

---

# **pymemcache Documentation**

***Release 3.2.0***

**Charles Gordon, Jon Parise, Joe Gordon**

**May 19, 2020**



# CONTENTS

<b>1</b>	<b>Getting started!</b>	<b>3</b>
1.1	Basic Usage . . . . .	3
1.2	Using UNIX domain sockets . . . . .	3
1.3	Using a client pool . . . . .	3
1.4	Using a memcached cluster . . . . .	4
1.5	Using TLS . . . . .	4
1.6	Serialization . . . . .	4
1.7	Deserialization with Python 3 . . . . .	5
1.8	Key Constraints . . . . .	5
1.9	Best Practices . . . . .	6
<b>2</b>	<b>pymemcache</b>	<b>7</b>
2.1	pymemcache package . . . . .	7
<b>3</b>	<b>Indices and tables</b>	<b>23</b>
	<b>Python Module Index</b>	<b>25</b>
	<b>Index</b>	<b>27</b>



Contents:



## GETTING STARTED!

A comprehensive, fast, pure-Python memcached client library.

### 1.1 Basic Usage

```
from pymemcache.client.base import Client

client = Client(('localhost', 11211))
client.set('some_key', 'some_value')
result = client.get('some_key')
```

### 1.2 Using UNIX domain sockets

You can also connect to a local memcached server over a UNIX domain socket by passing the socket's path to the client's server parameter:

```
from pymemcache.client.base import Client

client = Client('/var/run/memcached/memcached.sock')
```

### 1.3 Using a client pool

*pymemcache.client.base.PooledClient* is a thread-safe client pool that provides the same API as *pymemcache.client.base.Client*. It's useful in for cases when you want to maintain a pool of already-connected clients for improved performance.

```
from pymemcache.client.base import PooledClient

client = PooledClient(('127.0.0.1', 11211), max_pool_size=4)
```

## 1.4 Using a memcached cluster

This will use a consistent hashing algorithm to choose which server to set/get the values from. It will also automatically rebalance depending on if a server goes down.

```
from pymemcache.client.hash import HashClient

client = HashClient([
    ('127.0.0.1', 11211),
    ('127.0.0.1', 11212)
])
client.set('some_key', 'some value')
result = client.get('some_key')
```

## 1.5 Using TLS

Memcached supports authentication and encryption via TLS since version 1.5.13.

A Memcached server running with TLS enabled will only accept TLS connections.

To enable TLS in pymemcache, pass a valid TLS context to the client's `tls_context` parameter:

```
import ssl
from pymemcache.client.base import Client

context = ssl.create_default_context(
    cafile="my-ca-root.crt",
)

client = Client(('localhost', 11211), tls_context=context)
client.set('some_key', 'some_value')
result = client.get('some_key')
```

## 1.6 Serialization

```
import json
from pymemcache.client.base import Client

class JsonSerde(object):
    def serialize(self, key, value):
        if isinstance(value, str):
            return value, 1
        return json.dumps(value), 2

    def deserialize(self, key, value, flags):
        if flags == 1:
            return value
        if flags == 2:
            return json.loads(value)
        raise Exception("Unknown serialization format")

client = Client(('localhost', 11211), serde=JsonSerde())
```

(continues on next page)



(continued from previous page)

```
client.set('key', {'a': 'b', 'c': 'd'})
result = client.get('key')
```

pymemcache provides a default `pickle`-based serializer:

```
from pymemcache.client.base import Client
from pymemcache import serde

class Foo(object):
    pass

client = Client(('localhost', 11211), serde=serde.pickle_serde)
client.set('key', Foo())
result = client.get('key')
```

The serializer uses the highest pickle protocol available. In order to make sure multiple versions of Python can read the protocol version, you can specify the version by explicitly instantiating `pymemcache.serde.PickleSerde`:

```
client = Client(
    ('localhost', 11211),
    serde=serde.PickleSerde(pickle_version=2)
)
```

## 1.7 Deserialization with Python 3

Values passed to the `serde.deserialize()` method will be bytestrings. It is therefore necessary to encode and decode them correctly. Here's a version of the `JsonSerde` from above which is more careful with encodings:

```
class JsonSerde(object):
    def serialize(self, key, value):
        if isinstance(value, str):
            return value.encode('utf-8'), 1
        return json.dumps(value).encode('utf-8'), 2

    def deserialize(self, key, value, flags):
        if flags == 1:
            return value.decode('utf-8')
        if flags == 2:
            return json.loads(value.decode('utf-8'))
        raise Exception("Unknown serialization format")
```

## 1.8 Key Constraints

This client implements the ASCII protocol of memcached. This means keys should not contain any of the following illegal characters:

Keys cannot have spaces, new lines, carriage returns, or null characters. We suggest that if you have unicode characters, or long keys, you use an effective hashing mechanism before calling this client.

