()
Return the length of the list.

__setitem__(*idx*, *value*)
Set the value of the given index.

append(*value*)
Add the given value to the end of the list.

as_list(*decode=False*)
Return a list containing all the items in the list.

extend(*value*)
Extend the list by the given value.

insert_after(*value*, *key*)
Insert the given value into the list after the index containing `key`.

insert_before(*value*, *key*)
Insert the given value into the list before the index containing `key`.

popleft()
Remove the first item from the list.

popright()
Remove the last item from the list.

prepend(*value*)
Add the given value to the beginning of the list.

class walrus.**Set**(*Container*)
Redis Set object wrapper. Supports a set-like interface.

See Set commands for more info.

__and__(*other*)
Return the set intersection of the current set and the left- hand *Set* object.

__contains__(*item*)
Return a boolean value indicating whether the given item is a member of the set.

__delitem__(*item*)
Remove the given item from the set.

__iter__()
Return an iterable that yields the items of the set.

__len__()
Return the number of items in the set.

__or__(*other*)
Return the set union of the current set and the left-hand *Set* object.

__sub__(*other*)
Return the set difference of the current set and the left- hand *Set* object.

add(**items*)
Add the given items to the set.

as_set(*decode=False*)
Return a Python set containing all the items in the collection.

diffstore(*dest*, **others*)
Store the set difference of the current set and one or more others in a new key.

Parameters

- **dest** – the name of the key to store set difference
- **others** – One or more *Set* instances

Returns A *Set* referencing `dest`.

interstore(*dest*, **others*)
Store the intersection of the current set and one or more others in a new key.

Parameters

- **dest** – the name of the key to store intersection
- **others** – One or more *Set* instances

Returns A *Set* referencing `dest`.

members()
Return a `set()` containing the members of the set.

pop()
Remove an element from the set.

random(*n=None*)
Return a random member of the given set.

remove(**items*)
Remove the given item(s) from the set.

search(*pattern*, *count=None*)
Search the values of the given set using the specified pattern.

Parameters

- **pattern** (*str*) – Pattern used to match keys.
- **count** (*int*) – Limit number of results returned.

Returns An iterator yielding matching values.

unionstore(*dest*, **others*)
Store the union of the current set and one or more others in a new key.

Parameters

- **dest** – the name of the key to store union
- **others** – One or more *Set* instances

Returns A *Set* referencing `dest`.

class walrus.**ZSet**(*Container*)
Redis ZSet object wrapper. Acts like a set and a dictionary.

See Sorted set commands for more info.

__contains__(*item*)
Return a boolean indicating whether the given item is in the sorted set.

__delitem__(*item*)
Delete the given item(s) from the set. Like *__getitem__()*, this method supports a wide variety of indexing and slicing options.

__getitem__(*item*)
Retrieve the given values from the sorted set. Accepts a variety of parameters for the input:

```
zs = db.ZSet('my-zset')

# Return the first 10 elements with their scores.
zs[:10, True]

# Return the first 10 elements without scores.
zs[:10]
zs[:10, False]

# Return the range of values between 'k1' and 'k10' along
# with their scores.
zs['k1':'k10', True]

# Return the range of items preceding and including 'k5'
# without scores.
zs[:'k5', False]
```

__iter__()
Return an iterator that will yield (item, score) tuples.

__len__()
Return the number of items in the sorted set.

__setitem__(*item*, *score*)
Add item to the set with the given score.

add(*_mapping=None*, ***kwargs*)
Add the given item/score pairs to the ZSet. Arguments are specified as a dictionary of item: score, or as keyword arguments.

as_items(*decode=False*)
Return a list of 2-tuples consisting of key/score.

bpopmax(*timeout=0*)
Atomically remove the highest-scoring item from the set, blocking until an item becomes available or timeout is reached (0 for no timeout, default).

Returns a 2-tuple of (item, score).

bpopmin(*timeout=0*)
Atomically remove the lowest-scoring item from the set, blocking until an item becomes available or timeout is reached (0 for no timeout, default).

Returns a 2-tuple of (item, score).

count(*low*, *high=None*)
Return the number of items between the given bounds.

incr(*key*, *incr_by=1.0*)
Increment the score of an item in the ZSet.

Parameters

- **key** – Item to increment.
- **incr_by** – Amount to increment item's score.

interstore(*dest*, **others*, ***kwargs*)
Store the intersection of the current zset and one or more others in a new key.

Parameters

- **dest** – the name of the key to store intersection

- **others** – One or more *ZSet* instances

Returns A *ZSet* referencing `dest`.

lex_count(*low*, *high*)
Count the number of members in a sorted set between a given lexicographical range.

popmax(*count=1*)
Atomically remove the highest-scoring item(s) in the set.

Returns a list of item, score tuples or `None` if the set is empty.

popmax_compat(*count=1*)
Atomically remove the highest-scoring item(s) in the set. Compatible with Redis versions < 5.0.

Returns a list of item, score tuples or `None` if the set is empty.

popmin(*count=1*)
Atomically remove the lowest-scoring item(s) in the set.

Returns a list of item, score tuples or `None` if the set is empty.

popmin_compat(*count=1*)
Atomically remove the lowest-scoring item(s) in the set. Compatible with Redis versions < 5.0.

Returns a list of item, score tuples or `None` if the set is empty.

range(*low*, *high*, *with_scores=False*, *desc=False*, *reverse=False*)
Return a range of items between `low` and `high`. By default scores will not be included, but this can be controlled via the `with_scores` parameter.

Parameters

- **low** – Lower bound.
- **high** – Upper bound.
- **with_scores** (*bool*) – Whether the range should include the scores along with the items.
- **desc** (*bool*) – Whether to sort the results descendingly.
- **reverse** (*bool*) – Whether to select the range in reverse.

range_by_lex(*low*, *high*, *start=None*, *num=None*, *reverse=False*)
Return a range of members in a sorted set, by lexicographical range.

rank(*item*, *reverse=False*)
Return the rank of the given item.

remove(**items*)
Remove the given items from the ZSet.

remove_by_rank(*low*, *high=None*)
Remove elements from the ZSet by their rank (relative position).

Parameters

- **low** – Lower bound.
- **high** – Upper bound.

remove_by_score(*low*, *high=None*)
Remove elements from the ZSet by their score.

Parameters

- **low** – Lower bound.
- **high** – Upper bound.

score(*item*)
Return the score of the given item.

search(*pattern*, *count=None*)
Search the set, returning items that match the given search pattern.

Parameters

- **pattern** (*str*) – Search pattern using wildcards.
- **count** (*int*) – Limit result set size.

Returns Iterator that yields matching item/score tuples.

unionstore(*dest*, **others*, ***kwargs*)
Store the union of the current set and one or more others in a new key.

Parameters

- **dest** – the name of the key to store union
- **others** – One or more *ZSet* instances

Returns A *ZSet* referencing `dest`.

class walrus.**HyperLogLog**(*Container*)
Redis HyperLogLog object wrapper.

