vedis-python Documentation

Release 0.3.1

charles leifer

Jul 20, 2018

Contents

1 Installation **3**

2 Quickstart **5**
- 2.1 Key/value features . . . 5
- 2.2 Transactions . . . 6
- 2.3 Hashes . . . 6
- 2.4 Sets . . . 7
- 2.5 Lists . . . 8
- 2.6 Misc . . . 8

3 API Documentation **11**
- 3.1 Hash objects . . . 26
- 3.2 Set objects . . . 27
- 3.3 List objects . . . 28
- 3.4 Vedis Context . . . 28
- 3.5 Transactions . . . 30

4 Creating Your Own Commands **31**
- 4.1 Valid return types for user-defined commands . . . 32
- 4.2 Operations supported by VedisContext . . . 32

5 Indices and tables **33**

vedis-python

Fast Python bindings for Vedis, an embedded, NoSQL key/value and data-structure store modeled after Redis.

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

Vedis features:

- Embedded, zero-conf database
- Transactional (ACID)
- Single file or in-memory database
- Key/value store
- Over 70 commands similar to standard Redis commands.
- Thread-safe
- Terabyte-sized databases

Vedis-Python features:

- Compiled library, extremely fast with minimal overhead.
- Supports key/value operations and transactions using Pythonic APIs.
- Support for executing Vedis commands.
- Write custom commands in Python.
- Python 2.x and 3.x.

Limitations:

- Not tested on Windoze.

The previous version (0.2.0) of `vedis-python` utilized `ctypes` to wrap the Vedis C library. By switching to Cython, key/value and Vedis command operations are significantly faster.

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 Vedis you might also want to check out UnQLite, an embedded key/value database and JSON document store (python bindings: unqlite-python.

Contents:

CHAPTER 1

Installation

You can use `pip` to install `vedis-python`:

```
pip install cython vedis
```

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

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

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

After installing vedis-python, you can run the unit tests by executing the `tests` module:

```
python tests.py
```

CHAPTER 2

Quickstart

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

2.1 Key/value features

You can use Vedis like a dictionary for simple key/value lookups:

```
>>> from vedis import Vedis
>>> db = Vedis(':mem:')  # Create an in-memory database. Alternatively you could␣
↪supply a filename for an on-disk database.
>>> db['k1'] = 'v1'
>>> db['k1']
'v1'

>>> db.append('k1', 'more data')  # Returns length of value after appending new data.
11
>>> db['k1']
'v1more data'

>>> del db['k1']
>>> db['k1'] is None
True
```

You can set and get multiple items at a time:

```
>>> db.mset(dict(k1='v1', k2='v2', k3='v3'))
True

>>> db.mget(['k1', 'k2', 'missing key', 'k3'])
['v1', 'v2', None, 'v3']
```

In addition to storing string keys/values, you can also implement counters:

```
>>> db.incr('counter')
1

>>> db.incr('counter')
2

>>> db.incr_by('counter', 10)
12

>>> db.decr('counter')
11
```

2.2 Transactions

Vedis has support for transactions when you are using an on-disk database. You can use the *transaction()* context manager or explicitly call *begin()*, *commit()* and *rollback()*.

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

>>> with db.transaction():
...     db['k1'] = 'modified'
...     db.rollback()  # Undo changes.
...
>>> db['k1']  # Value is not modified.
'v1'

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

2.3 Hashes

Vedis supports nested key/value lookups which have the additional benefit of supporting operations to retrieve all keys, values, the number of items in the hash, and so on.

```
>>> h = db.Hash('some key')
>>> h['k1'] = 'v1'
>>> h.update(k2='v2', k3='v3')

>>> h
<Hash: {'k3': 'v3', 'k2': 'v2', 'k1': 'v1'}>

>>> h.to_dict()
```

(continues on next page)

(continued from previous page)

```
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1'}

>>> h.items()
[('k1', 'v1'), ('k3', 'v3'), ('k2', 'v2')]

>>> list(h.keys())
['k1', 'k3', 'k2']

>>> del h['k2']

>>> len(h)
2

>>> 'k1' in h
True

>>> [key for key in h]
['k1', 'k3']
```

2.4 Sets

Vedis supports a set data-type which stores a unique collection of items.

```
>>> s = db.Set('some set')
>>> s.add('v1', 'v2', 'v3')
3

>>> len(s)
3

>>> 'v1' in s, 'v4' in s
(True, False)

>>> s.top()
'v1'

>>> s.peek()
'v3'

>>> s.remove('v2')
1

>>> s.add('v4', 'v5')
2

>>> s.pop()
'v5'

>>> [item for item in s]
['v1', 'v3', 'v4']

>>> s.to_set()
set(['v1', 'v3', 'v4'])
```

