wheezy.caching documentation

Release latest

Andriy Kornatskyy

Jun 22, 2021

Contents

CHAPTER 1

Introduction

wheezy.caching is a python package written in pure Python code. It is a lightweight caching library that provides integration with:

- python-memcached - Pure Python memcached client.
- pylibmc - Quick and small memcached client for Python written in C.

It introduces the idea of *cache dependency* (effectively invalidate dependent cache items) and other cache related algorithms.

It is optimized for performance, well tested and documented.

Resources:

- source code and issues tracker are available on github
- documentation

CHAPTER 2

Contents

2.1 Getting Started

2.1.1 Install

wheezy.caching requires python version 3.6+. It is operating system independent. You can install it from the pypi site:

```
$ pip install wheezy.caching
```

2.2 Examples

We start with a simple example. Before we proceed let's setup a virtualenv environment, activate it and install:

```
$ pip install install wheezy.caching
```

2.2.1 Playing Around

We are going create a number of items, add them to cache, try to get them back, establish dependency and finally invalidate all together:

```
from wheezy.caching import MemoryCache as Cache
from wheezy.caching import CacheDependency

cache = Cache()

# Add a single item
cache.add('k1', 1)

# Add few more
```

(continues on next page)

(continued from previous page)

```
cache.add_multi({'k2': 2, 'k3': 3})

# Get a single item
cache.get('k2')

# Get several at once
cache.get_multi(['k1', 'k2', 'k3'])

# Establish dependency somewhere in code place A
dependency = CacheDependency('master-key')
dependency.add(cache, 'k1')

# Establish dependency somewhere in code place B
dependency.add_multi(cache, ['k1', 'k2', 'k3'])

# Invalidate dependency somewhere in code place C
dependency.delete(cache)
```

2.3 User Guide

wheezy.caching comes with the following cache implementations:

- CacheClient
- MemoryCache
- NullCache

wheezy.caching provides integration with:

- python-memcached - Pure Python memcached client.
- pylibmc - Quick and small memcached client for Python written in C.

It introduces the idea of *cache dependency* that lets you effectively invalidate dependent cache items.

2.3.1 Contract

All cache implementations and integrations provide the same contract. That means caches can be swapped without a need to modify the code. However there does exist a challenge: some caches are singletons and correctly provide inter-thread synchronization (thread safe), while others require an instance per thread (not thread safe), for which some sort of pooling is required. This challenge is transparently resolved.

Here is an example how to configure pylibmc - memcached (client written in C):

```
from wheezy.core.pooling import EagerPool
from wheezy.caching.pylibmc import MemcachedClient
from wheezy.caching.pylibmc import client_factory

# Cache Pool
pool = EagerPool(lambda: client_factory(['/tmp/memcached.sock']), size=10)
# Factory
cache = MemcachedClient(pool)

# Client code
cache.set(...)
```

The client code remains unchanged even some cache implementations require pooling to remain thread safe.

2.3.2 CacheClient

CacheClient serves as mediator between a single entry point that implements Cache and one or many namespaces targeted to cache factories.

CacheClient lets us partition application cache by namespaces, effectively hiding details from client code.

CacheClient accepts the following arguments:

- `namespaces` - a mapping between namespace and cache factory.
- `default_namespace` - namespace to use in case it is not specified in cache operation.

In the example below we partition application cache into three (default, membership and funds):

```
from wheezy.caching import ClientCache
from wheezy.caching import MemoryCache
from wheezy.caching import NullCache

default_cache = MemoryCache()
membership_cache = MemoryCache()
funds_cache = NullCache()
cache = ClientCache({
    'default': default_cache,
    'membership': membership_cache,
    'funds': funds_cache,
}, default_namespace='default')
```

Application code is designed to work with a single cache by specifying namespace to use:

```
cache.add('x1', 1, namespace='default')
```

At some point of time we might change our partitioning scheme so all namespaces reside in a single cache:

```
default_cache = MemoryCache()
cachey = ClientCache({
    'default': default_cache,
    'membership': default_cache,
    'funds': default_cache
}, default_namespace='default')
```

What happened with no changes to application code? These are just configuration settings.

2.3.3 MemoryCache

MemoryCache is an effective, high performance in-memory cache implementation. There is no background routine to invalidate expired items in the cache, instead they are checked on each get operation.

In order to effectively manage invalidation of expired items (those that are not actively requested) each item being added to cache is assigned to a time bucket. Each time bucket has a number associated with a point in time. So if incoming store operation relates to time bucket N, all items from that bucket are being checked and expired items removed.

You control a number of buckets during initialization of *MemoryCache*. Here are attributes that are accepted:

- `buckets` - a number of buckets present in cache (defaults to 60).

- `bucket_interval` - what is interval in seconds between time buckets (defaults to 15).

Interval set by `bucket_interval` shows how often items in cache will be checked for expiration. So if it set to 15 means that every 15 seconds cache will choose a bucket related to that point in time and all items in bucket will be checked for expiration. Since there are 60 buckets in the cache that means only 1/60 part of cache items are locked. This lock does not impact items requested by `get`/`get_multi` operations. Taking into account this lock happens only once per 15 seconds it cause minor impact on overall cache performance.

2.3.4 NullCache

`NullCache` is a cache implementation that actually does not do anything but silently performs cache operations that result in no change to state.

- `get`, `get_multi` operations always report miss.
- `set`, `add`, etc (all store operations) always succeed.