See HyperLogLog commands for more info.

add(**items*)
Add the given items to the HyperLogLog.

merge(*dest*, **others*)
Merge one or more *HyperLogLog* instances.

Parameters

- **dest** – Key to store merged result.
- **others** – One or more `HyperLogLog` instances.

class walrus.**Array**(*Container*)
Custom container that emulates an array (as opposed to the linked-list implementation of *List*). This gives:

- O(1) append, get, len, pop last, set
- O(n) remove from middle

Array is built on top of the hash data type and is implemented using lua scripts.

__contains__(*item*)
Return a boolean indicating whether the given item is stored in the array. O(n).

__delitem__(*idx*)
Delete the given index.

__getitem__(*idx*)
Get the value stored in the given index.

__iter__()
Return an iterable that yields array items.

__len__()
Return the number of items in the array.

__setitem__(*idx*, *value*)
Set the value at the given index.

append(*value*)
Append a new value to the end of the array.

as_list(*decode=False*)
Return a list of items in the array.

extend(*values*)
Extend the array, appending the given values.

pop(*idx=None*)
Remove an item from the array. By default this will be the last item by index, but any index can be specified.

class walrus.**Stream**(*Container*)
Redis stream container.

__delitem__(*item*)
Delete one or more messages by id. The index can be either a single message id or a list/tuple of multiple ids.

__getitem__(*item*)
Read a range of values from a stream.

The index must be a message id or a slice. An empty slice will result in reading all values from the stream. Message ids provided as lower or upper bounds are inclusive.

To specify a maximum number of messages, use the "step" parameter of the slice.

__len__()
Return the length of a stream.

add(*data*, *id='*'*, *maxlen=None*, *approximate=True*)
Add data to a stream.

Parameters

- **data** (*dict*) – data to add to stream
- **id** – identifier for message ('*' to automatically append)
- **maxlen** – maximum length for stream
- **approximate** – allow stream max length to be approximate

Returns the added message id.

consumers_info(*group*)
Retrieve information about consumers within the given consumer group operating on the stream. Calls `xinfo_consumers()`.

Parameters **group** – consumer group name

Returns a dictionary containing consumer metadata

delete(**id_list*)
Delete one or more message by id. The index can be either a single message id or a list/tuple of multiple ids.

get(*docid*)
Get a message by id.

Parameters **docid** – the message id to retrieve.

Returns a 2-tuple of (message id, data) or None if not found.

groups_info()
Retrieve information about consumer groups for the stream. Wraps call to `xinfo_groups()`.

Returns a dictionary containing consumer group metadata

info()
Retrieve information about the stream. Wraps call to `xinfo_stream()`.

Returns a dictionary containing stream metadata

range(*start='-'*, *stop='+'*, *count=None*)
Read a range of values from a stream.

Parameters

- **start** – start key of range (inclusive) or '-' for oldest message
- **stop** – stop key of range (inclusive) or '+' for newest message
- **count** – limit number of messages returned

read(*count=None*, *block=None*, *last_id=None*)
Monitor stream for new data.

Parameters

- **count** (*int*) – limit number of messages returned
- **block** (*int*) – milliseconds to block, 0 for indefinitely
- **last_id** – Last id read (an exclusive lower-bound). If the '$' value is given, we will only read values added *after* our command started blocking.

Returns a list of (message id, data) 2-tuples.

set_id(*id*)
Set the maximum message id for the stream.

Parameters **id** – id of last-read message

trim(*count=None*, *approximate=True*, *minid=None*, *limit=None*)
Trim the stream to the given "count" of messages, discarding the oldest messages first.

Parameters

- **count** – maximum size of stream (maxlen)
- **approximate** – allow size to be approximate
- **minid** – evicts entries with IDs lower than the given min id.
- **limit** – maximum number of entries to evict.

class walrus.**ConsumerGroup**(*database*, *name*, *keys*, *consumer=None*)
Helper for working with Redis Streams consumer groups functionality. Each stream associated with the consumer group is exposed as a special attribute of the `ConsumerGroup` object, exposing stream-specific functionality within the context of the group.

Rather than creating this class directly, use the *Database.consumer_group()* method.

Each registered stream within the group is exposed as a special attribute that provides stream-specific APIs within the context of the group. For more information see `ConsumerGroupStream`.

The streams managed by a consumer group must exist before the consumer group can be created. By default, calling *`ConsumerGroup.create()`* will automatically create stream keys for any that do not exist.

Example:

```
cg = db.consumer_group('groupname', ['stream-1', 'stream-2'])
cg.create()  # Create consumer group.
cg.stream_1  # ConsumerGroupStream for "stream-1"
cg.stream_2  # ConsumerGroupStream for "stream-2"
# or, alternatively:
cg.streams['stream-1']
```

Parameters

- **`database`** (`Database`) – Redis client
- **`name`** – consumer group name
- **`keys`** – stream identifier(s) to monitor. May be a single stream key, a list of stream keys, or a key-to-minimum id mapping. The minimum id for each stream should be considered an exclusive lower-bound. The '$' value can also be used to only read values added *after* our command started blocking.
- **`consumer`** – name for consumer

`consumer`(*name*)

Create a new consumer for the *`ConsumerGroup`*.

Parameters **`name`** – name of consumer

Returns a *`ConsumerGroup`* using the given consumer name.

`create`(*ensure_keys_exist=True*, *mkstream=False*)

Create the consumer group and register it with the group's stream keys.

Parameters

- **`ensure_keys_exist`** – Ensure that the streams exist before creating the consumer group. Streams that do not exist will be created.
- **`mkstream`** – Use the "MKSTREAM" option to ensure stream exists (may require unstable version of Redis).

`destroy`()

Destroy the consumer group.

`read`(*count=None*, *block=None*, *consumer=None*)

Read unseen messages from all streams in the consumer group. Wrapper for `Database.xreadgroup` method.

Parameters

- **`count`** (*int*) – limit number of messages returned
- **`block`** (*int*) – milliseconds to block, 0 for indefinitely.
- **`consumer`** – consumer name

Returns a list of (stream key, messages) tuples, where messages is a list of (message id, data) 2-tuples.

reset()
Reset the consumer group, clearing the last-read status for each stream so it will read from the beginning of each stream.

set_id(*id='$'*)
Set the last-read message id for each stream in the consumer group. By default, this will be the special "$" identifier, meaning all messages are marked as having been read.

Parameters **id** – id of last-read message (or "$").

stream_info()
Retrieve information for each stream managed by the consumer group. Calls `xinfo_stream()` for each stream.

Returns a dictionary mapping stream key to a dictionary of metadata

class walrus.containers.**ConsumerGroupStream**(*Stream*)
Helper for working with an individual stream within the context of a consumer group. This object is exposed as an attribute on a `ConsumerGroup` object using the stream key for the attribute name.

This class should not be created directly. It will automatically be added to the `ConsumerGroup` object.

