unqlite-python Documentation

Release 0.2.0

charles leifer

May 01, 2017

Contents

Fast Python bindings for UnQLite, a lightweight, embedded NoSQL database and JSON document store.

UnQLite features:

- Embedded, zero-conf database
- Transactional (ACID)
- Single file or in-memory database
- Key/value store
- Cursor support and linear record traversal
- JSON document store
- Thread-safe
- Terabyte-sized databases

UnQLite-Python features:

- Compiled library, extremely fast with minimal overhead.
- Supports key/value operations, cursors, and transactions using Pythonic APIs.
- Support for Jx9 scripting.
- APIs for working with Jx9 JSON document collections.
- Supports both Python 2 and Python 3.

The previous version (0.2.0) of `unqlite-python` utilized `ctypes` to wrap the UnQLite C library. By switching to Cython, key/value, cursor and Jx9 collection operations are an order of magnitude faster. In particular, filtering collections using user-defined Python functions is now *much*, *much* more performant.

The source code for unqlite-python is hosted on GitHub.

Note: If you encounter any bugs in the library, please open an issue, including a description of the bug and any related traceback.

Note: If you like UnQLite, you might also want to check out Vedis, an embedded key/value database modeled after Redis (python bindings: vedis-python).

Contents:

CHAPTER 1

Installation

You can use `pip` to install `unqlite`:

```
pip install cython unqlite
```

The project is hosted at https://github.com/coleifer/unqlite-python and can be installed from source:

```
git clone https://github.com/coleifer/unqlite-python
cd unqlite-python
python setup.py build
python setup.py install
```

Note: `unqlite-python` depends on Cython to generate the Python extension. As of 0.5.0, unqlite-python no longer ships with a generated C source file, so it is necessary to install Cython in order to compile `unqlite-python`.

After installing unqlite-python, you can run the unit tests by executing the *tests* module:

```
python tests.py
```

CHAPTER 2

Quick-start

Below is a sample interactive console session designed to show some of the basic features and functionality of the unqlite-python library. Also check out the full API documentation.

To begin, instantiate an *UnQLite* object. You can specify either the path to a database file, or use UnQLite as an in-memory database.

```
>>> from unqlite import UnQLite
>>> db = UnQLite()  # Create an in-memory database.
```

Key/value features

UnQLite can be used as a key/value store.

```
>>> db['foo'] = 'bar'  # Use as a key/value store.
>>> print db['foo']
bar

>>> for i in range(4):
...     db['k%s' % i] = str(i)
...

>>> 'k3' in db
True
>>> 'k4' in db
False
>>> del db['k3']

>>> db.append('k2', 'XXXX')
>>> db['k2']
'2XXXX'
```

The database can also be iterated through directly:

```
>>> [item for item in db]
[('foo', 'bar'), ('k0', '0'), ('k1', '1'), ('k2', '2XXXX')]
```

UnQLite databases support common `dict` APIs, such as *keys()*, *values()*, *items()*, and *update()*.

Cursors

For finer-grained record traversal, you can use cursors.

```
>>> with db.cursor() as cursor:
...     for key, value in cursor:
...         print key, '=>', value
...
k0 => 0
k1 => 1
k2 => 2XXXX

>>> with db.cursor() as cursor:
...     cursor.seek('k2')
...     print cursor.value()
...
2

>>> with db.cursor() as cursor:
...     cursor.seek('k0')
...     print list(cursor.fetch_until('k2', include_stop_key=False))
...
[('k0', '0'), ('k1', '1')]
```

For more information, see the *Cursor* API documentation.

Document store features

In my opinion the most interesting feature of UnQLite is its JSON document store. The Jx9 scripting language is used to interact with the document store, and it is a wacky mix of C, JavaScript and PHP.

Interacting with the document store basically consists of creating a Jx9 script (you might think of it as an imperative SQL query), compiling it, and then executing it.

```
>>> script = """
...     db_create('users');
...     db_store('users', $list_of_users);
...     $users_from_db = db_fetch_all('users');
... """

>>> list_of_users = [
...     {'username': 'Huey', 'age': 3},
...     {'username': 'Mickey', 'age': 5}
... ]

>>> with db.vm(script) as vm:
...     vm['list_of_users'] = list_of_users
...     vm.execute()
...     users_from_db = vm['users_from_db']
```

```
...
True

>>> users_from_db  # UnQLite assigns items in a collection an ID.
[{'username': 'Huey', 'age': 3, '__id': 0},
 {'username': 'Mickey', 'age': 5, '__id': 1}]
```

This is just a taste of what is possible with Jx9. More information can be found in the *VM* documentation.