(continues on next page)

(continued from previous page)

```
>>> s2 = db.Set('another set')
>>> s2.add('v1', 'v4', 'v5', 'v6')
4

>>> s2 & s  # Intersection.
set(['v1', 'v4'])

>>> s2 - s  # Difference.
set(['v5', 'v6'])
```

2.5 Lists

Vedis also supports a list data type.

```
>>> l = db.List('my list')
>>> l.append('v1')
1
>>> l.extend(['v2', 'v3', 'v1'])
4

>>> for item in l:
...     print item
...
v1
v2
v3
v4

>>> for item in l[1:3]:
...     print item
v2
v3

>>> len(l)
4

>>> l[1]
'v2'

>>> db.llen('my_list')
2

>>> l.pop(), l.pop()
('v1', 'v2')

>>> len(l)
2
```

2.6 Misc

Vedis has a somewhat quirky collection of other miscellaneous commands. Below is a sampling:

```
>>> db.base64('encode me')
'ZW5jb2RlIG1l'

>>> db.base64_decode('ZW5jb2RlIG1l')
'encode me'

>>> db.random_string(10)
'raurquvsnx'

>>> db.rand(1, 6)
4

>>> db.str_split('abcdefghijklmnop', 5)
['abcde', 'fghij', 'klmno', 'p']

>>> db['data'] = 'abcdefghijklmnop'
>>> db.strlen('data')
16

>>> db.strip_tags('<p>This <span>is</span> a <a href="#">test</a>.</p>')
'This is a test.'
```

CHAPTER 3

API Documentation

class Vedis([*filename=':mem:'*[, *open_database=True*]])

The *Vedis* object provides a pythonic interface for interacting with vedis databases. Vedis is a lightweight, embedded NoSQL database modeled after Redis.

Parameters

- **filename** (*str*) – The path to the database file. For in-memory databases, you can either leave this parameter empty or specify the string `:mem:`.
- **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: Vedis 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 = Vedis()  # Create an in-memory database.
>>> db['foo'] = 'bar'  # Use as a key/value store.
>>> print db['foo']
bar

>>> db.update({'k0': 'v0', 'k1': 'v1', 'k2': 'v2', 'k3': 'v3'})

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

>>> db.append('k2', 'XXXX')
>>> db.mget(['k1', 'k2', 'k3'])
['1', '2XXXX', None]
```

(continues on next page)

(continued from previous page)

```
>>> db.incr('counter_1')
1
>>> db.incr_by('counter_1', 10)
11

>>> hash_obj = db.Hash('my-hash')
>>> hash_obj['k1'] = 'v1'
>>> hash_obj.update(k2='v2', k3='v3')
2
>>> hash_obj.to_dict()
{'k1': 'v1', 'k2': 'v2', 'k3': 'v3'}
>>> hash_obj.items()
[('k1', 'v1'), ('k2', 'v2'), ('k3', 'v3')]
>>> [key for key in hash_obj]
['k1', 'k2', 'k3']
>>> len(hash_obj)
3

>>> set_obj = db.Set('my-set')
>>> set_obj.add('foo', 'bar', 'baz', 'foo')
3
>>> 'foo' in set_obj
True
>>> del set_obj['bar']
1
>>> set_obj.to_set()
{'baz', 'foo'}
>>> [item for item in set_obj]
['baz', 'foo']
>>> len(set_obj)
2

>>> list_obj = db.List('my-list')
>>> list_obj.extend(['foo', 'bar', 'baz', 'nug'])
4
>>> list_obj.append('another')
5
>>> [item for item in list_obj]
['foo', 'bar', 'baz', 'nug', 'another']
>>> list_obj[1]
'bar'
>>> list_obj.pop()
'foo'
>>> len(l)
4
```

open()
Open the database connection.

close()
Close the database connection.

> **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 Vedis('my_db.vdb') 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.

---

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

Parameters

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

Example:

```
db = Vedis()
db.set('some key', 'some value')
db.set('another key', 'another value')
```

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

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

get(*key*)
Retrieve the value stored at the given `key`. If no value exists, 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 = Vedis()
db.set('some key', 'some value')
value = db.get('some key')
```

You can also use the dictionary-style `[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 *set()*.

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. Oddly, this only seems to work for simple key/value pairs. If, for instance, you have stored a hash at the given key, `exists` will return `False`.