For example:

```
cg = db.consumer_group('groupname', ['stream-1', 'stream-2'])
cg.stream_1  # ConsumerGroupStream for "stream-1"
cg.stream_2  # ConsumerGroupStream for "stream-2"
```

ack(**id_list*)
Acknowledge that the message(s) were been processed by the consumer associated with the parent `ConsumerGroup`.

Parameters **id_list** – one or more message ids to acknowledge

Returns number of messages marked acknowledged

autoclaim(*consumer*, *min_idle_time*, *start_id=0*, *count=None*, *justid=False*)
Transfer ownership of pending stream entries that match the specified criteria. Similar to calling XPENDING and XCLAIM, but provides a more straightforward way to deal with message delivery failures.

Parameters

- **consumer** – name of consumer that claims the message.
- **min_idle_time** – in milliseconds
- **start_id** – start id
- **count** – optional, upper limit of entries to claim. Default 100.
- **justid** – return just IDs of messages claimed.

Returns [next start id, [messages that were claimed]

claim(**id_list*, ***kwargs*)
Claim pending - but unacknowledged - messages for this stream within the context of the parent `ConsumerGroup`.

Parameters

- **id_list** – one or more message ids to acknowledge
- **min_idle_time** – minimum idle time in milliseconds (keyword-arg).

Returns list of (message id, data) 2-tuples of messages that were successfully claimed

consumers_info()
Retrieve information about consumers within the given consumer group operating on the stream. Calls `xinfo_consumers()`.

Returns a list of dictionaries containing consumer metadata

pending(*start='-'*, *stop='+'*, *count=1000*, *consumer=None*, *idle=None*)
List pending messages within the consumer group for this stream.

Parameters

- **start** – start id (or '-' for oldest pending)
- **stop** – stop id (or '+' for newest pending)
- **count** – limit number of messages returned
- **consumer** – restrict message list to the given consumer
- **idle** (*int*) – filter by idle-time in milliseconds (6.2)

Returns A list containing status for each pending message. Each pending message returns [id, consumer, idle time, deliveries].

read(*count=None*, *block=None*, *last_id=None*)
Monitor the stream for new messages within the context of the parent `ConsumerGroup`.

Parameters

- **count** (*int*) – limit number of messages returned
- **block** (*int*) – milliseconds to block, 0 for indefinitely.
- **last_id** (*str*) – optional last ID, by default uses the special token ">", which reads the oldest unread message.

Returns a list of (message id, data) 2-tuples.

set_id(*id='$'*)
Set the last-read message id for the stream within the context of the parent `ConsumerGroup`. By default this will be the special "$" identifier, meaning all messages are marked as having been read.

Parameters **id** – id of last-read message (or "$").

class walrus.**BitField**(*Container*)
Wrapper that provides a convenient API for constructing and executing Redis BITFIELD commands. The BITFIELD command can pack multiple operations into a single logical command, so the *BitField* supports a method- chaining API that allows multiple operations to be performed atomically.

Rather than instantiating this class directly, you should use the *Database.bit_field()* method to obtain a *BitField*.

__delitem__(*item*)
Clear a range of bits in a bitfield. Note that the item **must** be a slice specifying the start and end of the range of bits to clear.

__getitem__(*item*)
Short-hand for getting a range of bits in a bitfield. Note that the item **must** be a slice specifying the start and end of the range of bits to read.

__setitem__(*item*, *value*)
Short-hand for setting a range of bits in a bitfield. Note that the item **must** be a slice specifying the start and end of the range of bits to read. If the value representation exceeds the number of bits implied by the slice range, a `ValueError` is raised.

bit_count(*start=None*, *end=None*)
Count the set bits in a string. Note that the *start* and *end* parameters are offsets in **bytes**.

get(*fmt*, *offset*)
Get the value of a given bitfield.

Parameters

- **fmt** – format-string for the bitfield being read, e.g. u8 for an unsigned 8-bit integer.
- **offset** (*int*) – offset (in number of bits).

Returns a `BitFieldOperation` instance.

get_bit(*offset*)
Get the bit value at the given offset (in bits).

Parameters **offset** (*int*) – bit offset

Returns value at bit offset, 1 or 0

get_raw()
Return the raw bytestring that comprises the bitfield. Equivalent to a normal GET command.

incrby(*fmt*, *offset*, *increment*, *overflow=None*)
Increment a bitfield by a given amount.

Parameters

- **fmt** – format-string for the bitfield being updated, e.g. u8 for an unsigned 8-bit integer.
- **offset** (*int*) – offset (in number of bits).
- **increment** (*int*) – value to increment the bitfield by.
- **overflow** (*str*) – overflow algorithm. Defaults to WRAP, but other acceptable values are SAT and FAIL. See the Redis docs for descriptions of these algorithms.

Returns a `BitFieldOperation` instance.

set(*fmt*, *offset*, *value*)
Set the value of a given bitfield.

Parameters

- **fmt** – format-string for the bitfield being read, e.g. u8 for an unsigned 8-bit integer.
- **offset** (*int*) – offset (in number of bits).
- **value** (*int*) – value to set at the given position.

Returns a `BitFieldOperation` instance.

set_bit(*offset*, *value*)
Set the bit value at the given offset (in bits).

Parameters

- **offset** (*int*) – bit offset
- **value** (*int*) – new value for bit, 1 or 0

Returns previous value at bit offset, 1 or 0

set_raw(*value*)
Set the raw bytestring that comprises the bitfield. Equivalent to a normal SET command.

class walrus.containers.**BitFieldOperation**(*database*, *key*)
Command builder for BITFIELD commands.

__iter__()
Implicit execution and iteration of the return values for a sequence of operations.

execute()
Execute the operation(s) in a single BITFIELD command. The return value is a list of values corresponding to each operation.

get(*fmt*, *offset*)
Get the value of a given bitfield.

Parameters

- **fmt** – format-string for the bitfield being read, e.g. u8 for an unsigned 8-bit integer.
- **offset** (*int*) – offset (in number of bits).

Returns a *BitFieldOperation* instance.

incrby(*fmt*, *offset*, *increment*, *overflow=None*)
Increment a bitfield by a given amount.

Parameters

- **fmt** – format-string for the bitfield being updated, e.g. u8 for an unsigned 8-bit integer.
- **offset** (*int*) – offset (in number of bits).
- **increment** (*int*) – value to increment the bitfield by.
- **overflow** (*str*) – overflow algorithm. Defaults to WRAP, but other acceptable values are SAT and FAIL. See the Redis docs for descriptions of these algorithms.

Returns a *BitFieldOperation* instance.

set(*fmt*, *offset*, *value*)
Set the value of a given bitfield.

Parameters

- **fmt** – format-string for the bitfield being read, e.g. u8 for an unsigned 8-bit integer.
- **offset** (*int*) – offset (in number of bits).
- **value** (*int*) – value to set at the given position.

Returns a *BitFieldOperation* instance.

class walrus.**BloomFilter**(*Container*)
Bloom-filters are probabilistic data-structures that are used to answer the question: "is X a member of set S?" It is possible to receive a false positive, but impossible to receive a false negative (in other words, if the bloom filter contains a value, it will never erroneously report that it does *not* contain such a value). The accuracy of the bloom-filter and the likelihood of a false positive can be reduced by increasing the size of the bloomfilter. The default size is 64KB (or 524,288 bits).