Collections

To simplify working with JSON document collections, `unqlite-python` provides a light API for executing Jx9 queries on collections. A collection is an ordered list of JSON objects (records). Records can be appended, updated or deleted. be support for updates as well.

To begin working with *Collection*, you can use the *UnQLite.collection()* factory method:

```
>>> users = db.collection('users')
>>> users.create()  # Create the collection if it does not exist.
>>> users.exists()
True
```

You can use the *Collection.store()* method to add one or many records. To add a single record just pass in a python `dict`. To add multiple records, pass in a list of dicts. Records can be fetched and deleted by ID using *fetch()* and *delete()*.

```
>>> users.store([
...     {'name': 'Charlie', 'color': 'green'},
...     {'name': 'Huey', 'color': 'white'},
...     {'name': 'Mickey', 'color': 'black'}])
True
>>> users.store({'name': 'Leslie', 'color': 'also green'})
True

>>> users.fetch(0)  # Fetch the first record, user with "__id" = 0.
{'__id': 0, 'color': 'green', 'name': 'Charlie'}

>>> users.delete(0)  # Delete the first record (user "__id" = 0).
True
>>> users.delete(users.last_record_id())  # Delete the last record.
True
```

You can retrieve all records in the collection, or specify a filtering function. The filtering function will be registered as a foreign function with the Jx9 VM and called *from* the VM.

```
>>> users.all()
[{'__id': 1, 'color': 'white', 'name': 'Huey'},
 {'__id': 2, 'color': 'black', 'name': 'Mickey'}]

>>> users.filter(lambda obj: obj['name'].startswith('H'))
[{'__id': 1, 'color': 'white', 'name': 'Huey'}]
```

More information can be found in the *Collection* documentation.

Transactions

UnQLite supports transactions for file-backed databases (since transactions occur at the filesystem level, they have no effect on in-memory databases).

The easiest way to create a transaction is with the context manager:

```
>>> db = UnQLite('/tmp/test.db')
>>> with db.transaction():
...     db['k1'] = 'v1'
...     db['k2'] = 'v2'
...
>>> db['k1']
'v1'
```

You can also use the transaction decorator which will wrap a function call in a transaction and commit upon successful execution (rolling back if an exception occurs).

```
>>> @db.commit_on_success
... def save_value(key, value, exc=False):
...     db[key] = value
...     if exc:
...         raise Exception('uh-oh')
...
>>> save_value('k3', 'v3')
>>> save_value('k3', 'vx', True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "unqlite/core.py", line 312, in wrapper
    return fn(*args, **kwargs)
  File "<stdin>", line 5, in save_value
Exception: uh-oh
>>> db['k3']
'v3'
```

For finer-grained control you can call *begin()*, *rollback()* and *commit()* manually.

```
>>> db.begin()
>>> db['k3'] = 'v3-xx'
>>> db.commit()
True
>>> db['k3']
'v3-xx'
```

CHAPTER 3

API Documentation

class UnQLite(*[database=':mem:'[, flags=UNQLITE_OPEN_CREATE[, open_database=True]]*)

The *UnQLite* object provides a pythonic interface for interacting with UnQLite databases. UnQLite is a lightweight, embedded NoSQL database and JSON document store.

Parameters

- **database** (*str*) – The path to the database file.
- **flags** (*int*) – How the database file should be opened.
- **open_database** (*bool*) – When set to `True`, the database will be opened automatically when the class is instantiated. If set to `False` you will need to manually call *open()*.

Note: UnQLite supports in-memory databases, which can be created by passing in `':mem:'` as the database file. This is the default behavior if no database file is specified.

Example usage:

```
>>> db = UnQLite()  # Create an in-memory database.
>>> db['foo'] = 'bar'  # Use as a key/value store.
>>> print db['foo']
bar

>>> for i in range(4):
...     db['k%s' % i] = str(i)
...

>>> 'k3' in db
True
>>> 'k4' in db
False
>>> del db['k3']

>>> db.append('k2', 'XXXX')
>>> db['k2']
```

```
'2XXXX'

>>> with db.cursor() as cursor:
...     for key, value in cursor:
...         print key, '=>', value
...
foo => bar
k0 => 0
k1 => 1
k2 => 2XXXX

>>> script = """
...     db_create('users');
...     db_store('users', $list_of_users);
...     $users_from_db = db_fetch_all('users');
... """

>>> list_of_users = [
...     {'username': 'Huey', 'age': 3},
...     {'username': 'Mickey', 'age': 5}
... ]

>>> with db.vm(script) as vm:
...     vm['list_of_users'] = list_of_users
...     vm.execute()
...     users_from_db = vm['users_from_db']
...
True

>>> users_from_db  # UnQLite assigns items in a collection an ID.
[{'username': 'Huey', 'age': 3, '__id': 0},
 {'username': 'Mickey', 'age': 5, '__id': 1}]
```

open()

Open the database. This method should only be called if the database was manually closed, or if the database was instantiated with `open_database=False`.