At Pinterest, we have found that murmur3 hash is a great candidate for this. Alternatively you can set `allow_unicode_keys` to support unicode keys, but beware of what unicode encoding you use to make sure multiple clients can find the same key.

## 1.9 Best Practices

- Always set the `connect_timeout` and `timeout` arguments in the `pymemcache.client.base.Client` constructor to avoid blocking your process when memcached is slow. You might also want to enable the `no_delay` option, which sets the `TCP_NODELAY` flag on the connection's socket.
- Use the `noreply` flag for a significant performance boost. The `noreply` flag is enabled by default for “set”, “add”, “replace”, “append”, “prepend”, and “delete”. It is disabled by default for “cas”, “incr” and “decr”. It obviously doesn't apply to any get calls.
- Use `pymemcache.client.base.Client.get_many()` and `pymemcache.client.base.Client.gets_many()` whenever possible, as they result in fewer round trip times for fetching multiple keys.
- Use the `ignore_exc` flag to treat memcache/network errors as cache misses on calls to the `get*` methods. This prevents failures in memcache, or network errors, from killing your web requests. Do not use this flag if you need to know about errors from memcache, and make sure you have some other way to detect memcache server failures.
- Unless you have a known reason to do otherwise, use the provided serializer in `pymemcache.serde.pickle_serde` for any de/serialization of objects.

**Warning:** `noreply` will not read errors returned from the memcached server.

If a function with `noreply=True` causes an error on the server, it will still succeed and your next call which reads a response from memcached may fail unexpectedly.

`pymemcached` will try to catch and stop you from sending malformed inputs to memcached, but if you are having unexplained errors, setting `noreply=False` may help you troubleshoot the issue.

## PYMEMCACHE

## 2.1 pymemcache package

### 2.1.1 Subpackages

**pymemcache.client package**

**Submodules**

**pymemcache.client.base module**

```
class pymemcache.client.base.Client(server,      serde=None,      serializer=None,      dese-
                                   rializer=None,      connect_timeout=None,      time-
                                   out=None,      no_delay=False,      ignore_exc=False,
                                   socket_module=<module      'socket'      from
                                   '/home/docs/.pyenv/versions/3.7.3/lib/python3.7/socket.py'>,
                                   key_prefix=b'',      default_noreply=True,      al-
                                   low_unicode_keys=False,      encoding='ascii',
                                   tls_context=None)
```

Bases: `object`

A client for a single memcached server.

#### *Server Connection*

The `server` parameter controls how the client connects to the memcached server. You can either use a (host, port) tuple for a TCP connection or a string containing the path to a UNIX domain socket.

The `connect_timeout` and `timeout` parameters can be used to set socket timeout values. By default, timeouts are disabled.

When the `no_delay` flag is set, the `TCP_NODELAY` socket option will also be set. This only applies to TCP-based connections.

Lastly, the `socket_module` allows you to specify an alternate socket implementation (such as `gevent.socket`).

#### *Keys and Values*

Keys must have a `__str__()` method which should return a str with no more than 250 ASCII characters and no whitespace or control characters. Unicode strings must be encoded (as UTF-8, for example) unless they consist only of ASCII characters that are neither whitespace nor control characters.

Values must have a `__str__()` method to convert themselves to a byte string. Unicode objects can be a problem since `str()` on a Unicode object will attempt to encode it as ASCII (which will fail if the

value contains code points larger than U+127). You can fix this with a serializer or by just calling `encode` on the string (using UTF-8, for instance).

If you intend to use anything but `str` as a value, it is a good idea to use a serializer. The `pymemcache.serde` library has an already implemented serializer which pickles and unpickles data.

### *Serialization and Deserialization*

The constructor takes an optional object, the “serializer/deserializer” (“serde”), which is responsible for both serialization and deserialization of objects. That object must satisfy the serializer interface by providing two methods: *serialize* and *deserialize*. *serialize* takes two arguments, a key and a value, and returns a tuple of two elements, the serialized value, and an integer in the range 0-65535 (the “flags”). *deserialize* takes three parameters, a key, value, and flags, and returns the deserialized value.