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']
```

update(*data*)

Parameters **data** (*dict*) – Dictionary of data to store in the database.

Set multiple key/value pairs in a single command, similar to Python's `dict.update()`.

Example:

```
db = Vedis()
db.update(dict(
    hostname=socket.gethostname(),
    user=os.environ['USER'],
    home_dir=os.environ['HOME'],
    path=os.environ['PATH']))
```

mget(*keys*)

Retrieve the values of multiple keys in a single command. In the event a key does not exist, `None` will be returned for that particular value.

Parameters **keys** (*list*) – A list of one or more keys to retrieve.

Returns The values for the given keys.

Example:

```
>>> db.update(dict(k1='v1', k2='v2', k3='v3', k4='v4'))
>>> db.mget(['k1', 'k3', 'missing', 'k4'])
['v1', 'v3', None, 'v4']
```

mset(*data*)

Parameters **data** (*dict*) – Dictionary of data to store in the database.

Set multiple key/value pairs in a single command. This is equivalent to the *update()* method.

setnx(*key*, *value*)

Set the value for the given key *only* if the key does not exist.

Returns `True` if the value was set, `False` if the key already existed.

Example:

```
def create_user(email, password_hash):
    if db.setnx(email, password_hash):
        print 'User added successfully'
        return True
    else:
        print 'Error: username already taken.'
        return False
```

msetnx(*kwargs*)

Similar to *update()*, except that existing keys will not be overwritten.

Returns `True` on success.

Example:

```
>>> db.msetnx({'k1': 'v1', 'k2': 'v2'})
>>> db.mget(['k1', 'k2'])
['v1', 'v2']

>>> db.msetnx({'k1': 'v1x', 'k2': 'v2x', 'k3': 'v3x'})
>>> db.mget(['k1', 'k2', 'k3'])
['v1', 'v2', 'v3x']
```

get_set(*key*, *value*)

Get the value at the given `key` and set it to the new `value` in a single operation.

Returns The original value at the given `key`.

Example:

```
>>> db['k1'] = 'v1'
>>> db.get_set('k1', 'v-x')
'v1'

>>> db['k1']
'v-x'
```

incr(*key*)

Increment the value stored in the given `key` by `1`. If no value exists or the value is not an integer, the counter will be initialized at zero then incremented.

Returns The integer value stored in the given counter.

```
>>> db.incr('my-counter')
1
>>> db.incr('my-counter')
2
```

decr(*key*)
Decrement the value stored in the given `key` by `1`. If no value exists or the value is not an integer, the counter will be initialized at zero then decremented.

Returns The integer value stored in the given counter.

Example:

```
>> db.decr('my-counter')
3
>> db.decr('my-counter')
2
>> db.decr('does-not-exist')
-1
```

incr_by(*key*, *amt*)
Increment the given `key` by the integer `amt`. This method has the same behavior as *incr()*.

decr_by(*key*, *amt*)
Decrement the given `key` by the integer `amt`. This method has the same behavior as *decr()*.

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'
```

Hash(*key*)

Create a *Hash* object, which provides a dictionary-like interface for working with Vedis hashes.

Parameters **key** (*str*) – The key for the Vedis hash object.

Returns a *Hash* object representing the Vedis hash at the specified key.

Example:

```
>>> my_hash = db.Hash('my_hash')
>>> my_hash.update(k1='v1', k2='v2')
>>> my_hash.to_dict()
{'k2': 'v2', 'k1': 'v1'}
```

hset(*hash_key*, *key*, *value*)

Set the value for the key in the Vedis hash identified by `hash_key`.

Example:

```
>>> db.hset('my_hash', 'k3', 'v3')
>>> db.hget('my_hash', 'k3')
'v3'
```

hsetnx(*hash_key*, *key*, *value*)

Set a value for the given key in a Vedis hash only if the key does not already exist. Returns boolean indicating whether the value was successfully set.

Return type bool

Example:

```
>>> db.hsetnx('my_hash', 'kx', 'vx')
True
>>> db.hsetnx('my_hash', 'kx', 'vx')
False
```

hget(*hash_key*, *key*)

Retrieve the value for the key in the Vedis hash identified by `hash_key`.

Returns The value for the given key, or `None` if the key does not exist.

Example:

```
>>> db.hset('my_hash', 'k3', 'v3')
>>> db.hget('my_hash', 'k3')
'v3'
```

hdel (*hash_key*, *key*)
Delete a `key` from a Vedis hash. If the key does not exist in the hash, the operation is a no-op.

Returns The number of keys deleted.