Valid flags:

- UNQLITE_OPEN_CREATE
- UNQLITE_OPEN_READONLY
- UNQLITE_OPEN_READWRITE
- UNQLITE_OPEN_CREATE
- UNQLITE_OPEN_EXCLUSIVE
- UNQLITE_OPEN_TEMP_DB
- UNQLITE_OPEN_NOMUTEX
- UNQLITE_OPEN_OMIT_JOURNALING
- UNQLITE_OPEN_IN_MEMORY
- UNQLITE_OPEN_MMAP

Detailed descriptions of these flags can be found in the unqlite_open docs.

close()
Close the database.

> **Warning:** If you are using a file-based database, by default any uncommitted changes will be committed when the database is closed. If you wish to discard uncommitted changes, you can use *disable_autocommit()*.

__enter__()
Use the database as a context manager, opening the connection and closing it at the end of the wrapped block:

```
with UnQLite('my_db.udb') as db:
    db['foo'] = 'bar'

# When the context manager exits, the database is closed.
```

disable_autocommit()
When the database is closed, prevent any uncommitted writes from being saved.

> **Note:** This method only affects file-based databases.

store(*key*, *value*)
Store a value in the given key.

Parameters

- **key** (*str*) – Identifier used for storing data.
- **value** (*str*) – A value to store in UnQLite.

Example:

```
db = UnQLite()
db.store('some key', 'some value')
db.store('another key', 'another value')
```

You can also use the dictionary-style `db[key] = value` to store a value:

```
db['some key'] = 'some value'
```

fetch(*key*)
Retrieve the value stored at the given `key`. If no value exists in the given key, a `KeyError` will be raised.

Parameters **key** (*str*) – Identifier to retrieve

Returns The data stored at the given key

Raises `KeyError` if the given key does not exist.

Example:

```
db = UnQLite()
db.store('some key', 'some value')
value = db.fetch('some key')
```

You can also use the dictionary-style `value = db[key]` lookup to retrieve a value:

```
value = db['some key']
```

delete(*key*)
Remove the key and its associated value from the database.

Parameters **key** (*str*) – The key to remove from the database.

Raises KeyError if the given key does not exist.

Example:

```
def clear_cache():
    db.delete('cached-data')
```

You can also use the python del keyword combined with a dictionary lookup:

```
def clear_cache():
    del db['cached-data']
```

append(*key*, *value*)
Append the given value to the data stored in the key. If no data exists, the operation is equivalent to *store()*.

Parameters

- **key** (*str*) – The identifier of the value to append to.
- **value** – The value to append.

exists(*key*)
Return whether the given key exists in the database.

Parameters **key** (*str*) –

Returns A boolean value indicating whether the given key exists in the database.

Example:

```
def get_expensive_data():
    if not db.exists('cached-data'):
        db.set('cached-data', calculate_expensive_data())
    return db.get('cached-data')
```

You can also use the python in keyword to determine whether a key exists:

```
def get_expensive_data():
    if 'cached-data' not in db:
        db['cached-data'] = calculate_expensive_data()
    return db['cached-data']
```

begin()
Begin a transaction.

rollback()
Roll back the current transaction.

commit()
Commit the current transaction.

transaction()
Create a context manager for performing multiple operations in a transaction.

Warning: Transactions occur at the disk-level and have no effect on in-memory databases.

Example:

```
# Transfer $100 in a transaction.
with db.transaction():
    db['from_acct'] = db['from_account'] - 100
    db['to_acct'] = db['to_acct'] + 100

# Make changes and then roll them back.
with db.transaction():
    db['foo'] = 'bar'
    db.rollback()  # Whoops, do not commit these changes.
```

commit_on_success(*fn*)

Function decorator that will cause the wrapped function to have all statements wrapped in a transaction. If the function returns without an exception, the transaction is committed. If an exception occurs in the function, the transaction is rolled back.