2.3.5 python-memcached

python-memcached is a pure Python memcached client. You can install this package via pip:

```
$ pip install python-memcached
```

Here is a typical use case:

```
from wheezy.caching.memcache import MemcachedClient

cache = MemcachedClient(['unix:/tmp/memcached.sock'])
```

You can specify a key encoding function by passing a `key_encode` argument that must be a callable that does key encoding. By default *`string_encode()`* is applied.

All arguments passed to *`MemcachedClient()`* are the same as those passed to the original `Client` from python-memcache. Note, python-memcached `Client` implementation is *thread local* object.

2.3.6 pylibmc

pylibmc is a quick and small memcached client for Python written in C. Since this package is an interface to *libmemcached*, you need the development version of this library installed so pylibmc can be compiled. If you are using Debian:

```
apt-get install libmemcached-dev
```

Now, you can install this package via pip:

```
$ pip install pylibmc
```

Here is a typical use case:

```
from wheezy.core.pooling import EagerPool
from wheezy.caching.pylibmc import MemcachedClient
from wheezy.caching.pylibmc import client_factory

pool = EagerPool(lambda: client_factory(['/tmp/memcached.sock']), size=10)
cache = MemcachedClient(pool)
```

You can specify a key encoding function by passing a `key_encode` argument that must be a callable that does key encoding. By default *`string_encode()`* is applied.

All arguments passed to `client_factory()` are the same as those passed to the original `Client` from pylibmc. Default client factory configures pylibmc Client to use binary protocol, tcp_nodelay and ketama algorithm.

Since pylibmc implementation is not thread safe it requires pooling, as we do here. `EagerPool` holds a number of pylibmc instances.

2.3.7 Key Encoding

Memcached has some restrictions concerning the keys used. Text protocol requires a valid key that contains only ASCII characters except space (0x20), carriage return (0x0d), and line feed (0x0a), since these characters are meaningful in text protocol. Key length is restricted to 250.

- *`string_encode()`* - encodes `key` with UTF-8 encoding.
- *`base64_encode()`* - encodes `key` with base64 encoding.
- *`hash_encode()`* - encodes `key` with given hash function. See list of available hashes in `hashlib` module from the Python Statndard Library. Additional algorithms may also be available depending upon the OpenSSL library that Python uses on your platform.

There is a general purpose function:

- *`encode_keys()`* - encodes all keys in mapping with `key_encode` callable. Returns a tuple of: *key mapping* (encoded key => key) and *value mapping* (encoded key => value).

You can specify the key encoding function to use, by passing the `key_encode` argument to *memcache* and/or *pylibmc* cache factory.

2.3.8 CacheDependency

`CacheDependency` introduces a *wire* between cache items so they can be invalidated via a single operation, thus simplifying code necessary to manage dependencies in cache.

`CacheDependency` is not related to any particular cache implementation.

`CacheDependency` can be used to invalidate items across different cache partitions (namespaces). Note that `delete` must be performed for each namespace and/or cache.

Master Key

It is important to avoid key collisions for the master key due to the way in which dependency keys are built. The dependency keys are built by adding a suffix with incremental number to the master key, e.g. if master key is 'key' than dependent keys used by CacheDependency will be 'key1', 'key2', 'key3', etc. The master key stores the number of dependent keys thus this number is incremented each time you add something to a dependency.

If a master key is composed as a concatenation with some id it must be suffixed with a delimiter (a symbol that is not part of the id) to avoid key collision. In the example below id is a number so choosing ':' as a delimiter suites our needs:

```
def master_key_order(id):
    return 'mk:order:' + str(id) + ':'
```

For order id 100 the master key is 'mk:order:100:' and dependent keys take space 'mk:order:100:1' for the first item added, 'mk:order:100:2' for the second, etc. If we add 2 items to cache dependency the value stored by the master key is 2.

Example

Let's demostrate this by example. We establish dependency between keys `k1`, `k2` and `k3` for 600 seconds. Please note that dependency does not need to be passed between various parts of application. You can create it in one place, than in other, etc. `CacheDependency` stores it state in cache:

```
# this is sample from module a.
dependency = CacheDependency('master-key', time=600)
dependency.add_multi(cache, ['k1', 'k2', 'k3'])

# this is sample from module b.
dependency = CacheDependency('master-key', time=600)
dependency.add(cache, 'k4')
```

Note that module *b* has no idea about keys used in module *a*. Instead they share a cache dependency *virtually*.

Once we need to invalidate items related to cache dependencies, this is what we do:

```
dependency = CacheDependency('master-key')
dependency.delete(cache)
```

`delete` operation must be repeated for each namespace (it doesn't manage namespace dependency) and/or cache:

```
# Using namespaces
dependency = CacheDependency('master-key')
dependency.delete(cache, namespace='membership')
dependency.delete(cache, namespace='funds')

# Using caches
dependency = CacheDependency('master-key')
dependency.delete(membership_cache)
dependency.delete(funds_cache)
```

Cache dependency is an effective way to reduce coupling between modules in terms of cache item invalidation.

2.4 Modules

2.4.1 wheezy.caching

class wheezy.caching.**CacheClient**(*namespaces*, *default_namespace*)

CacheClient serves mediator purpose between a single entry point that implements Cache and one or many namespaces targeted to concrete cache implementations.