Example:

```
>>> db.hdel('my_hash', 'k3')
1
>>> db.hget('my_hash', 'k3') is None
True
```

hkeys (*hash_key*)
Get the keys for the Vedis hash identified by `hash_key`.

Returns All keys for the Vedis hash.

Example:

```
>>> db.hkeys('my_hash')
['k2', 'k1']
```

hvals (*hash_key*)
Get the values for the Vedis hash identified by `hash_key`.

Returns All values for the Vedis hash.

Example:

```
>>> db.hvals('my_hash')
['v2', 'v1']
```

hgetall (*hash_key*)
Return a `dict` containing all items in the Vedis hash identified by `hash_key`.

Returns A dictionary containing the key/value pairs stored in the given Vedis hash, or an empty `dict` if a hash does not exist at the given key.

Return type dict

Example:

```
>>> db.hgetall('my_hash')
{'k2': 'v2', 'k1': 'v1'}

>>> db.hgetall('does not exist')
{}
```

hitems (*hash_key*)
Get a list to key/value pairs stored in the given Vedis hash.

Returns A list of key/value pairs stored in the given Vedis hash, or an empty list if a hash does not exist at the given key.

Return type list of 2-tuples

Example:

```
>>> db.hitems('my_hash')
[('k2', 'v2'), ('k1', 'v1')]
```

hlen(*hash_key*)

Return the number of items stored in a Vedis hash. If a hash does not exist at the given key, `0` will be returned.

Return type int

Example:

```
>>> db.hlen('my_hash')
2
>>> db.hlen('does not exist')
0
```

hexists(*hash_key*, *key*)

Return whether the given key is stored in a Vedis hash. If a hash does not exist at the given key, `False` will be returned.

Return type bool

Example:

```
>>> db.hexists('my_hash', 'k1')
True
>>> db.hexists('my_hash', 'kx')
False
>>> db.hexists('does not exist', 'kx')
False
```

hmset(*hash_key*, *data*)

Set multiple key/value pairs in the given Vedis hash. This method is analagous to Python's `dict.update`.

Example:

```
>>> db.hmset('my_hash', {'k1': 'v1', 'k2': 'v2', 'k3': 'v3', 'k4': 'v4'})
>>> db.hgetall('my_hash')
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1', 'k4': 'v4'}
```

hmget(*hash_key*, *keys*)

Return the values for multiple keys in a Vedis hash. If the key does not exist in the given hash, `None` will be returned for the missing key.

Example:

```
>>> db.hmget('my_hash', ['k1', 'k4', 'missing', 'k2'])
['v1', 'v4', None, 'v2']
```

hmdel(*hash_key*, *keys*)

Delete multiple keys from a Vedis hash.

Returns The number of keys actually deleted.

Example:

```
>>> db.hmdel('my_hash', ['k1', 'k2', 'invalid-key'])
2
```

Set(*key*)
Create a *Set* object, which provides a set-like interface for working with Vedis sets.

Parameters **key** (*str*) – The key for the Vedis set object.

Returns a *Set* object representing the Vedis set at the specified key.

Example:

```
>>> my_set = db.Set('my_set')
>>> my_set.add('v1', 'v2', 'v3')
3
>>> my_set.to_set()
set(['v1', 'v2', 'v3'])
```

sadd(*key*, *value*)
Add a single value to a Vedis set, returning the number of items added.

Example:

```
>>> db.sadd('my_set', 'v1')
1
>>> db.sadd('my_set', 'v2')
1
>>> db.smembers('my_set')
{'v1', 'v2'}
```

smadd(*key*, *values*)
Add one or more values to a Vedis set, returning the number of items added.

Unlike *sadd()*, smadd accepts a list of values to add to the set.

Example:

```
>>> db.smadd('my_set', ['v1', 'v2', 'v3'])
>>> db.smembers('my_set')
{'v1', 'v2', 'v3'}
```

scard(*key*)
Return the cardinality, or number of items, in the given set. If a Vedis set does not exist at the given key, 0 will be returned.

Example:

```
>>> db.scard('my_set')
3
>>> db.scard('does not exist')
0
```

sismember(*key*, *value*)
Return a boolean indicating whether the provided value is a member of a Vedis set. If a Vedis set does not exist at the given key, False will be returned.

Example:

```
>>> db.sismember('my_set', 'v1')
True
>>> db.sismember('my_set', 'vx')
False
>>> print db.sismember('does not exist', 'xx')
False
```

spop(*key*)
Remove and return the last record from a Vedis set. If a Vedis set does not exist at the given key, or the set is empty, `None` will be returned.