Example:

```
>>> @db.commit_on_success
... def save_value(key, value, exc=False):
...     db[key] = value
...     if exc:
...         raise Exception('uh-oh')
...
>>> save_value('k3', 'v3')
>>> save_value('k3', 'vx', True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "unqlite/core.py", line 312, in wrapper
    return fn()
  File "<stdin>", line 5, in save_value
Exception: uh-oh
>>> db['k3']
'v3'
```

cursor()

Returns a *Cursor* instance.

Create a cursor for traversing database records.

vm(*code*)

Parameters **code** (*str*) – a Jx9 script.

Returns a *VM* instance with the compiled script.

Compile the given Jx9 script and return an initialized *VM* instance.

Usage:

```
script = "$users = db_fetch_all('users');"
with db.vm(script) as vm:
    vm.execute()
    users = vm['users']
```

collection(*name*)

Parameters **name** (*str*) – The name of the collection.

Factory method for instantiating a *Collection* for working with a collection of JSON objects.

Usage:

```
Users = db.collection('users')

# Fetch all records in the collection.
all_users = Users.all()

# Create a new record.
Users.store({'name': 'Charlie', 'activities': ['reading', 'programming']})
```

See the *Collection* docs for more examples.

keys()

Returns A generator that successively yields the keys in the database.

values()

Returns A generator that successively yields the values in the database.

items()

Returns A generator that successively yields tuples containing the keys and values in the database.

update(*data*)

Parameters **data** (*dict*) – Dictionary of data to store in the database. If any keys in `data` already exist, the values will be overwritten.

__iter__()

UnQLite databases can be iterated over. The iterator is a *Cursor*, and will yield 2-tuples of keys and values:

```
db = UnQLite('my_db.udb')
for (key, value) in db:
    print key, '=>', value
```

range(*start_key*, *end_key*[, *include_end_key=True*])

Iterate over a range of key/value pairs in the database.

```
for key, value in db.range('d.20140101', 'd.20140201', False):
    calculate_daily_aggregate(key, value)
```

__len__()

Return the number of records in the database.

> **Warning:** This method calculates the lengthy by iterating and counting every record. At the time of writing, there is no C API for calculating the size of the database.

flush()

Delete all records in the database.

> **Warning:** This method works by iterating through all the records and deleting them one-by-one. At the time of writing there is no API for bulk deletes. If you are worried about speed, simply delete the database file and re-open it.

random_string(*nbytes*)

Parameters **nbytes** (*int*) – number of bytes to generate

Returns a string consisting of random lower-case letters (a-z).

random_number()

Returns a random positive integer

lib_version()

Returns The UnQLite library version.

class **Transaction**(*unqlite*)

Parameters **unqlite** (UnQLite) – An *UnQLite* instance.

Context-manager for executing wrapped blocks in a transaction. Rather than instantiating this object directly, it is recommended that you use *UnQLite.transaction()*.

Example:

```
with db.transaction():
    db['from_acct'] = db['from_acct'] + 100
    db['to_acct'] = db['to_acct'] - 100
```

To roll back changes inside a transaction, call *UnQLite.rollback()*:

```
with db.transaction():
    db['from_acct'] = db['from_acct'] + 100
    db['to_acct'] = db['to_acct'] - 100
    if int(db['to_acct']) < 0:
        db.rollback()  # Not enough funds!
```

class **Cursor**(*unqlite*)

Parameters **unqlite** (UnQLite) – An *UnQLite* instance.

Create a cursor. Cursors should generally be used as context managers.

Rather than instantiating this class directly, it is preferable to call the factory method *UnQLite.cursor()*.

```
for i in range(4):
    db['k%d' % i] = str(i)

# Cursor support iteration, which returns key/value pairs.
with db.cursor() as cursor:
    all_items = [(key, value) for key, value in cursor]

    # You can seek to a record, then iterate to retrieve a portion
    # of results.
    cursor.seek('k2')
    k2, k3 = [key for key, _ in cursor]

# Previous cursor was closed automatically, open a new one.
with db.cursor() as cursor:
```

```
    cursor.seek('k1')  # Jump to the 2nd record, k1
    assert cursor.key() == 'k1'  # Use the key()/value() methods.
    assert cursor.value() == '1'

    cursor.delete()  # Delete k1/v1
    cursor.first()  # Cursor now points to k0/0
    cursor.next()  # Cursor jumps to k2/2 since k1/1 is deleted.
    assert cursor.key() == 'k2'

    keys = [key for key, value in cursor]  # Cursor iterates from k2->k3
    assert keys == ['k2', 'k3']
```

reset()
Reset the cursor, which also resets the pointer to the first record.

seek(*key*[, *flags=UNQLITE_CURSOR_MATCH_EXACT*])
Advance the cursor to the given key using the comparison method described in the flags.