CacheClient let partition application cache by namespaces effectively hiding details from client code.

add(*key*, *value*, *time=0*, *namespace=None*)

Sets a key's value, if and only if the item is not already.

add_multi(*mapping*, *time=0*, *namespace=None*)

Adds multiple values at once, with no effect for keys already in cache.

decr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)

Atomically decrements a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then decremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

delete(*key*, *seconds=0*, *namespace=None*)
Deletes a key from cache.

delete_multi(*keys*, *seconds=0*, *namespace=None*)
Delete multiple keys at once.

flush_all()
Deletes everything in cache.

get(*key*, *namespace=None*)
Looks up a single key.

get_multi(*keys*, *namespace=None*)
Looks up multiple keys from cache in one operation. This is the recommended way to do bulk loads.

incr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically increments a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then incremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

replace(*key*, *value*, *time=0*, *namespace=None*)
Replaces a key's value, failing if item isn't already.

replace_multi(*mapping*, *time=0*, *namespace=None*)
Replaces multiple values at once, with no effect for keys not in cache.

set(*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, regardless of previous contents in cache.

set_multi(*mapping*, *time=0*, *namespace=None*)
Set multiple keys' values at once.

class wheezy.caching.**CacheDependency**(*cache*, *time=0*, *namespace=None*)
CacheDependency introduces a *wire* between cache items so they can be invalidated via a single operation, thus simplifing code necessary to manage dependencies in cache.

add(*master_key*, *key*)
Adds a given *key* to dependency.

add_multi(*master_key*, *keys*)
Adds several *keys* to dependency.

delete(*master_key*)
Delete all items wired by *master_key* cache dependency.

delete_multi(*master_keys*)
Delete all items wired by *master_keys* cache dependencies.

get_keys(*master_key*)
Returns all keys wired by *master_key* cache dependency.

get_multi_keys(*master_keys*)
Returns all keys wired by *master_keys* cache dependencies.

next_key(*master_key*)
Returns the next unique key for dependency.

master_key - a key used to track a number of issued dependencies.

next_keys(*master_key*, *n*)
Returns *n* number of dependency keys.

master_key - a key used to track a number of issued dependencies.

class wheezy.caching.**MemoryCache**(*buckets=60*, *bucket_interval=15*)

Effectively implements in-memory cache.

add(*key*, *value*, *time=0*, *namespace=None*)

Sets a key's value, if and only if the item is not already.

```
>>> c = MemoryCache()
>>> c.add('k', 'v', 100)
True
>>> c.add('k', 'v', 100)
False
```

add_multi(*mapping*, *time=0*, *namespace=None*)

Adds multiple values at once, with no effect for keys already in cache.

```
>>> c = MemoryCache()
>>> c.add_multi({'k': 'v'}, 100)
[]
>>> c.add_multi({'k': 'v'}, 100)
['k']
```

decr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)

Atomically decrements a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then decremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

```
>>> c = MemoryCache()
>>> c.decr('k')
>>> c.decr('k', initial_value=10)
9
>>> c.decr('k')
8
```

delete(*key*, *seconds=0*, *namespace=None*)

Deletes a key from cache.

If `key` is not found return False

```
>>> c = MemoryCache()
>>> c.delete('k')
False
>>> c.store('k', 'v', 100)
True
>>> c.delete('k')
True
```

There is item in cache that expired

```
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.delete('k')
False
```

delete_multi(*keys*, *seconds=0*, *namespace=None*)

Delete multiple keys at once.

```
>>> c = MemoryCache()
>>> c.delete_multi(('k1', 'k2', 'k3'))
True
>>> c.store_multi({'k1':1, 'k2': 2}, 100)
[]
>>> c.delete_multi(('k1', 'k2'))
True
```

There is item in cached that expired

```
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.get_multi(('k', ))
{}
```

flush_all()
Deletes everything in cache.

```
>>> c = MemoryCache()
>>> c.set_multi({'k1': 1, 'k2': 2}, 100)
[]
>>> c.flush_all()
True
```

get(*key*, *namespace=None*)
Looks up a single key.

If `key` is not found return None

```
>>> c = MemoryCache()
>>> c.get('k')
```

Otherwise return value

```
>>> c.set('k', 'v', 100)
True
>>> c.get('k')
'v'
```

There is item in cached that expired

```
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.get('k')
```

get_multi(*keys*, *namespace=None*)
Looks up multiple keys from cache in one operation. This is the recommended way to do bulk loads.

```
>>> c = MemoryCache()
>>> c.get_multi(('k1', 'k2', 'k3'))
{}
>>> c.store('k1', 'v1', 100)
True
>>> c.store('k2', 'v2', 100)
True
>>> sorted(c.get_multi(('k1', 'k2')).items())
[('k1', 'v1'), ('k2', 'v2')]
```

There is item in cache that expired

```
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.get_multi(('k', ))
{}
```

incr (*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically increments a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then incremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

```
>>> c = MemoryCache()
>>> c.incr('k')
>>> c.incr('k', initial_value=0)
1
>>> c.incr('k')
2
```

There is item in cached that expired

```
>>> c.items['k'] = CacheItem('k', 1, 1)
>>> c.incr('k')
```

replace (*key*, *value*, *time=0*, *namespace=None*)
Replaces a key's value, failing if item isn't already.

```
>>> c = MemoryCache()
>>> c.replace('k', 'v', 100)
False
>>> c.add('k', 'v', 100)
True
>>> c.replace('k', 'v', 100)
True
```

replace_multi (*mapping*, *time=0*, *namespace=None*)
Replaces multiple values at once, with no effect for keys not in cache.

```
>>> c = MemoryCache()
>>> c.replace_multi({'k': 'v'}, 100)
['k']
>>> c.add_multi({'k': 'v'}, 100)
[]
>>> c.replace_multi({'k': 'v'}, 100)
[]
```

set (*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, regardless of previous contents in cache.

```
>>> c = MemoryCache()
>>> c.set('k', 'v', 100)
True
```

set_multi (*mapping*, *time=0*, *namespace=None*)
Set multiple keys' values at once.

```
>>> c = MemoryCache()
>>> c.set_multi({'k1': 1, 'k2': 2}, 100)
[]
```

store(*key*, *value*, *time=0*, *op=0*)