Example:

```
>>> db.sadd('my_set', 'v1', 'v2', 'v3')
3
>>> db.spop('my_set')
'v3'
```

speek(*key*)
Return the last record from a Vedis set without removing it. If a Vedis set does not exist at the given key, or the set is empty, `None` will be returned.

Example:

```
>>> db.sadd('my_set', 'v1', 'v2', 'v3')
3
>>> db.speek('my_set')
'v3'
```

stop(*key*)
Return the first record from a Vedis set without removing it.

Example:

```
>>> db.sadd('my_set', 'v1', 'v2', 'v3')
>>> db.stop('my_set')
'v1'
```

srem(*key*, *value*)
Remove the given value from a Vedis set.

Returns The number of items removed.

Example:

```
>>> db.sadd('my_set', 'v1', 'v2', 'v3')
3
>>> db.srem('my_set', 'v2')
1
>>> db.srem('my_set', 'v2')
0
>>> list(db.smembers('my_set'))
['v1', 'v3']
```

smrem(*key*, *values*)
Remove one or more values from the Vedis set.

Returns The number of items removed.

Example:

```
>>> db.smadd('my_set', ['v1', 'v2', 'v3'])
3
>>> db.smrem('my_set', ['v1', 'v2', 'xxx'])
>>> db.smembers('my_set')
{'v3'}
```

smembers(*key*)
Return all members of a given set.

Return type set

Example:

```
>>> db.smembers('my_set')
{'v1', 'v3'}
```

sdiff(*k1*, *k2*)
Return the set difference of two Vedis sets identified by `k1` and `k2`.

Example:

```
>>> db.sadd('my_set', 'v1', 'v2', 'v3')
3
>>> db.sadd('other_set', 'v2', 'v3', 'v4')
3
>>> db.sdiff('my_set', 'other_set')
{'v1'}
```

sinter(*k1*, *k2*)
Return the intersection of two Vedis sets identified by `k1` and `k2`.

Example:

```
>>> db.sadd('my_set', 'v1', 'v2', 'v3')
3
>>> db.sadd('other_set', 'v2', 'v3', 'v4')
3
>>> db.sinter('my_set', 'other_set')
{'v3', 'v2'}
```

List(*key*)
Create a *List* object, which provides a list-like interface for working with Vedis lists.

Parameters **key** (*str*) – The key for the Vedis list object.

Returns a *List* object representing the Vedis list at the specified key.

Example:

```
>>> my_list = db.List('my_list')
>>> my_list.append('i1')
>>> my_list.extend(['i2', 'i3'])
>>> my_list[0]
'i1'
>>> my_list.pop()
'i1'
>>> len(my_list)
2
```

lpush(*key*, *value*)
Append one value to a Vedis list, returning the number of items added.

Example:

```
>>> db.lpush('my_list', 'i1')
1
```

lmpush(*key*, *values*)
Append one or more values to a Vedis list, returning the number of items added.

Example:

```
>>> db.lmpush('my_list', ['i2', 'i3', 'i4'])
3
```

lindex(*key*, *idx*)
Returns the element at the given index in the Vedis list. Indices are zero-based, and negative indices can be used to designate elements starting from the end of the list.

Example:

```
>>> db.lmpush('my_list', ['i1', 'i2', 'i3'])
>>> db.lindex('my_list', 0)
'i1'
>>> db.lindex('my_list', -1)
'i3'
```

llen(*key*)
Return the length of a Vedis list.

Example:

```
>>> db.llen('my_list')
3
>>> db.llen('does not exist')
0
```

lpop(*key*)
Remove and return the first element of a Vedis list. If no elements exist, `None` is returned.

Example:

```
>>> db.lmpush('a list', ['i1', 'i2'])
2
>>> db.lpop('a list')
'i1'
```

register(*command_name*)
Function decorator used to register user-defined Vedis commands. User-defined commands must accept a special *VedisContext* as their first parameter, followed by any number of parameters specified when the command was invoked. The following are valid return types for user-defined commands:

- lists (arbitrarily nested)
- strings
- boolean values
- integers
- floating point numbers
- `None`

Here is a simple example of a custom command that converts its arguments to title-case:

```
@db.register('TITLE')
def title_cmd(vedis_ctx, *params):
    return [param.title() for param in params]
```

Here is how you might call your user-defined function:

```
>>> db.execute('TITLE %s %s %s', ('foo', 'this is a test', 'bar'))
['Foo', 'This Is A Test', 'Bar']
```

You can also call the wrapped function directly, and the call will be routed through Vedis:

```
>>> title('foo', 'this is a test', 'bar')
['Foo', 'This Is A Test', 'Bar']
```

For more information, see the custom_commands section.

delete_command(*command_name*)
Unregister a custom command.

strlen(*key*)
Return the length of the value stored at the given key.