Rather than instantiate this class directly, use *Database.bloom_filter()*.

__contains__(*data*)
Check if an item has been added to the bloomfilter.

Parameters **data** (*bytes*) – a bytestring representing the item to check.

Returns a boolean indicating whether or not the item is present in the bloomfilter. False-positives are possible, but a negative return value is definitive.

add(*data*)
Add an item to the bloomfilter.

Parameters **data** (*bytes*) – a bytestring representing the item to add.

contains(*data*)
Check if an item has been added to the bloomfilter.

Parameters **data** (*bytes*) – a bytestring representing the item to check.

Returns a boolean indicating whether or not the item is present in the bloomfilter. False-positives are possible, but a negative return value is definitive.

1.11.2 High-level APIs

class walrus.**Autocomplete**(*database*, *namespace='walrus'*, *cache_timeout=600*, *stopwords_file='stopwords.txt'*, *use_json=True*)
Autocompletion for ascii-encoded string data. Titles are stored, along with any corollary data in Redis. Substrings of the title are stored in sorted sets using a unique scoring algorithm. The scoring algorithm aims to return results in a sensible order, by looking at the entire title and the position of the matched substring within the title.

Additionally, the autocomplete object supports boosting search results by object ID or object type.

__init__(*database*, *namespace='walrus'*, *cache_timeout=600*, *stopwords_file='stopwords.txt'*, *use_json=True*)

Parameters

- **database** – A *Database* instance.
- **namespace** – Namespace to prefix keys used to store metadata.
- **cache_timeout** – Complex searches using boosts will be cached. Specify the amount of time these results are cached for.
- **stopwords_file** – Filename containing newline-separated stopwords. Set to *None* to disable stopwords filtering.
- **use_json** (*bool*) – Whether object data should be serialized as JSON.

boost_object(*obj_id=None*, *obj_type=None*, *multiplier=1.1*, *relative=True*)
Boost search results for the given object or type by the amount specified. When the `multiplier` is greater than 1, the results will percolate to the top. Values between 0 and 1 will percolate results to the bottom.

Either an `obj_id` or `obj_type` (or both) must be specified.

Parameters

- **obj_id** – An object's unique identifier (optional).
- **obj_type** – The object's type (optional).
- **multiplier** – A positive floating-point number.
- **relative** – If `True`, then any pre-existing saved boost will be updated using the given multiplier.

Examples:

```
# Make all objects of type=photos percolate to top.
ac.boost_object(obj_type='photo', multiplier=2.0)

# Boost a particularly popular blog entry.
ac.boost_object(
    popular_entry.id,
    'entry',
    multipler=5.0,
    relative=False)
```

exists(*obj_id*, *obj_type=None*)
Return whether the given object exists in the search index.

Parameters

- **obj_id** – The object's unique identifier.
- **obj_type** – The object's type.

flush(*batch_size=1000*)
Delete all autocomplete indexes and metadata.

list_data()
Return all the data stored in the autocomplete index. If the data was stored as serialized JSON, then it will be de-serialized before being returned.

Return type list

list_titles()
Return the titles of all objects stored in the autocomplete index.

Return type list

remove(*obj_id*, *obj_type=None*)
Remove an object identified by the given `obj_id` (and optionally `obj_type`) from the search index.

Parameters

- **obj_id** – The object's unique identifier.
- **obj_type** – The object's type.

search(*phrase*, *limit=None*, *boosts=None*, *chunk_size=1000*)
Perform a search for the given phrase. Objects whose title matches the search will be returned. The values returned will be whatever you specified as the `data` parameter when you called *`store()`*.

Parameters

- **phrase** – One or more words or substrings.
- **limit** (*int*) – Limit size of the result set.
- **boosts** (*dict*) – A mapping of object id/object type to floating point multipliers.

Returns A list containing the object data for objects matching the search phrase.

store(*obj_id*, *title=None*, *data=None*, *obj_type=None*)
Store data in the autocomplete index.

Parameters

- **obj_id** – Either a unique identifier for the object being indexed or the word/phrase to be indexed.

- **title** – The word or phrase to be indexed. If not provided, the `obj_id` will be used as the title.
- **data** – Arbitrary data to index, which will be returned when searching for results. If not provided, this value will default to the title being indexed.
- **obj_type** – Optional object type. Since results can be boosted by type, you might find it useful to specify this when storing multiple types of objects.

You have the option of storing several types of data as defined by the parameters. At the minimum, you can specify an `obj_id`, which will be the word or phrase you wish to index. Alternatively, if for instance you were indexing blog posts, you might specify all parameters.

class `walrus.`**Cache**(*database*, *name='cache'*, *default_timeout=None*, *debug=False*)

Cache implementation with simple `get/set` operations, and a decorator.

__init__(*database*, *name='cache'*, *default_timeout=None*, *debug=False*)

Parameters

- **database** – *Database* instance.
- **name** – Namespace for this cache.
- **default_timeout** (*int*) – Default cache timeout.
- **debug** – Disable cache for debugging purposes. Cache will no-op.

cache_async(*key_fn=<function Cache._key_fn>*, *timeout=3600*)

Decorator that will execute the cached function in a separate thread. The function will immediately return, returning a callable to the user. This callable can be used to check for a return value.

For details, see the *Cache Asynchronously* section of the docs.

Parameters

- **key_fn** – Function used to generate cache key.
- **timeout** (*int*) – Cache timeout in seconds.

Returns A new function which can be called to retrieve the return value of the decorated function.

cached(*key_fn=<function Cache._key_fn>*, *timeout=None*, *metrics=False*)

Decorator that will transparently cache calls to the wrapped function. By default, the cache key will be made up of the arguments passed in (like memoize), but you can override this by specifying a custom `key_fn`.

Parameters

- **key_fn** – Function used to generate a key from the given args and kwargs.
- **timeout** – Time to cache return values.
- **metrics** – Keep stats on cache utilization and timing.

Returns Return the result of the decorated function call with the given args and kwargs.

Usage:

```
cache = Cache(my_database)

@cache.cached(timeout=60)
def add_numbers(a, b):
    return a + b
```

(continues on next page)

(continued from previous page)

```
print add_numbers(3, 4)  # Function is called.
print add_numbers(3, 4)  # Not called, value is cached.

add_numbers.bust(3, 4)  # Clear cache for (3, 4).
print add_numbers(3, 4)  # Function is called.
```

The decorated function also gains a new attribute named `bust` which will clear the cache for the given args.

cached_property(*key_fn=<function Cache._key_fn>*, *timeout=None*)

Decorator that will transparently cache calls to the wrapped method. The method will be exposed as a property.