There is item in cached that expired

```
>>> c = MemoryCache()
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.store('k', 'v', 100)
True
```

There is item in expire_buckets that expired

```
>>> c = MemoryCache()
>>> i = int((int(unixtime()) % c.period)
...         / c.interval) - 1
>>> c.expire_buckets[i] = (allocate_lock(), [('x', 10)])
>>> c.store('k', 'v', 100)
True
```

store_multi(*mapping*, *time=0*, *op=0*)

There is item in cached that expired

```
>>> c = MemoryCache()
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.store_multi({'k': 'v'}, 100)
[]
```

There is item in expire_buckets that expired

```
>>> c = MemoryCache()
>>> i = int((int(unixtime()) % c.period)
...         / c.interval) - 1
>>> c.expire_buckets[i] = (allocate_lock(), [('x', 10)])
>>> c.store_multi({'k': 'v'}, 100)
[]
```

class wheezy.caching.**NullCache**

NullCache is a cache implementation that actually doesn't do anything but silently performs cache operations that result no change to state.

add(*key*, *value*, *time=0*, *namespace=None*)

Sets a key's value, if and only if the item is not already.

```
>>> c = NullCache()
>>> c.add('k', 'v')
True
```

add_multi(*mapping*, *time=0*, *namespace=None*)

Adds multiple values at once, with no effect for keys already in cache.

```
>>> c = NullCache()
>>> c.add_multi({})
[]
```

decr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)

Atomically decrements a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then decremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

```
>>> c = NullCache()
>>> c.decr('k')
```

delete (*key*, *seconds=0*, *namespace=None*)
Deletes a key from cache.

```
>>> c = NullCache()
>>> c.delete('k')
True
```

delete_multi (*keys*, *seconds=0*, *namespace=None*)
Delete multiple keys at once.

```
>>> c = NullCache()
>>> c.delete_multi([])
True
```

flush_all ()
Deletes everything in cache.

```
>>> c = NullCache()
>>> c.flush_all()
True
```

get (*key*, *namespace=None*)
Looks up a single key.

```
>>> c = NullCache()
>>> c.get('k')
```

get_multi (*keys*, *namespace=None*)
Looks up multiple keys from cache in one operation. This is the recommended way to do bulk loads.

```
>>> c = NullCache()
>>> c.get_multi([])
{}
```

incr (*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically increments a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then incremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

```
>>> c = NullCache()
>>> c.incr('k')
```

replace (*key*, *value*, *time=0*, *namespace=None*)
Replaces a key's value, failing if item isn't already.

```
>>> c = NullCache()
>>> c.replace('k', 'v')
True
```

replace_multi(*mapping*, *time=0*, *namespace=None*)
Replaces multiple values at once, with no effect for keys not in cache.

```
>>> c = NullCache()
>>> c.replace_multi({})
[]
```

set(*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, regardless of previous contents in cache.

```
>>> c = NullCache()
>>> c.set('k', 'v')
True
```

set_multi(*mapping*, *time=0*, *namespace=None*)
Set multiple keys' values at once.

```
>>> c = NullCache()
>>> c.set_multi({})
[]
```

2.4.2 wheezy.caching.client

`client` module.

class wheezy.caching.client.**CacheClient**(*namespaces*, *default_namespace*)
CacheClient serves mediator purpose between a single entry point that implements Cache and one or many namespaces targeted to concrete cache implementations.

CacheClient let partition application cache by namespaces effectively hiding details from client code.

add(*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, if and only if the item is not already.

add_multi(*mapping*, *time=0*, *namespace=None*)
Adds multiple values at once, with no effect for keys already in cache.

decr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically decrements a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then decremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

delete(*key*, *seconds=0*, *namespace=None*)
Deletes a key from cache.

delete_multi(*keys*, *seconds=0*, *namespace=None*)
Delete multiple keys at once.

flush_all()
Deletes everything in cache.

get(*key*, *namespace=None*)
Looks up a single key.

get_multi(*keys*, *namespace=None*)
Looks up multiple keys from cache in one operation. This is the recommended way to do bulk loads.

incr (*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically increments a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then incremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

replace (*key*, *value*, *time=0*, *namespace=None*)
Replaces a key's value, failing if item isn't already.

replace_multi (*mapping*, *time=0*, *namespace=None*)
Replaces multiple values at once, with no effect for keys not in cache.

set (*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, regardless of previous contents in cache.

set_multi (*mapping*, *time=0*, *namespace=None*)
Set multiple keys' values at once.