Example:

```
>>> db = Vedis()
>>> db['foo'] = 'testing'
>>> db.strlen('foo')
7
```

copy(*src*, *dest*)
Copy the contents of one key to another, leaving the original intact.

move(*src*, *dest*)
Move the contents of one key to another, deleting the original key.

strip_tags(*html*)
Remove HTML formatting from a given string.

Parameters **html** (*str*) – A string containing HTML.

Returns A string with all HTML removed.

Example:

```
>>> db.strip_tags('<p>This <span>is</span> <a href="#">a <b>test</b></a>.</p>
↪')
'This is a test.'
```

str_split(*s*[, *nchars=1*])
Split the given string, `s`.

Example:

```
>>> db.str_split('abcdefghijklmnop', 5)
['abcde', 'fghij', 'klmno', 'p']
```

size_format(*nbytes*)
Return a user-friendly representation of a given number of bytes.

Example:

```
>>> db.size_format(1337)
'1.3 KB'
>>> db.size_format(1337000)
'1.2 MB'
```

soundex(*s*)
Calculate the `soundex` value for a given string.

Example:

```
>>> db.soundex('howdy')
'H300'
>>> db.soundex('huey')
'H000'
```

base64(*data*)
Encode `data` in base64.

Example:

```
>>> db.base64('hello')
'aGVsbG8='
```

base64_decode(*data*)
Decode the base64-encoded `data`.

Example:

```
>>> db.base64_decode('aGVsbG8=')
'hello'
```

rand(*lower_bound*, *upper_bound*)
Return a random integer within the lower and upper bounds (inclusive).

randstr(*nbytes*)
Return a random string of `nbytes` length, made up of the characters a-z.

time()
Return the current GMT time, formatted as HH:MM:SS.

date()
Return the current date in ISO-8601 format (YYYY-MM-DD).

operating_system()
Return a brief description of the host operating system.

table_list()
Return a list of all vedis tables (i.e. Hashes, Sets, List) in memory.

execute(*cmd*[, *params=None*[, *result=True*]])
Execute a Vedis command.

Parameters

- **cmd** (*str*) – The command to execute.
- **params** (*tuple*) – A tuple of parameters to pass into the command.
- **result** (*bool*) – Return the result of this command.

Example:

```
db = Vedis()

# Execute a command, ignoring the result.
db.execute('HSET %s %s %s', ('hash_key', 'key', 'some value'))
```

(continues on next page)

(continued from previous page)

```
# Execute a command that returns a single result.
val = db.execute('HGET %s %s', ('hash_key', 'key'))

# Execute a command return returns multiple values.
keys = db.execute('HKEYS %s', ('hash_key',))
for key in keys:
    print 'Hash "hash_key" contains key "%s"' % key
```

3.1 Hash objects

class Hash(*vedis*, *key*)

Provides a high-level API for working with Vedis hashes. As much as seemed sensible, the *Hash* acts like a python dictionary.

Note: This class should not be constructed directly, but through the factory method *Vedis.Hash()*.

Here is an example of how you might use the various `Hash` APIs:

```
>>> h = db.Hash('my_hash')

>>> h['k1'] = 'v1'
>>> h.update(k2='v2', k3='v3')

>>> len(h)
3

>>> 'k1' in h
True
>>> 'k4' in h
False

>>> h.to_dict()
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1'}

>>> h.keys()
['k1', 'k3', 'k2']
>>> h.values()
['v1', 'v3', 'v2']
>>> h.items()
[('k1', 'v1'), ('k3', 'v3'), ('k2', 'v2')]

>>> del h['k2']
>>> h.items()
[('k1', 'v1'), ('k3', 'v3')]

>>> h.mget('k3', 'kx', 'k1')
['v3', None, 'v1']

>>> h
<Hash: {'k3': 'v3', 'k1': 'v1'}>
```

3.2 Set objects

class Set (*vedis*, *key*)

Provides a high-level API for working with Vedis sets. As much as seemed sensible, the *Set* acts like a python set.

Note: This class should not be constructed directly, but through the factory method *Vedis.Set()*.