Usage:

```
cache = Cache(my_database)

class Clock(object):
    @cache.cached_property()
    def now(self):
        return datetime.datetime.now()

clock = Clock()
print clock.now
```

delete(*key*)

Remove the given key from the cache.

delete_many(*keys*)

Delete multiple keys from the cache in one operation.

Parameters **keys** (*list*) – keys to delete.

Returns number of keys removed.

flush()

Remove all cached objects from the database.

get(*key*, *default=None*)

Retreive a value from the cache. In the event the value does not exist, return the `default`.

get_many(*keys*)

Retrieve multiple values from the cache. Missing keys are not included in the result dictionary.

Parameters **keys** (*list*) – list of keys to fetch.

Returns dictionary mapping keys to cached values.

keys()

Return all keys for cached values.

set(*key*, *value*, *timeout=None*)

Cache the given `value` in the specified `key`. If no timeout is specified, the default timeout will be used.

set_many(*_Cache__data=None*, *timeout=None*, ***kwargs*)

Set multiple key/value pairs in one operation.

Parameters

- **__data** (*dict*) – provide data as dictionary of key/value pairs.

- **timeout** – optional timeout for data.
- **kwargs** – alternatively, provide data as keyword arguments.

Returns True on success.

class walrus.**Counter**(*database*, *name*)
Simple counter.

__init__(*database*, *name*)

Parameters

- **database** – A walrus Database instance.
- **name** (*str*) – The name for the counter.

class walrus.**Index**(*db*, *name*, ***tokenizer_settings*)
Full-text search index.

Store documents, along with arbitrary metadata, and perform full-text search on the document content. Supports porter-stemming, stopword filtering, basic result ranking, and (optionally) double-metaphone for phonetic search.

__init__(*db*, *name*, ***tokenizer_settings*)

Parameters

- **db** (Database) – a walrus database object.
- **name** (*str*) – name for the search index.
- **stemmer** (*bool*) – use porter stemmer (default True).
- **metaphone** (*bool*) – use double metaphone (default False).
- **stopwords_file** (*str*) – defaults to walrus stopwords.txt.
- **min_word_length** (*int*) – specify minimum word length.

Create a search index for storing and searching documents.

add(*key*, *content*, *_Index__metadata=None*, ***metadata*)

Parameters

- **key** – Document unique identifier.
- **content** (*str*) – Content to store and index for search.
- **metadata** – Arbitrary key/value pairs to store for document.

Add a document to the search index.

get_document(*document_id*)

Parameters **document_id** – Document unique identifier.

Returns a dictionary containing the document content and any associated metadata.

remove(*key*, *preserve_data=False*)

Parameters **key** – Document unique identifier.

Remove the document from the search index.

replace(*key*, *content*, *_Index__metadata=None*, ***metadata*)

Parameters

- **key** – Document unique identifier.
- **content** (*str*) – Content to store and index for search.
- **metadata** – Arbitrary key/value pairs to store for document.

Update the given document. Existing metadata will not be removed and replaced with the provided metadata.

search(*query*)

Parameters **query** (*str*) – Search query. May contain boolean/set operations and parentheses.

Returns a list of document hashes corresponding to matching documents.

Search the index. The return value is a list of dictionaries corresponding to the documents that matched. These dictionaries contain a `content` key with the original indexed content, along with any additional metadata that was specified.

search_items(*query*)

Parameters **query** (*str*) – Search query. May contain boolean/set operations and parentheses.

Returns a list of (key, document hashes) tuples corresponding to matching documents.

Search the index. The return value is a list of (key, document dict) corresponding to the documents that matched. These dictionaries contain a `content` key with the original indexed content, along with any additional metadata that was specified.

update(*key*, *content*, *_Index__metadata=None*, ***metadata*)

Parameters

- **key** – Document unique identifier.
- **content** (*str*) – Content to store and index for search.
- **metadata** – Arbitrary key/value pairs to store for document.

Update the given document. Existing metadata will be preserved and, optionally, updated with the provided metadata.

class walrus.**Graph**(*walrus*, *namespace*)

Simple hexastore built using Redis ZSets. The basic idea is that we have a collection of relationships of the form subject-predicate-object. For example:

- charlie – friends – huey
- charlie – lives – Kansas
- huey – lives – Kansas

We might wish to ask questions of our data-store like "which of charlie's friends live in Kansas?" To do this we will store every permutation of the S-P-O triples, then we can do efficient queries using the parts of the relationship we know:

- query the "object" portion of the "charlie – friends" subject and predicate.
- for each object returned, turn it into the subject of a second query whose predicate is "lives" and whose object is "Kansas"

So we would return the subjects that satisfy the following expression:

```
("charlie -- friends") -- lives -- Kansas.
```

To accomplish this in Python we could write:

```
db = Database()
graph = db.graph('people')

# Store my friends.
graph.store_many(
    ('charlie', 'friends', 'huey'),
    ('charlie', 'friends', 'zaizee'),
    ('charlie', 'friends', 'nuggie'))

# Store where people live.
graph.store_many(
    ('huey', 'lives', 'Kansas'),
    ('zaizee', 'lives', 'Missouri'),
    ('nuggie', 'lives', 'Kansas'),
    ('mickey', 'lives', 'Kansas'))

# Perform our search. We will use a variable (X) to indicate the
# value we're interested in.
X = graph.v.X  # Create a variable placeholder.

# In the first clause we indicate we are searching for my friends.
# In the second clause, we only want those friends who also live in
# Kansas.
results = graph.search(
    {'s': 'charlie', 'p': 'friends', 'o': X},
    {'s': X, 'p': 'lives', 'o': 'Kansas'})
print results

# Prints: {'X': {'huey', 'nuggie'}}
```

See: http://redis.io/topics/indexes#representing-and-querying-graphs-using-an-hexastore

__init__(*walrus*, *namespace*)
Initialize self. See help(type(self)) for accurate signature.

delete(*s*, *p*, *o*)
Remove the given subj-pred-obj triple from the database.

query(*s=None*, *p=None*, *o=None*)
Return all triples that satisfy the given expression. You may specify all or none of the fields (s, p, and o). For instance, if I wanted to query for all the people who live in Kansas, I might write:

```
for triple in graph.query(p='lives', o='Kansas'):
    print triple['s'], 'lives in Kansas!'
```

search(**conditions*)
Given a set of conditions, return all values that satisfy the conditions for a given set of variables.

For example, suppose I wanted to find all of my friends who live in Kansas:

```
X = graph.v.X
results = graph.search(
    {'s': 'charlie', 'p': 'friends', 'o': X},
    {'s': X, 'p': 'lives', 'o': 'Kansas'})
```

The return value consists of a dictionary keyed by variable, whose values are `set` objects containing the values that satisfy the query clauses, e.g.:

```
print results

# Result has one key, for our "X" variable. The value is the set
# of my friends that live in Kansas.
# {'X': {'huey', 'nuggie'}}

# We can assume the following triples exist:
# ('charlie', 'friends', 'huey')
# ('charlie', 'friends', 'nuggie')
# ('huey', 'lives', 'Kansas')
# ('nuggie', 'lives', 'Kansas')
```

store(*s*, *p*, *o*)
Store a subject-predicate-object triple in the database.

store_many(*items*)
Store multiple subject-predicate-object triples in the database.