2.4.3 wheezy.caching.dependency

`dependency` module.

class `wheezy.caching.dependency.`**CacheDependency** (*cache*, *time=0*, *namespace=None*)
CacheDependency introduces a *wire* between cache items so they can be invalidated via a single operation, thus simplifing code necessary to manage dependencies in cache.

add (*master_key*, *key*)
Adds a given *key* to dependency.

add_multi (*master_key*, *keys*)
Adds several *keys* to dependency.

delete (*master_key*)
Delete all items wired by *master_key* cache dependency.

delete_multi (*master_keys*)
Delete all items wired by *master_keys* cache dependencies.

get_keys (*master_key*)
Returns all keys wired by *master_key* cache dependency.

get_multi_keys (*master_keys*)
Returns all keys wired by *master_keys* cache dependencies.

next_key (*master_key*)
Returns the next unique key for dependency.

master_key - a key used to track a number of issued dependencies.

next_keys (*master_key*, *n*)
Returns *n* number of dependency keys.

master_key - a key used to track a number of issued dependencies.

2.4.4 wheezy.caching.encoding

`encoding` module.

`wheezy.caching.encoding.`**base64_encode** (*key*)
Encodes `key` with base64 encoding.

```
>>> result = base64_encode('my key')
>>> result == 'bXkga2V5'.encode('latin1')
True
```

wheezy.caching.encoding.**encode_keys**(*mapping*, *key_encode*)

Encodes all keys in mapping with `key_encode` callable. Returns tuple of: key mapping (encoded key => key) and value mapping (encoded key => value).

```
>>> mapping = {'k1': 1, 'k2': 2}
>>> keys, mapping = encode_keys(mapping,
...         lambda k: str(base64_encode(k).decode('latin1')))
>>> sorted(keys.items())
[('azE=', 'k1'), ('azI=', 'k2')]
>>> sorted(mapping.items())
[('azE=', 1), ('azI=', 2)]
```

wheezy.caching.encoding.**hash_encode**(*hash_factory*)

Encodes `key` with given hash function.

See list of available hashes in `hashlib` module from Python Statndard Library.

Additional algorithms may also be available depending upon the OpenSSL library that Python uses on your platform.

```
>>> try:
...     from hashlib import sha1
...     key_encode = hash_encode(sha1)
...     r = base64_encode(key_encode('my key'))
...     assert r == 'RigVwkWdSuGyFu7au08PzUMloU8='.encode('latin1')
... except ImportError:  # Python2.4
...     pass
```

wheezy.caching.encoding.**string_encode**(*key*)

Encodes `key` with UTF-8 encoding.

2.4.5 wheezy.caching.lockout

`lockout` module.

class wheezy.caching.lockout.**Counter**(*key_func*, *count*, *period*, *duration*, *reset=True*, *alert=None*)

A container of various attributes used by lockout.

class wheezy.caching.lockout.**Locker**(*cache*, *forbid_action*, *namespace=None*, *key_prefix='c'*, ***terms*)

Used to define lockout terms.

define(*name*, ***terms*)

Defines a new lockout with given *name* and *terms*. The *terms* keys must correspond to *known terms* of locker.

class wheezy.caching.lockout.**Lockout**(*name*, *counters*, *forbid_action*, *cache*, *namespace*, *key_prefix*)

A lockout is used to enforce terms of use policy.

forbid_locked(*wrapped=None*, *action=None*)

A decorator that forbids access (by a call to *forbid_action*) to *func* once the counter threshold is reached (lock is set).

You can override default forbid action by *action*.

See *test_lockout.py* for an example.

force_reset(*ctx*)
Removes locks for all counters.

guard(*func*)
A guard decorator is applied to a *func* which returns a boolean indicating success or failure. Each failure is a subject to increase counter. The counters that support *reset* (and related locks) are deleted on success.

incr(*ctx*)
Increments lockout counters for given context.

quota(*func*)
A quota decorator is applied to a *func* which returns a boolean indicating success or failure. Each success is a subject to increase counter.

reset(*ctx*)
Removes locks for counters that support reset.

class wheezy.caching.lockout.**NullLocker**(*cache*, *forbid_action*, *namespace=None*, *key_prefix='c'*, ***terms*)
Null locker implementation.

class wheezy.caching.lockout.**NullLockout**
Null lockout implementation.

2.4.6 wheezy.caching.logging

logging module.

class wheezy.caching.logging.**OnePassHandler**(*inner*, *cache*, *time*, *key_encode=None*, *namespace=None*)
One pass logging handler is used to proxy a message to inner handler once per one pass duration.

emit(*record*)
Emit a record. Use log record message as a key in cache.

2.4.7 wheezy.caching.memcache

`memcache` module.

class wheezy.caching.memcache.**MemcachedClient**(**args*, ***kwargs*)
A wrapper around python-memcache Client in order to adapt cache contract.

add(*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, if and only if the item is not already.

add_multi(*mapping*, *time=0*, *namespace=None*)
Adds multiple values at once, with no effect for keys already in cache.

decr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically decrements a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then decremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

delete(*key*, *seconds=0*, *namespace=None*)
Deletes a key from cache.

delete_multi (*keys*, *seconds=0*, *namespace=None*)
Delete multiple keys at once.

flush_all ()
Deletes everything in cache.

get (*key*, *namespace=None*)
Looks up a single key.

get_multi (*keys*, *namespace=None*)
Looks up multiple keys from cache in one operation. This is the recommended way to do bulk loads.

incr (*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically increments a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then incremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

replace (*key*, *value*, *time=0*, *namespace=None*)
Replaces a key's value, failing if item isn't already.

replace_multi (*mapping*, *time=0*, *namespace=None*)
Replaces multiple values at once, with no effect for keys not in cache.

set (*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, regardless of previous contents in cache.

set_multi (*mapping*, *time=0*, *namespace=None*)
Set multiple keys' values at once.