Here is an example of how you might use the various `Set` APIs:

```
>>> s = db.Set('my_set')

>>> s.add('v1', 'v2', 'v1', 'v3')
4
>>> len(s)
3

>>> [item for item in s]
['v1', 'v2', 'v3']

>>> s.top()
'v1'
>>> s.peek()
'v3'
>>> s.pop()
'v3'

>>> 'v2' in s
True
>>> 'v3' in s
False

>>> s.add('v3', 'v4')
2
>>> del s['v4']
>>> s.to_set()
{'v1', 'v2', 'v3'}
```

Vedis also supports set difference and intersection:

```
>>> s2 = db.Set('other_set')
>>> s2.add('v3', 'v4', 'v5')
3

>>> s - s2
{'v1', 'v2'}

>>> s2 - s
{'v4', 'v5'}

>>> s & s2
{'v3'}
```

3.3 List objects

class List(*vedis*, *key*)

Provides a high-level API for working with Vedis lists.

Note: This class should not be constructed directly, but through the factory method *Vedis.List()*.

Here is an example of how you might use the various `List` APIs:

```
>>> l = db.List('my_list')

>>> l.append('v1')
1
>>> l.extend(['v2', 'v3', 'v4'])
4

>>> len(l)
4

>>> l[0]
'v1'
>>> l[-1]
'v4'

>>> [item for item in l]
['v1', 'v2', 'v3', 'v4']

>>> l.pop()
'v1'
```

3.4 Vedis Context

When a user-defined command is executed, the first parameter sent to the callback is a `vedis_context` instance. The `vedis_context` allows user-defined commands to set return codes (handled automatically by vedis-python), but perhaps more interestingly, modify other keys and values in the database.

In this way, your user-defined command can set, get, and delete keys in the vedis database. Because the vedis_context APIs are a bit low-level, vedis-python wraps the `vedis_context`, providing a nicer API to work with.

class VedisContext(*vedis_context*)

This class will almost never be instantiated directly, but will instead by created by vedis-python when executing a user-defined callback.

Parameters **vedis_context** – A pointer to a `vedis_context`.

Usage:

```
@db.register('TITLE_VALUES')
def title_values(context, *values):
    """
    Create key/value pairs for each value consisting of the
    original value -> the title-cased version of the value.

    Returns the number of values processed.
```

(continues on next page)

(continued from previous page)

```
    """
    for value in values:
        context[value] = value.title()
    return len(values)
```

```
>>> db.execute('TITLE_VALUES %s %s', ('val 1', 'another value'))
2
>>> db['val 1']
'Val 1'
>>> db['another val']
'Another Val'
```

fetch(*key*)
Return the value of the given key. Identical to *Vedis.get()*.

Instead of calling `fetch()` you can also use a dictionary-style lookup on the context:

```
@db.register('MY_COMMAND')
def my_command(context, *values):
    some_val = context['the key']
    # ...
```

store(*key*, *value*)
Set the value of the given key. Identical to *Vedis.set()*.

Instead of calling `store()` you can also use a dictionary-style assignment on the context:

```
@db.register('MY_COMMAND')
def my_command(context, *values):
    context['some key'] = 'some value'
    # ...
```

append(*key*, *value*)
Append a value to the given key. If the key does not exist, the operation is equivalent to *store()*. Identical to *Vedis.append()*.

delete(*key*)
Delete the given key. Identical to *Vedis.delete()*.

Instead of calling `delete()` you can also use a the python `del` keyword:

```
@db.register('MY_COMMAND')
def my_command(context, *values):
    del context['some key']
    # ...
```

exists(*key*)
Check for the existence of the given key. Identical to *Vedis.exists()*.

Instead of calling `exists()` you can also use a the python `in` keyword:

```
@db.register('MY_COMMAND')
def my_command(context, *values):
    if 'some key' in context:
        # ...
```

3.5 Transactions

class Transaction(*vedis*)

Parameters **vedis** (Vedis) – An *Vedis* instance.

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

CHAPTER 4

Creating Your Own Commands

It is possible to create your own Vedis commands and execute them like any other. Use the *Vedis.register()* method to decorate the function you wish to turn into a Vedis command. Your command callback must accept at least one argument, the *VedisContext* (which wraps vedis context). Any arguments supplied by the caller will also be passed to your callback. Using the *VedisContext* object, your function can perform key/value operations on the database.