Here is an example using JSON for non-str values:

```
class JSONSerde(object):
    def serialize(self, key, value):
        if isinstance(value, str):
            return value, 1
        return json.dumps(value), 2

    def deserialize(self, key, value, flags):
        if flags == 1:
            return value

        if flags == 2:
            return json.loads(value)

        raise Exception("Unknown flags for value: {1}".format(flags))
```

---

**Note:** Most write operations allow the caller to provide a `flags` value to support advanced interaction with the server. This will **override** the “flags” value returned by the serializer and should therefore only be used when you have a complete understanding of how the value should be serialized, stored, and deserialized.

---

### *Error Handling*

All of the methods in this class that talk to memcached can throw one of the following exceptions:

- `pymemcache.exceptions.MemcacheUnknownCommandError`
- `pymemcache.exceptions.MemcacheClientError`
- `pymemcache.exceptions.MemcacheServerError`
- `pymemcache.exceptions.MemcacheUnknownError`
- `pymemcache.exceptions.MemcacheUnexpectedCloseError`
- `pymemcache.exceptions.MemcacheIllegalInputError`
- `socket.timeout`
- `socket.error`

Instances of this class maintain a persistent connection to memcached which is terminated when any of these exceptions are raised. The next call to a method on the object will result in a new connection being made to memcached.

**add** (*key*, *value*, *expire*=0, *noreply*=None, *flags*=None)

The memcached “add” command.

**Parameters**

- **key** – str, see class docs for details.
- **value** – str, see class docs for details.
- **expire** – optional int, number of seconds until the item is expired from the cache, or zero for no expiry (the default).
- **noreply** – optional bool, True to not wait for the reply (defaults to self.default\_noreply).
- **flags** – optional int, arbitrary bit field used for server-specific flags

**Returns** If `noreply` is True (or if it is unset and `self.default_noreply` is True), the return value is always True. Otherwise the return value is True if the value was stored, and False if it was not (because the key already existed).

**append** (*key*, *value*, *expire*=0, *noreply*=None, *flags*=None)

The memcached “append” command.

**Parameters**

- **key** – str, see class docs for details.
- **value** – str, see class docs for details.
- **expire** – optional int, number of seconds until the item is expired from the cache, or zero for no expiry (the default).
- **noreply** – optional bool, True to not wait for the reply (defaults to self.default\_noreply).
- **flags** – optional int, arbitrary bit field used for server-specific flags

**Returns** True.

**cache\_memlimit** (*memlimit*)

The memcached “cache\_memlimit” command.

**Parameters** **memlimit** – int, the number of megabytes to set as the new cache memory limit.

**Returns** If no exception is raised, always returns True.

**cas** (*key*, *value*, *cas*, *expire*=0, *noreply*=False, *flags*=None)

The memcached “cas” command.

**Parameters**

- **key** – str, see class docs for details.
- **value** – str, see class docs for details.
- **cas** – int or str that only contains the characters ‘0’-‘9’.
- **expire** – optional int, number of seconds until the item is expired from the cache, or zero for no expiry (the default).
- **noreply** – optional bool, False to wait for the reply (the default).
- **flags** – optional int, arbitrary bit field used for server-specific flags

**Returns** If `noreply` is True (or if it is unset and `self.default_noreply` is True), the return value is always True. Otherwise returns None if the key didn’t exist, False if it existed but had a different cas value and True if it existed and was changed.

**check\_key** (*key*)

Checks key and add key\_prefix.

**close** ()

Close the connection to memcached, if it is open. The next call to a method that requires a connection will re-open it.

**decr** (*key, value, noreply=False*)

The memcached “decr” command.

**Parameters**

- **key** – str, see class docs for details.
- **value** – int, the amount by which to increment the value.
- **noreply** – optional bool, False to wait for the reply (the default).

**Returns** If noreply is True, always returns None. Otherwise returns the new value of the key, or None if the key wasn’t found.

**delete** (*key, noreply=None*)

The memcached “delete” command.

**Parameters**

- **key** – str, see class docs for details.
- **noreply** – optional bool, True to not wait for the reply (defaults to self.default\_noreply).

**Returns** If noreply is True (or if it is unset and self.default\_noreply is True), the return value is always True. Otherwise returns True if the key was deleted, and False if it wasn’t found.

**delete\_many** (*keys, noreply=None*)

A convenience function to delete multiple keys.