2.4.8 wheezy.caching.memory

`memory` module.

class wheezy.caching.memory.**CacheItem** (*key*, *value*, *expires*)
A single cache item stored in cache.

class wheezy.caching.memory.**MemoryCache** (*buckets=60*, *bucket_interval=15*)
Effectively implements in-memory cache.

add (*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, if and only if the item is not already.

```
>>> c = MemoryCache()
>>> c.add('k', 'v', 100)
True
>>> c.add('k', 'v', 100)
False
```

add_multi (*mapping*, *time=0*, *namespace=None*)
Adds multiple values at once, with no effect for keys already in cache.

```
>>> c = MemoryCache()
>>> c.add_multi({'k': 'v'}, 100)
[]
>>> c.add_multi({'k': 'v'}, 100)
['k']
```

decr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)

Atomically decrements a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then decremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

```
>>> c = MemoryCache()
>>> c.decr('k')
>>> c.decr('k', initial_value=10)
9
>>> c.decr('k')
8
```

delete(*key*, *seconds=0*, *namespace=None*)

Deletes a key from cache.

If `key` is not found return False

```
>>> c = MemoryCache()
>>> c.delete('k')
False
>>> c.store('k', 'v', 100)
True
>>> c.delete('k')
True
```

There is item in cache that expired

```
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.delete('k')
False
```

delete_multi(*keys*, *seconds=0*, *namespace=None*)

Delete multiple keys at once.

```
>>> c = MemoryCache()
>>> c.delete_multi(('k1', 'k2', 'k3'))
True
>>> c.store_multi({'k1':1, 'k2': 2}, 100)
[]
>>> c.delete_multi(('k1', 'k2'))
True
```

There is item in cached that expired

```
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.get_multi(('k', ))
{}
```

flush_all()

Deletes everything in cache.

```
>>> c = MemoryCache()
>>> c.set_multi({'k1': 1, 'k2': 2}, 100)
[]
>>> c.flush_all()
True
```

get(*key*, *namespace=None*)
Looks up a single key.

If `key` is not found return None

```
>>> c = MemoryCache()
>>> c.get('k')
```

Otherwise return value

```
>>> c.set('k', 'v', 100)
True
>>> c.get('k')
'v'
```

There is item in cached that expired

```
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.get('k')
```

get_multi(*keys*, *namespace=None*)
Looks up multiple keys from cache in one operation. This is the recommended way to do bulk loads.

```
>>> c = MemoryCache()
>>> c.get_multi(('k1', 'k2', 'k3'))
{}
>>> c.store('k1', 'v1', 100)
True
>>> c.store('k2', 'v2', 100)
True
>>> sorted(c.get_multi(('k1', 'k2')).items())
[('k1', 'v1'), ('k2', 'v2')]
```

There is item in cache that expired

```
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.get_multi(('k', ))
{}
```

incr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically increments a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then incremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

```
>>> c = MemoryCache()
>>> c.incr('k')
>>> c.incr('k', initial_value=0)
1
>>> c.incr('k')
2
```

There is item in cached that expired

```
>>> c.items['k'] = CacheItem('k', 1, 1)
>>> c.incr('k')
```

replace (*key*, *value*, *time=0*, *namespace=None*)
Replaces a key's value, failing if item isn't already.

```
>>> c = MemoryCache()
>>> c.replace('k', 'v', 100)
False
>>> c.add('k', 'v', 100)
True
>>> c.replace('k', 'v', 100)
True
```

replace_multi (*mapping*, *time=0*, *namespace=None*)
Replaces multiple values at once, with no effect for keys not in cache.

```
>>> c = MemoryCache()
>>> c.replace_multi({'k': 'v'}, 100)
['k']
>>> c.add_multi({'k': 'v'}, 100)
[]
>>> c.replace_multi({'k': 'v'}, 100)
[]
```

set (*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, regardless of previous contents in cache.

```
>>> c = MemoryCache()
>>> c.set('k', 'v', 100)
True
```

set_multi (*mapping*, *time=0*, *namespace=None*)
Set multiple keys' values at once.

```
>>> c = MemoryCache()
>>> c.set_multi({'k1': 1, 'k2': 2}, 100)
[]
```

store (*key*, *value*, *time=0*, *op=0*)
There is item in cached that expired

```
>>> c = MemoryCache()
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.store('k', 'v', 100)
True
```

There is item in expire_buckets that expired

```
>>> c = MemoryCache()
>>> i = int((int(unixtime()) % c.period)
...         / c.interval) - 1
>>> c.expire_buckets[i] = (allocate_lock(), [('x', 10)])
>>> c.store('k', 'v', 100)
True
```

store_multi (*mapping*, *time=0*, *op=0*)
There is item in cached that expired

```
>>> c = MemoryCache()
>>> c.items['k'] = CacheItem('k', 'v', 1)
>>> c.store_multi({'k': 'v'}, 100)
[]
```

There is item in expire_buckets that expired

```
>>> c = MemoryCache()
>>> i = int((int(unixtime()) % c.period)
...         / c.interval) - 1
>>> c.expire_buckets[i] = (allocate_lock(), [('x', 10)])
>>> c.store_multi({'k': 'v'}, 100)
[]
```

wheezy.caching.memory.**expires**(*now*, *time*)

time is below 1 month

```
>>> expires(10, 1)
11
```

more than month

```
>>> expires(10, 3000000)
3000000
```

otherwise

```
>>> expires(0, 0)
2147483647
>>> expires(0, -1)
2147483647
```

wheezy.caching.memory.**find_expired**(*bucket_items*, *now*)