Here is a small example:

```
db = Vedis()

@db.register('CONCAT')
def concat(context, glue, *params):
    return glue.join(params)

@db.register('TITLE')
def title(context, *params):
    return [param.title() for param in params]

@db.register('HASH_VALUES')
def hash_values(context, *values):
    # Calculate a hash for each value and store it in a
    # key.
    for value in values:
        context[value] = hashlib.sha1(value).hexdigest()
    return len(values)
```

Usage:

```
>>> print db.execute('CONCAT | foo bar baz')
foo|bar|baz

>>> print db.execute('TITLE "testing" "this is a test" "another"')
['Testing', 'This Is A Test', 'Another']
```

(continues on next page)

(continued from previous page)

```
>>> print db.execute('HASH_VALUES shh secret')
2
>>> db.mget(['shh', 'secret'])
['0c731a5f1dc781894b434c27b9f6a9cd9d9bdfcb',
 'e5e9fa1ba31ecd1ae84f75caaa474f3a663f05f4']
```

You can also directly call the function with your arguments, and the call will automatically be routed through Vedis:

```
>>> print title('testing', 'this is a test', 'another')
['Testing', 'This Is A Test', 'Another']

>>> print concat('#', 'foo', '1', 'hello')
'foo#1#hello'
```

4.1 Valid return types for user-defined commands

- `list` or `tuple` (containing arbitrary levels of nesting).
- `str`
- `int` and `long`
- `float`
- `bool`
- `None`

4.2 Operations supported by VedisContext

The first parameter of your custom command is always a *`VedisContext`* instance. This object can be used to access the key/value features of the database. It supports the following APIs:

- Getting, setting and deleting items using `dict` APIs.
- Checking whether a key exists using `in`.
- Appending to an existing key.

Example:

```
@db.register('STORE_DATA')
def store_data(context):
    context['foo'] = 'bar'
    assert context['foo'] == 'bar'
    del context['other key']
    assert 'foo' in context
    context.append('foo', 'more data')
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

Symbols

__enter__() (Vedis method), 13

A

append() (Vedis method), 14
append() (VedisContext method), 29

B

base64() (Vedis method), 25
base64_decode() (Vedis method), 25
begin() (Vedis method), 16

C

close() (Vedis method), 12
commit() (Vedis method), 16
commit_on_success() (Vedis method), 16
copy() (Vedis method), 24

D

date() (Vedis method), 25
decr() (Vedis method), 16
decr_by() (Vedis method), 16
delete() (Vedis method), 13
delete() (VedisContext method), 29
delete_command() (Vedis method), 24
disable_autocommit() (Vedis method), 13

E

execute() (Vedis method), 25
exists() (Vedis method), 14
exists() (VedisContext method), 29

F

fetch() (VedisContext method), 29

G

get() (Vedis method), 13
get_set() (Vedis method), 15

H

Hash (built-in class), 26
Hash() (Vedis method), 17
hdel() (Vedis method), 18
hexists() (Vedis method), 19
hget() (Vedis method), 17
hgetall() (Vedis method), 18
hitems() (Vedis method), 18
hkeys() (Vedis method), 18
hlen() (Vedis method), 19
hmdel() (Vedis method), 19
hmget() (Vedis method), 19
hmset() (Vedis method), 19
hset() (Vedis method), 17
hsetnx() (Vedis method), 17
hvals() (Vedis method), 18

I

incr() (Vedis method), 15
incr_by() (Vedis method), 16

L

lindex() (Vedis method), 23
List (built-in class), 28
List() (Vedis method), 22
llen() (Vedis method), 23
lmpush() (Vedis method), 22
lpop() (Vedis method), 23
lpush() (Vedis method), 22

M

mget() (Vedis method), 14
move() (Vedis method), 24
mset() (Vedis method), 15
msetnx() (Vedis method), 15

O

open() (Vedis method), 12
operating_system() (Vedis method), 25

R

rand() (Vedis method), 25
randstr() (Vedis method), 25
register() (Vedis method), 23
rollback() (Vedis method), 16

S

sadd() (Vedis method), 20
scard() (Vedis method), 20
sdiff() (Vedis method), 22
Set (built-in class), 27
Set() (Vedis method), 19
set() (Vedis method), 13
setnx() (Vedis method), 15
sinter() (Vedis method), 22
sismember() (Vedis method), 20
size_format() (Vedis method), 24
smadd() (Vedis method), 20
smembers() (Vedis method), 21
smrem() (Vedis method), 21
soundex() (Vedis method), 24
speek() (Vedis method), 21
spop() (Vedis method), 20
srem() (Vedis method), 21
stop() (Vedis method), 21
store() (VedisContext method), 29
str_split() (Vedis method), 24
strip_tags() (Vedis method), 24
strlen() (Vedis method), 24

T

table_list() (Vedis method), 25
time() (Vedis method), 25
Transaction (built-in class), 30
transaction() (Vedis method), 16

U

update() (Vedis method), 14

V

Vedis (built-in class), 11
VedisContext (built-in class), 28