Parameters **items** – A list of (subj, pred, obj) 3-tuples.

v(*name*)
Create a named variable, used to construct multi-clause queries with the *Graph.search()* method.

class walrus.**Lock**(*database*, *name*, *ttl=None*, *lock_id=None*)
Lock implementation. Can also be used as a context-manager or decorator.

Unlike the redis-py lock implementation, this Lock does not use a spin-loop when blocking to acquire the lock. Instead, it performs a blocking pop on a list. When a lock is released, a value is pushed into this list, signalling that the lock is available.

The lock uses Lua scripts to ensure the atomicity of its operations.

You can set a TTL on a lock to reduce the potential for deadlocks in the event of a crash. If a lock is not released before it exceeds its TTL, and threads that are blocked waiting for the lock could potentially re-acquire it.

Note: TTL is specified in **milliseconds**.

Locks can be used as context managers or as decorators:

```
lock = db.lock('my-lock')

with lock:
    perform_some_calculations()

@lock
def another_function():
    # The lock will be acquired when this function is
    # called, and released when the function returns.
    do_some_more_calculations()
```

__init__(*database*, *name*, *ttl=None*, *lock_id=None*)

Parameters

- **database** – A walrus Database instance.
- **name** (*str*) – The name for the lock.
- **ttl** (*int*) – The time-to-live for the lock in milliseconds.

- **lock_id** (*str*) – Unique identifier for the lock instance.

acquire(*block=True*)
Acquire the lock. The lock will be held until it is released by calling *Lock.release()*. If the lock was initialized with a `ttl`, then the lock will be released automatically after the given number of milliseconds.

By default this method will block until the lock becomes free (either by being released or expiring). The blocking is accomplished by performing a blocking left-pop on a list, as opposed to a spin-loop.

If you specify `block=False`, then the method will return `False` if the lock could not be acquired.

Parameters **block** (*bool*) – Whether to block while waiting to acquire the lock.

Returns Returns `True` if the lock was acquired.

clear()
Clear the lock, allowing it to be acquired. Do not use this method except to recover from a deadlock. Otherwise you should use *Lock.release()*.

release()
Release the lock.

Returns Returns `True` if the lock was released.

class walrus.**Model**(**args*, ***kwargs*)
A collection of fields to be stored in the database. Walrus stores model instance data in hashes keyed by a combination of model name and primary key value. Instance attributes are automatically converted to values suitable for storage in Redis (i.e., datetime becomes timestamp), and vice-versa.

Additionally, model fields can be `indexed`, which allows filtering. There are three types of indexes:

- Absolute
- Scalar
- Full-text search

Absolute indexes are used for values like strings or UUIDs and support only equality and inequality checks.

Scalar indexes are for numeric values as well as datetimes, and support equality, inequality, and greater or less-than.

The final type of index, FullText, can only be used with the *TextField*. FullText indexes allow search using the `match()` method. For more info, see *Full-text search*.

__database__ = None
Required: the *Database* instance to use to persist model data.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__namespace__ = None
Optional: namespace to use for model data.

classmethod all()
Return an iterator that successively yields saved model instances. Models are saved in an unordered *Set*, so the iterator will return them in arbitrary order.

Example:

```
for note in Note.all():
    print note.content
```

To return models in sorted order, see *Model.query()*. Example returning all records, sorted newest to oldest:

```
for note in Note.query(order_by=Note.timestamp.desc()):
    print note.timestamp, note.content
```

classmethod count()
Return the number of objects in the given collection.

classmethod create(***kwargs*)
Create a new model instance and save it to the database. Values are passed in as keyword arguments.

Example:

```
user = User.create(first_name='Charlie', last_name='Leifer')
```

delete(*for_update=False*)
Delete the given model instance.

classmethod get(*expression*)
Retrieve the model instance matching the given expression. If the number of matching results is not equal to one, then a `ValueError` will be raised.

Parameters **expression** – A boolean expression to filter by.

Returns The matching *Model* instance.

Raises `ValueError` if result set size is not 1.

incr(*field*, *incr_by=1*)
Increment the value stored in the given field by the specified amount. Any indexes will be updated at the time `incr()` is called.

Parameters

- **field** (`Field`) – A field instance.
- **incr_by** – An `int` or `float`.

Example:

```
# Retrieve a page counter object for the given URL.
page_count = PageCounter.get(PageCounter.url == url)

# Update the hit count, persisting to the database and
# updating secondary indexes in one go.
page_count.incr(PageCounter.hits)
```

index_separator = '.'
Required: character to use as a delimiter for indexes, default ".".

classmethod load(*primary_key*, *convert_key=True*)
Retrieve a model instance by primary key.

Parameters **primary_key** – The primary key of the model instance.

Returns Corresponding *Model* instance.

Raises `KeyError` if object with given primary key does not exist.

classmethod query(*expression=None*, *order_by=None*)
Return model instances matching the given expression (if specified). Additionally, matching instances can be returned sorted by field value.

Example:

```
# Get administrators sorted by username.
admin_users = User.query(
    (User.admin == True),
    order_by=User.username)

# List blog entries newest to oldest.
entries = Entry.query(order_by=Entry.timestamp.desc())

# Perform a complex filter.
values = StatData.query(
    (StatData.timestamp < datetime.date.today()) &
    ((StatData.type == 'pv') | (StatData.type == 'cv')))
```

Parameters

- **expression** – A boolean expression to filter by.
- **order_by** – A field whose value should be used to sort returned instances.

classmethod query_delete(*expression=None*)

Delete model instances matching the given expression (if specified). If no expression is provided, then all model instances will be deleted.

Parameters **expression** – A boolean expression to filter by.

save(*_is_create=False*)

Save the given model instance. If the model does not have a primary key value, Walrus will call the primary key field's `generate_key()` method to attempt to generate a suitable value.

to_hash()

Return a *Hash* instance corresponding to the raw model data.

class walrus.**RateLimit**(*database*, *name*, *limit=5*, *per=60*, *debug=False*)

Rate limit implementation. Allows up to "limit" number of events every per the given number of seconds.

__init__(*database*, *name*, *limit=5*, *per=60*, *debug=False*)

Parameters

- **database** – *Database* instance.
- **name** – Namespace for this cache.
- **limit** (*int*) – Number of events allowed during a given time period.
- **per** (*int*) – Time period the `limit` applies to, in seconds.
- **debug** – Disable rate-limit for debugging purposes. All events will appear to be allowed and valid.

limit(*key*)

Function to log an event with the given key. If the `key` has not exceeded their allotted events, then the function returns `False` to indicate that no limit is being imposed.

If the `key` has exceeded the number of events, then the function returns `True` indicating rate-limiting should occur.

Parameters **key** (*str*) – A key identifying the source of the event.