If there are no expired items in the bucket returns empty list

```
>>> bucket_items = [('k1', 1), ('k2', 2), ('k3', 3)]
>>> find_expired(bucket_items, 0)
[]
>>> bucket_items
[('k1', 1), ('k2', 2), ('k3', 3)]
```

Expired items are returned in the list and deleted from the bucket

```
>>> find_expired(bucket_items, 2)
['k1']
>>> bucket_items
[('k2', 2), ('k3', 3)]
```

2.4.9 wheezy.caching.null

interface module.

class wheezy.caching.null.**NullCache**

NullCache is a cache implementation that actually doesn't do anything but silently performs cache operations that result no change to state.

add(*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, if and only if the item is not already.

```
>>> c = NullCache()
>>> c.add('k', 'v')
True
```

add_multi(*mapping*, *time=0*, *namespace=None*)
Adds multiple values at once, with no effect for keys already in cache.

```
>>> c = NullCache()
>>> c.add_multi({})
[]
```

decr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically decrements a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then decremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

```
>>> c = NullCache()
>>> c.decr('k')
```

delete(*key*, *seconds=0*, *namespace=None*)
Deletes a key from cache.

```
>>> c = NullCache()
>>> c.delete('k')
True
```

delete_multi(*keys*, *seconds=0*, *namespace=None*)
Delete multiple keys at once.

```
>>> c = NullCache()
>>> c.delete_multi([])
True
```

flush_all()
Deletes everything in cache.

```
>>> c = NullCache()
>>> c.flush_all()
True
```

get(*key*, *namespace=None*)
Looks up a single key.

```
>>> c = NullCache()
>>> c.get('k')
```

get_multi(*keys*, *namespace=None*)
Looks up multiple keys from cache in one operation. This is the recommended way to do bulk loads.

```
>>> c = NullCache()
>>> c.get_multi([])
{}
```

incr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically increments a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then incremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

```
>>> c = NullCache()
>>> c.incr('k')
```

replace(*key*, *value*, *time=0*, *namespace=None*)
Replaces a key's value, failing if item isn't already.

```
>>> c = NullCache()
>>> c.replace('k', 'v')
True
```

replace_multi(*mapping*, *time=0*, *namespace=None*)
Replaces multiple values at once, with no effect for keys not in cache.

```
>>> c = NullCache()
>>> c.replace_multi({})
[]
```

set(*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, regardless of previous contents in cache.

```
>>> c = NullCache()
>>> c.set('k', 'v')
True
```

set_multi(*mapping*, *time=0*, *namespace=None*)
Set multiple keys' values at once.

```
>>> c = NullCache()
>>> c.set_multi({})
[]
```

2.4.10 wheezy.caching.patterns

`patterns` module.

class wheezy.caching.patterns.**Cached**(*cache*, *key_builder=None*, *time=0*, *namespace=None*, *timeout=10*, *key_prefix='one_pass:'*)
Specializes access to cache by using a number of common settings for various cache operations and patterns.

add(*key*, *value*, *dependency_key=None*)
Sets a key's value, if and only if the item is not already.

add_multi(*mapping*)
Adds multiple values at once, with no effect for keys already in cache.

decr(*key*, *delta=1*, *initial_value=None*)
Atomically decrements a key's value.

delete(*key*, *seconds=0*)
Deletes a key from cache.

delete_multi(*keys*, *seconds=0*)
Delete multiple keys at once.

get(*key*)
Looks up a single key.

get_multi(*keys*)
Looks up multiple keys from cache in one operation. This is the recommended way to do bulk loads.

get_or_add(*key*, *create_factory*, *dependency_key_factory*)
Cache Pattern: get an item by *key* from *cache* and if it is not available use *create_factory* to aquire one. If result is not *None* use cache *add* operation to store result and if operation succeed use *dependency_key_factory* to get an instance of *dependency_key* to link with *key*.

get_or_create(*key*, *create_factory*, *dependency_key_factory=None*)
Cache Pattern: get an item by *key* from *cache* and if it is not available see *one_pass_create*.

get_or_set(*key*, *create_factory*, *dependency_key_factory=None*)
Cache Pattern: get an item by *key* from *cache* and if it is not available use *create_factory* to aquire one. If result is not *None* use cache *set* operation to store result and use *dependency_key_factory* to get an instance of *dependency_key* to link with *key*.

get_or_set_multi(*make_key*, *create_factory*, *args*)
Cache Pattern: *get_multi* items by *make_key* over *args* from *cache* and if there are any missing use *create_factory* to aquire them, if result available use cache *set_multi* operation to store results, return cached items if any.

incr(*key*, *delta=1*, *initial_value=None*)
Atomically increments a key's value.

one_pass_create(*key*, *create_factory*, *dependency_key_factory=None*)
Cache Pattern: try enter one pass: (1) if entered use *create_factory* to get a value if result is not *None* use cache *set* operation to store result and use *dependency_key_factory* to get an instance of *dependency_key* to link with *key*; (2) if not entered *wait* until one pass is available and it is not timed out get an item by *key* from *cache*.

replace(*key*, *value*)
Replaces a key's value, failing if item isn't already.

replace_multi(*mapping*)
Replaces multiple values at once, with no effect for keys not in cache.

set(*key*, *value*, *dependency_key=None*)
Sets a key's value, regardless of previous contents in cache.

set_multi(*mapping*)
Set multiple keys' values at once.

wraps_get_or_add(*wrapped=None*, *make_key=None*)
Returns specialized decorator for *get_or_add* cache pattern.