A detailed description of alternate flags and their usage can be found in the unqlite_kv_cursor docs.

Usage:

```
with db.cursor() as cursor:
    cursor.seek('item.20140101')
    while cursor.is_valid():
        data_for_day = cursor.value()
        # do something with data for day
        handle_data(data_for_day)
        if cursor.key() == 'item.20140201':
            break
        else:
            cursor.next()
```

first()
Place cursor at the first record.

last()
Place cursor at the last record.

next_entry()
Move the cursor to the next record.

Raises `StopIteration` if you have gone past the last record.

previous_entry()
Move the cursor to the previous record.

Raises `StopIteration` if you have gone past the first record.

is_valid()

Return type bool

Indicate whether this cursor is pointing to a valid record.

__iter__()
Iterate over the keys in the database, returning 2-tuples of key/value.

Note: Iteration will begin wherever the cursor is currently pointing, rather than starting at the first record.

key()
Return the key of the current record.

value()
Return the value of the current record.

delete()
Delete the record currently pointed to by the cursor.

> **Warning:** The *delete()* method is a little weird in that it only seems to work if you explicitly call *seek()* beforehand.

fetch_until(*stop_key*[, *include_stop_key=True*])

Parameters

- **stop_key** (*str*) – The key at which the cursor should stop iterating.
- **include_stop_key** (*bool*) – Whether the stop key/value pair should be returned.

Yield successive key/value pairs until the `stop_key` is reached. By default the `stop_key` and associated value will be returned, but this behavior can be controlled using the `include_stop_key` flag.

class VM(*unqlite*, *code*)

Parameters

- **unqlite** (`UnQLite`) – An *UnQLite* instance.
- **code** (*str*) – A Jx9 script.

Python wrapper around an UnQLite virtual machine. The VM is the primary means of executing Jx9 scripts and interacting with the JSON document store.

VM instances should not be instantiated directly, but created by calling *UnQLite.vm()*.

Note: For information on Jx9 scripting, see the Jx9 docs.

Example of passing values into a Jx9 script prior to execution, then extracting values afterwards:

```
script = """
    $collection = 'users';
    db_create($collection);
    db_store($collection, $values);
    $users = db_fetch_all($collection);
"""

# We can pass all sorts of interesting data in to our script.
values = [
    {'username': 'huey', 'color': 'white'},
    {'username': 'mickey', 'color': 'black'},
]

with db.vm(script) as vm:
    # Set the value of the `values` variable in the Jx9 script:
    vm['values'] = values

    # Execute the script, which creates the collection and stores
    # the two records.
```

```
    vm.execute()

    # After execution, we can extract the value of the `users` variable.
    users = vm['users']

    # Jx9 document store assigns a unique 0-based id to each record
    # in a collection. The extracted variable `users` will now equal:
    print users == [
        {'username': 'huey', 'color': 'white', '__id': 0},
        {'username': 'mickey', 'color': 'black', '__id': 1},
    ]  # prints `True`
```

execute()
Execute the compiled Jx9 script.

close()
Release the VM, deallocating associated memory.

Note: When using the VM as a context manager, this is handled automatically.

__enter__()
Typically the VM should be used as a context manager. The context manager API handles compiling the Jx9 code and releasing the data-structures afterwards.

```
with db.vm(jx9_script) as vm:
    vm.execute()
```

set_value(*name*, *value*)

Parameters

- **name** (*str*) – A variable name
- **value** – Value to pass in to the scope of the Jx9 script, which should be either a string, int, float, bool, list, dict, or None (basically a valid JSON type).

Set the value of a Jx9 variable. You can also use dictionary-style assignment to set the value.

get_value(*name*)

Parameters **name** (*str*) – A variable name

Retrieve the value of a variable after the execution of a Jx9 script. You can also use dictionary-style lookup to retrieve the value.

compile(*code*)

Parameters **code** (*str*) – A Jx9 script.

Compile the Jx9 script and initialize the VM.

Warning: It is not necessary to call this method yourself, as it is called automatically when the VM is used as a context manager.