**Parameters**

- **keys** – list(str), the list of keys to delete.
- **noreply** – optional bool, True to not wait for the reply (defaults to self.default\_noreply).

**Returns** True. If an exception is raised then all, some or none of the keys may have been deleted. Otherwise all the keys have been sent to memcache for deletion and if noreply is False, they have been acknowledged by memcache.

**delete\_multi** (*keys, noreply=None*)

A convenience function to delete multiple keys.

**Parameters**

- **keys** – list(str), the list of keys to delete.
- **noreply** – optional bool, True to not wait for the reply (defaults to self.default\_noreply).

**Returns** True. If an exception is raised then all, some or none of the keys may have been deleted. Otherwise all the keys have been sent to memcache for deletion and if noreply is False, they have been acknowledged by memcache.

**flush\_all** (*delay=0, noreply=None*)

The memcached “flush\_all” command.

**Parameters**

- **delay** – optional int, the number of seconds to wait before flushing, or zero to flush immediately (the default).
- **noreply** – optional bool, True to not wait for the reply (defaults to self.default\_noreply).

**Returns** True.

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

The memcached “get” command, but only for one key, as a convenience.

**Parameters**

- **key** – str, see class docs for details.
- **default** – value that will be returned if the key was not found.

**Returns** The value for the key, or default if the key wasn’t found.

**get\_many** (*keys*)

The memcached “get” command.

**Parameters** **keys** – list(str), see class docs for details.

**Returns** A dict in which the keys are elements of the “keys” argument list and the values are values from the cache. The dict may contain all, some or none of the given keys.

**get\_multi** (*keys*)

The memcached “get” command.

**Parameters** **keys** – list(str), see class docs for details.

**Returns** A dict in which the keys are elements of the “keys” argument list and the values are values from the cache. The dict may contain all, some or none of the given keys.

**gets** (*key*, *default=None*, *cas\_default=None*)

The memcached “gets” command for one key, as a convenience.

**Parameters**

- **key** – str, see class docs for details.
- **default** – value that will be returned if the key was not found.
- **cas\_default** – same behaviour as default argument.

**Returns** A tuple of (value, cas) or (default, cas\_defaults) if the key was not found.

**gets\_many** (*keys*)

The memcached “gets” command.

**Parameters** **keys** – list(str), see class docs for details.

**Returns** A dict in which the keys are elements of the “keys” argument list and the values are tuples of (value, cas) from the cache. The dict may contain all, some or none of the given keys.

**incr** (*key*, *value*, *noreply=False*)

The memcached “incr” command.

**Parameters**

- **key** – str, see class docs for details.
- **value** – int, the amount by which to increment the value.
- **noreply** – optional bool, False to wait for the reply (the default).

**Returns** If `noreply` is `True`, always returns `None`. Otherwise returns the new value of the key, or `None` if the key wasn't found.

**prepend** (*key, value, expire=0, noreply=None, flags=None*)

The memcached "prepend" command.

**Parameters**

- **key** – str, see class docs for details.
- **value** – str, see class docs for details.
- **expire** – optional int, number of seconds until the item is expired from the cache, or zero for no expiry (the default).
- **noreply** – optional bool, `True` to not wait for the reply (defaults to `self.default_noreply`).
- **flags** – optional int, arbitrary bit field used for server-specific flags

**Returns** `True`.

**quit** ()

The memcached "quit" command.

This will close the connection with memcached. Calling any other method on this object will re-open the connection, so this object can be re-used after quit.

**replace** (*key, value, expire=0, noreply=None, flags=None*)

The memcached "replace" command.

**Parameters**

- **key** – str, see class docs for details.
- **value** – str, see class docs for details.
- **expire** – optional int, number of seconds until the item is expired from the cache, or zero for no expiry (the default).
- **noreply** – optional bool, `True` to not wait for the reply (defaults to `self.default_noreply`).
- **flags** – optional int, arbitrary bit field used for server-specific flags

**Returns** If `noreply` is `True` (or if it is unset and `self.default_noreply` is `True`), the return value is always `True`. Otherwise returns `True` if the value was stored and `False` if it wasn't (because the key didn't already exist).

**set** (*key, value, expire=0, noreply=None, flags=None*)

The memcached "set" command.

**Parameters**

- **key** – str, see class docs for details.
- **value** – str, see class docs for details.
- **expire** – optional int, number of seconds until the item is expired from the cache, or zero for no expiry (the default).
- **noreply** – optional bool, `True` to not wait for the reply (defaults to `self.default_noreply`).
- **flags** – optional int, arbitrary bit field used for server-specific flags

**Returns** If no exception is raised, always returns `True`. If an exception is raised, the set may or may not have occurred. If `noreply` is `True`, then a successful return does not guarantee a successful set.