Example:

```
kb = key_builder('repo')
cached = Cached(cache, kb, time=60)

@cached.wraps_get_or_add
def list_items(self, locale):
    pass
```

wraps_get_or_create(*wrapped=None*, *make_key=None*)
Returns specialized decorator for *get_or_create* cache pattern.

Example:

```
kb = key_builder('repo')
cached = Cached(cache, kb, time=60)

@cached.wraps_get_or_create
def list_items(self, locale):
    pass
```

wraps_get_or_set(*wrapped=None*, *make_key=None*)

Returns specialized decorator for *get_or_set* cache pattern.

Example:

```
kb = key_builder('repo')
cached = Cached(cache, kb, time=60)

@cached
# or @cached.wraps_get_or_set
def list_items(self, locale):
    pass
```

wraps_get_or_set_multi(*make_key*)

Returns specialized decorator for *get_or_set_multi* cache pattern.

Example:

```
cached = Cached(cache, kb, time=60)

@cached.wraps_get_or_set_multi(
    make_key=lambda i: 'key:%r' % i)
def get_multi_account(account_ids):
    pass
```

class wheezy.caching.patterns.**OnePass**(*cache*, *key*, *time=10*, *namespace=None*)

A solution to *Thundering Head* problem.

see http://en.wikipedia.org/wiki/Thundering_herd_problem

Typical use:

```
with OnePass(cache, 'op:' + key) as one_pass:
    if one_pass.acquired:
        # update *key* in cache
    elif one_pass.wait():
        # obtain *key* from cache
    else:
        # timeout
```

wait(*timeout=None*)

Wait *timeout* seconds for the one pass become available.

timeout - if not passed defaults to *time* used during initialization.

wheezy.caching.patterns.**key_builder**(*key_prefix=''*)

Returns a key builder that allows build a make cache key function at runtime.

```
>>> def list_items(self, locale='en', sort_order=1):
...     pass
```

```
>>> repo_key_builder = key_builder('repo')
>>> make_key = repo_key_builder(list_items)
>>> make_key('self')
"repo-list_items:'en':1"
>>> make_key('self', 'uk')
"repo-list_items:'uk':1"
>>> make_key('self', sort_order=0)
"repo-list_items:'en':0"
```

Here is an example of make key function:

```
def key_list_items(self, locale='en', sort_order=1):
    return "repo-list_items:%r:%r" % (locale, sort_order)
```

wheezy.caching.patterns.**key_format**(*func*, *key_prefix*)

Returns a key format for *func* and *key_prefix*.

```
>>> def list_items(self, locale='en', sort_order=1):
...     pass
>>> key_format(list_items, 'repo')
'repo-list_items:%r:%r'
```

wheezy.caching.patterns.**key_formatter**(*key_prefix*)

Specialize a key format with *key_prefix*.

```
>>> def list_items(self, locale='en', sort_order=1):
...     pass
>>> repo_key_format = key_formatter('repo')
>>> repo_key_format(list_items)
'repo-list_items:%r:%r'
```

2.4.11 wheezy.caching.pylibmc

pylibmc module.

class wheezy.caching.pylibmc.**MemcachedClient**(*pool*, *key_encode=None*)

A wrapper around pylibmc Client in order to adapt cache contract.

add(*key*, *value*, *time=0*, *namespace=None*)

Sets a key's value, if and only if the item is not already.

add_multi(*mapping*, *time=0*, *namespace=None*)

Adds multiple values at once, with no effect for keys already in cache.

decr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)

Atomically decrements a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then decremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

delete(*key*, *seconds=0*, *namespace=None*)

Deletes a key from cache.

delete_multi(*keys*, *seconds=0*, *namespace=None*)

Delete multiple keys at once.

flush_all()
Deletes everything in cache.

get(*key*, *namespace=None*)
Looks up a single key.

get_multi(*keys*, *namespace=None*)
Looks up multiple keys from cache in one operation. This is the recommended way to do bulk loads.

incr(*key*, *delta=1*, *namespace=None*, *initial_value=None*)
Atomically increments a key's value. The value, if too large, will wrap around.

If the key does not yet exist in the cache and you specify an initial_value, the key's value will be set to this initial value and then incremented. If the key does not exist and no initial_value is specified, the key's value will not be set.

replace(*key*, *value*, *time=0*, *namespace=None*)
Replaces a key's value, failing if item isn't already.

replace_multi(*mapping*, *time=0*, *namespace=None*)
Replaces multiple values at once, with no effect for keys not in cache.

set(*key*, *value*, *time=0*, *namespace=None*)
Sets a key's value, regardless of previous contents in cache.

set_multi(*mapping*, *time=0*, *namespace=None*)
Set multiple keys' values at once.

2.4.12 wheezy.caching.utils

`utils` module.

`wheezy.caching.utils.`**total_seconds**(*delta*)
Returns a total number of seconds for the given delta.

`delta` can be `datetime.timedelta`.

```
>>> total_seconds(timedelta(hours=2))
7200
```

or int:

```
>>> total_seconds(100)
100
```

otherwise raise `TypeError`.

```
>>> total_seconds('100') # doctest: +ELLIPSIS
Traceback (most recent call last):
    ...
TypeError: ...
```

Python Module Index

W

Index

E

F

G

H

I

K

L

M

N

O

Q

R

S

T

W