Returns Boolean indicating whether the event should be rate-limited or not.

rate_limited(*key_function=None*)
Function or method decorator that will prevent calls to the decorated function when the number of events has been exceeded for the given time period.

It is probably important that you take care to choose an appropriate key function. For instance, if rate-limiting a web-page you might use the requesting user's IP as the key.

If the number of allowed events has been exceeded, a `RateLimitException` will be raised.

Parameters **key_function** – Function that accepts the params of the decorated function and returns a string key. If not provided, a hash of the args and kwargs will be used.

Returns If the call is not rate-limited, then the return value will be that of the decorated function.

Raises `RateLimitException`.

class walrus.**TimeSeries**(*ConsumerGroup*)
TimeSeries is a consumer-group that provides a higher level of abstraction, reading and writing message ids as datetimes, and returning messages using a convenient, lightweight `Message` class.

Rather than creating this class directly, use the *Database.time_series()* method.

Each registered stream within the group is exposed as a special attribute that provides stream-specific APIs within the context of the group. For more information see `TimeSeriesStream`.

Example:

```
ts = db.time_series('groupname', ['stream-1', 'stream-2'])
ts.stream_1  # TimeSeriesStream for "stream-1"
ts.stream_2  # TimeSeriesStream for "stream-2"
```

Parameters

- **database** (`Database`) – Redis client
- **group** – name of consumer group
- **keys** – stream identifier(s) to monitor. May be a single stream key, a list of stream keys, or a key-to-minimum id mapping. The minimum id for each stream should be considered an exclusive lower-bound. The '$' value can also be used to only read values added *after* our command started blocking.
- **consumer** – name for consumer within group

Returns a *TimeSeries* instance

consumer(*name*)
Create a new consumer for the *ConsumerGroup*.

Parameters **name** – name of consumer

Returns a *ConsumerGroup* using the given consumer name.

create(*ensure_keys_exist=True*, *mkstream=False*)
Create the consumer group and register it with the group's stream keys.

Parameters

- **ensure_keys_exist** – Ensure that the streams exist before creating the consumer group. Streams that do not exist will be created.
- **mkstream** – Use the "MKSTREAM" option to ensure stream exists (may require unstable version of Redis).

destroy()
Destroy the consumer group.

read(*count=None*, *block=None*)
Read unseen messages from all streams in the consumer group. Wrapper for `Database.xreadgroup` method.

Parameters

- **count** (*int*) – limit number of messages returned
- **block** (*int*) – milliseconds to block, 0 for indefinitely.

Returns a list of `Message` objects

reset()
Reset the consumer group, clearing the last-read status for each stream so it will read from the beginning of each stream.

set_id(*id='$'*)
Set the last-read message id for each stream in the consumer group. By default, this will be the special "$" identifier, meaning all messages are marked as having been read.

Parameters **id** – id of last-read message (or "$").

1.11.3 Field types

class walrus.**Field**(*index=False*, *primary_key=False*, *default=None*)
Named attribute on a model that will hold a value of the given type. Fields are declared as attributes on a model class.

Example:

```
walrus_db = Database()

class User(Model):
    __database__ = walrus_db
    __namespace__ = 'my-app'

    # Use the user's email address as the primary key.
    # All primary key fields will also get a secondary
    # index, so there's no need to specify index=True.
    email = TextField(primary_key=True)

    # Store the user's interests in a free-form text
    # field. Also create a secondary full-text search
    # index on this field.
    interests = TextField(
        fts=True,
        stemmer=True,
        min_word_length=3)

class Note(Model):
    __database__ = walrus_app
    __namespace__ = 'my-app'

    # A note is associated with a user. We will create a
    # secondary index on this field so we can efficiently
    # retrieve all notes created by a specific user.
```

(continues on next page)

(continued from previous page)

```
    user_email = TextField(index=True)

    # Store the note content in a searchable text field. Use
    # the double-metaphone algorithm to index the content.
    content = TextField(
        fts=True,
        stemmer=True,
        metaphone=True)

    # Store the timestamp the note was created automatically.
    # Note that we do not call `now()`, but rather pass the
    # function itself.
    timestamp = DateTimeField(default=datetime.datetime.now)
```

__init__(*index=False*, *primary_key=False*, *default=None*)

Parameters

- **index** (*bool*) – Use this field as an index. Indexed fields will support *Model.get()* lookups.
- **primary_key** (*bool*) – Use this field as the primary key.

get_indexes()

Return a list of secondary indexes to create for the field. For instance, a TextField might have a full-text search index, whereas an IntegerField would have a scalar index that supported range queries.

class walrus.**TextField**(*fts=False*, *stemmer=True*, *metaphone=False*, *stopwords_file=None*, *min_word_length=None*, **args*, ***kwargs*)

Store unicode strings, encoded as UTF-8. *TextField* also supports full-text search through the optional `fts` parameter.

Note: If full-text search is enabled for the field, then the `index` argument is implied.

Parameters

- **fts** (*bool*) – Enable simple full-text search.
- **stemmer** (*bool*) – Use porter stemmer to process words.
- **metaphone** (*bool*) – Use the double metaphone algorithm to process words.
- **stopwords_file** (*str*) – File containing stopwords, one per line. If not specified, the default stopwords will be used.
- **min_word_length** (*int*) – Minimum length (inclusive) of word to be included in search index.

search(*query*[, *default_conjunction='and'*])

Parameters

- **query** (*str*) – Search query.
- **default_conjunction** (*str*) – Either `'and'` or `'or'`.

Create an expression corresponding to the given search query. Search queries can contain conjunctions (`AND` and `OR`).

Example:

```
class Message(Model):
    database = my_db
    content = TextField(fts=True)

expression = Message.content.search('python AND (redis OR walrus)')
messages = Message.query(expression)
for message in messages:
    print message.content
```

get_indexes()
Return a list of secondary indexes to create for the field. For instance, a TextField might have a full-text search index, whereas an IntegerField would have a scalar index that supported range queries.

class walrus.**IntegerField**(*index=False*, *primary_key=False*, *default=None*)
Store integer values.

class walrus.**AutoIncrementField**(*IntegerField*)
Auto-incrementing primary key field.

class walrus.**FloatField**(*index=False*, *primary_key=False*, *default=None*)
Store floating point values.

class walrus.**ByteField**(*index=False*, *primary_key=False*, *default=None*)
Store arbitrary bytes.

class walrus.**BooleanField**(*index=False*, *primary_key=False*, *default=None*)
Store boolean values.

class walrus.**UUIDField**(***kwargs*)
Store unique IDs. Can be used as primary key.

class walrus.**DateTimeField**(*index=False*, *primary_key=False*, *default=None*)
Store Python datetime objects.

class walrus.**DateField**(*index=False*, *primary_key=False*, *default=None*)
Store Python date objects.

class walrus.**JSONField**(*index=False*, *primary_key=False*, *default=None*)
Store arbitrary JSON data.