Note: This does not execute the code. To execute the code, you must also call *VM.execute()*.

class `Collection` (*unqlite*, *name*)

Parameters

- **unqlite** – a *UnQLite* instance
- **name** (*str*) – the name of the collection

Perform common operations on a JSON document collection.

Note: Rather than instantiating this class directly, use the factory method *UnQLite.collection()*.

Basic operations:

```
>>> users = db.collection('users')
>>> users.create()  # Create the collection if it does not exist.
>>> users.exists()
True

>>> users.store([
...     {'name': 'Charlie', 'color': 'green'},
...     {'name': 'Huey', 'color': 'white'},
...     {'name': 'Mickey', 'color': 'black'}])
True
>>> users.store({'name': 'Leslie', 'color': 'also green'})
True

>>> users.fetch(0)  # Fetch the first record (user "__id" = 0).
{'__id': 0, 'color': 'green', 'name': 'Charlie'}

>>> users.delete(0)  # Delete the first record (user "__id" = 0).
True
>>> users.delete(users.last_record_id())  # Delete the last record.
True

>>> users.update(1, {'color': 'white', 'name': 'Baby Huey'})
True

>>> users.all()
[{'__id': 1, 'color': 'white', 'name': 'Baby Huey'},
 {'__id': 2, 'color': 'black', 'name': 'Mickey'}]

>>> users.filter(lambda obj: obj['name'].startswith('B'))
[{'__id': 1, 'color': 'white', 'name': 'Baby Huey'}]
```

all()

Return a list containing all records in the collection.

filter(*filter_fn*)

Filter the list of records using the provided function (or lambda). Your filter function should accept a single parameter, which will be the record, and return a boolean value indicating whether the record should be returned.

Example:

```
>>> users.filter(lambda user: user['is_admin'] == True)
[{'__id': 0, 'username': 'Huey', 'is_admin': True},
 {'__id': 3, 'username': 'Zaizee', 'is_admin': True},
 {'__id': 4, 'username': 'Charlie', 'is_admin': True}]
```

create()
Create the collection if it does not exist.

drop()
Drop the collection, deleting all records.

exists()

Returns boolean value indicating whether the collection exists.

last_record_id()

Returns The integer ID of the last record stored in the collection.

current_record_id()

Returns The integer ID of the record pointed to by the active cursor.

reset_cursor()
Reset the collection cursor to point to the first record in the collection.

__len__()
Return the number of records in the collection.

fetch(*record_id*)
Return the record with the given id.

```
>>> users = db.collection('users')
>>> users.fetch(0)  # Fetch the first record in collection (id=0).
{'name': 'Charlie', 'color': 'green', '__id': 0}

>>> users[1]  # You can also use dictionary-style lookup.
{'name': 'Huey', 'color': 'white', '__id': 1}
```

You can also use the dictionary API:

```
>>> users[0]
{'name': 'Charlie', 'color': 'green', '__id': 0}
```

store(*record*[, *return_id=True*])

Parameters

- **record** – Either a dictionary (single-record), or a list of dictionaries.
- **return_id** (*bool*) – Return the ID of the newly-created object.

Returns New object's ID, or a boolean indicating if the record was stored successfully.

Store the record(s) in the collection.

```
>>> users = db.collection('users')
>>> users.store({'name': 'Charlie', 'color': 'green'})
True
>>> users.store([
...     {'name': 'Huey', 'color': 'white'},
...     {'name': 'Mickey', 'color': 'black'}])
True
```

update(*record_id*, *record*)

Parameters

- **record_id** – The ID of the record to update.

- **record** – A dictionary of data to update the given record ID.

Returns Boolean value indicating if the update was successful.

Update the data stored for the given `record_id`. The data is completely replaced, rather than being appended to.

```
>>> users = db.collection('users')
>>> users.store({'name': 'Charlie'})
True
>>> users.update(users.last_record_id(), {'name': 'Chuck'})
True
>>> users.fetch(users.last_record_id())
{'__id': 0, 'name': 'Chuck'}
```

delete(*record_id*)

Parameters **record_id** – The database-provided ID of the record to delete.

Returns Boolean indicating if the record was deleted successfully.

Delete the record with the given id.

```
>>> data = db.collection('data')
>>> data.create()
>>> data.store({'foo': 'bar'})
True
>>> data.delete(data.last_record_id())
True
>>> data.all()
[]
```

You can also use the dictionary API:

```
>>> del users[1]  # Delete user object with `__id=1`.
```

fetch_current()

Fetch the record pointed to by the collection cursor.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

I

K

L

N

O

P

R

S

T

U

V