Container Field Types

class walrus.**HashField**(**args*, ***kwargs*)
Store values in a Redis hash.

container_class
alias of walrus.containers.Hash

class walrus.**ListField**(**args*, ***kwargs*)
Store values in a Redis list.

container_class
alias of walrus.containers.List

class walrus.**SetField**(**args*, ***kwargs*)
Store values in a Redis set.

container_class
alias of walrus.containers.Set

class walrus.**ZSetField**(**args*, ***kwargs*)
Store values in a Redis sorted set.

container_class
alias of walrus.containers.ZSet

1.12 Alternative Backends ("tusks")

In addition to Redis, I've been experimenting with adding support for alternative *redis-like* backends. These alternative backends are referred to as *tusks*, and currently Walrus supports the following:

- RLite, a self-contained and serverless Redis-compatible database engine. Use `rlite` if you want all the features of Redis, without the separate server process..
- Vedis, an embeddable data-store written in C with over 70 commands similar in concept to Redis. Vedis is built on a fast key/value store and supports writing custom commands in Python. Use `vedis` if you are OK working with a smaller subset of commands out-of-the-box or are interested in writing your own commands.
- ledisdb, Redis-like database written in Golang. Supports almost all the Redis commands. Requires ledis-py.

1.12.1 rlite

rlite is an embedded Redis-compatible database.

According to the project's README,

> rlite is to Redis what SQLite is to Postgres.

The project's features are:

- **Supports virtually every Redis command**.
- Self-contained embedded data-store.
- Serverless / zero-configuration.
- Transactions.
- Databases can be in-memory or stored in a single file on-disk.

Use-cases for `rlite`:

- **Mobile** environments, where it is not practical to run a database server.
- **Development** or **testing** environments. Database fixtures can be distributed as a simple binary file.
- **Slave of Redis** for additional durability.
- Application file format, alternative to a proprietary format or SQLite.

Python bindings

rlite-py allows `rlite` to be embedded in your Python apps. To install `rlite-py`, you can use `pip`:

```
$ pip install hirlite
```

Using with Walrus

To use `rlite` instead of Redis in your `walrus` application, simply use the `WalrusLite` in place of the usual `Walrus` object:

```
from walrus.tusks.rlite import WalrusLite

walrus = WalrusLite('/path/to/database.db')
```

`WalrusLite` can also be used as an in-memory database by omitting a path to a database file when instantiating, or by passing the special string `':memory:'`:

```
from walrus.tusks.rlite import WalrusLite

walrus_mem_db = WalrusLite(':memory:')
```

1.12.2 Vedis

Vedis is an embedded Redis-like database with over 70 commands. `Vedis`, like `rlite`, does not have a separate server process. And like `rlite`, `Vedis` supports both file-backed databases and transient in-memory databases.

According to the project's README,

> Vedis is a self-contained C library without dependency. It requires very minimal support from external libraries or from the operating system. This makes it well suited for use in embedded devices that lack the support infrastructure of a desktop computer. This also makes Vedis appropriate for use within applications that need to run without modification on a wide variety of computers of varying configurations.

The project's features are:

- Serverless / zero-configuration.
- Transactional (ACID) datastore.
- Databases can be in-memory or stored in a single file on-disk.
- Over 70 commands covering many Redis features.
- Cross-platform file format.
- Includes fast low-level key/value store.
- Thread-safe and fully re-entrant.
- Support for Terabyte-sized databases.
- Python bindings allow you to write your own Vedis commands in Python.

Use-cases for `Vedis`:

- **Mobile** environments, where it is not practical to run a database server.
- **Development** or **testing** environments. Database fixtures can be distributed as a simple binary file.
- Application file format, alternative to a proprietary format or SQLite.
- Extremely large databases that do not fit in RAM.
- Embedded platforms with limited resources.

Note: Unlike `rlite`, which supports virtually all the Redis commands, `Vedis` supports a more limited subset. Notably lacking are sorted-set operations and many of the list operations. Hashes, Sets and key/value operations are very well supported, though.

Warning: The authors of Vedis have indicated that they are not actively working on new features for Vedis right now.

Python bindings

vedis-python allows `Vedis` to be embedded in your Python apps. To install `vedis-python`, you can use `pip`:

```
$ pip install vedis
```

Using with Walrus

To use `Vedis` instead of Redis in your `walrus` application, simply use the `WalrusVedis` in place of the usual `Walrus` object:

```
from walrus.tusks.vedisdb import WalrusVedis

walrus = WalrusVedis('/path/to/database.db')
```

`WalrusVedis` can also be used as an in-memory database by omitting a path to a database file when instantiating, or by passing the special string `':memory:'`:

```
from walrus.tusks.vedisdb import WalrusVedis

walrus_mem_db = WalrusVedis(':memory:')
```

Writing a custom command

One of the neat features of `Vedis` is the ease with which you can write your own commands. Here are a couple examples:

```
from walrus.tusks.vedisdb import WalrusVedis

db = WalrusVedis()  # Create an in-memory database.

@db.command('SUNION')  # Vedis supports SDIFF and SINTER, but not SUNION.
def sunion(context, key1, key2):
    return list(db.smembers(key1) | db.smembers(key2))

@db.command('KTITLE')  # Access the low-level key/value store via the context.
def ktitle(context, source, dest_key):
    source_val = context[source]
    if source_val:
        context[dest_key] = source_val.title()
        return True
    return False
```

We can use these commands like so:

```
>>> s1 = db.Set('s1')
>>> s1.add(*range(3))
3
>>> s2.add(*range(1, 5))
4
>>> db.SUNION('s1', 's2')
['1', '0', '3', '2', '4']

>>> db['user.1.username'] = 'charles'
>>> db.KTITLE('user.1.username', 'user.1.display_name')
1
>>> print db['user.1.display_name']
Charles
```

1.12.3 Ledis

ledis is a Redis-like database written in Golang.

The project's features are:

- **Supports virtually every Redis command**.
- Supports multiple backends, including LevelDB, RocksDB, LMDB, BoltDB and in-memory databases.
- Data storage is not limited by RAM, since the databases are disk-based.
- Transactions.
- Supports the Redis protocol for communication, so most Redis clients work with Ledis.
- Written in golang, easy to deploy.

Use-cases for `ledisdb`:

- Store data-sets that exceed RAM.
- Use with LevelDB, RocksDB, etc.

Python bindings

ledis-py allows you to connect to `ledisdb`. To install `ledis-py`, you can use `pip`:

```
$ pip install ledis
```

Using with Walrus

To use `ledisdb` instead of Redis in your `walrus` application, simply use the `WalrusLedis` in place of the usual `Walrus` object:

```
from walrus.tusks.ledisdb import WalrusLedis

walrus = WalrusLedis()
```

1.13 Contributing

I'd love help making walrus a better, more useful library so if you have any questions, comments or suggestions please feel free to open a GitHub ticket:

https://github.com/coleifer/walrus/issues/new

1.13.1 Found a bug?

If you think you've found a bug in walrus, please create a GitHub ticket and include any traceback if applicable.

https://github.com/coleifer/walrus/issues/new

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

P

Q

R

S

T

U

V

W

X

Z