**set\_many** (*values*, *expire=0*, *noreply=None*, *flags=None*)

A convenience function for setting multiple values.

**Parameters**

- **values** – dict(str, str), a dict of keys and values, see class docs for details.
- **expire** – optional int, number of seconds until the item is expired from the cache, or zero for no expiry (the default).
- **noreply** – optional bool, True to not wait for the reply (defaults to self.default\_noreply).
- **flags** – optional int, arbitrary bit field used for server-specific flags

**Returns** Returns a list of keys that failed to be inserted. If noreply is True, always returns empty list.

**set\_multi** (*values*, *expire=0*, *noreply=None*, *flags=None*)

A convenience function for setting multiple values.

**Parameters**

- **values** – dict(str, str), a dict of keys and values, see class docs for details.
- **expire** – optional int, number of seconds until the item is expired from the cache, or zero for no expiry (the default).
- **noreply** – optional bool, True to not wait for the reply (defaults to self.default\_noreply).
- **flags** – optional int, arbitrary bit field used for server-specific flags

**Returns** Returns a list of keys that failed to be inserted. If noreply is True, always returns empty list.

**stats** (\*args)

The memcached “stats” command.

The returned keys depend on what the “stats” command returns. A best effort is made to convert values to appropriate Python types, defaulting to strings when a conversion cannot be made.

**Parameters** **\*arg** – extra string arguments to the “stats” command. See the memcached protocol documentation for more information.

**Returns** A dict of the returned stats.

**touch** (*key*, *expire=0*, *noreply=None*)

The memcached “touch” command.

**Parameters**

- **key** – str, see class docs for details.
- **expire** – optional int, number of seconds until the item is expired from the cache, or zero for no expiry (the default).
- **noreply** – optional bool, True to not wait for the reply (defaults to self.default\_noreply).

**Returns** True if the expiration time was updated, False if the key wasn’t found.

**version** ()

The memcached “version” command.

**Returns** A string of the memcached version.

```
class pymemcache.client.base.PooledClient(server, serde=None, serializer=None, deserializer=None, connect_timeout=None, timeout=None, no_delay=False, ignore_exc=False, socket_module=<module 'socket' from '/home/docs/.pyenv/versions/3.7.3/lib/python3.7/socket.py'>, key_prefix=b'', max_pool_size=None, lock_generator=None, default_noreply=True, allow_unicode_keys=False, encoding='ascii', tls_context=None)
```

Bases: `object`

A thread-safe pool of clients (with the same client api).

#### Parameters

- **max\_pool\_size** – maximum pool size to use (going above this amount triggers a runtime error), by default this is 2147483648L when not provided (or none).
- **lock\_generator** – a callback/type that takes no arguments that will be called to create a lock or semaphore that can protect the pool from concurrent access (for example a eventlet lock or semaphore could be used instead)

Further arguments are interpreted as for `Client` constructor.

Note: if *serde* is given, the same object will be used for *all* clients in the pool. Your *serde* object must therefore be thread-safe.

**add** (*key, value, expire=0, noreply=None, flags=None*)

**append** (*key, value, expire=0, noreply=None, flags=None*)

**cas** (*key, value, cas, expire=0, noreply=False, flags=None*)

**check\_key** (*key*)

Checks key and add key\_prefix.

**client\_class**

`Client` class used to create new clients

alias of `Client`

**close** ()

**decr** (*key, value, noreply=False*)

**delete** (*key, noreply=None*)

**delete\_many** (*keys, noreply=None*)

**delete\_multi** (*keys, noreply=None*)

**flush\_all** (*delay=0, noreply=None*)

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

**get\_many** (*keys*)

**get\_multi** (*keys*)

**gets** (*key*)

**gets\_many** (*keys*)

**incr** (*key, value, noreply=False*)

**prepend** (*key, value, expire=0, noreply=None, flags=None*)

```

quit ()
replace (key, value, expire=0, noreply=None, flags=None)
set (key, value, expire=0, noreply=None, flags=None)
set_many (values, expire=0, noreply=None, flags=None)
set_multi (values, expire=0, noreply=None, flags=None)
stats (*args)
touch (key, expire=0, noreply=None)
version ()

```

`pymemcache.client.base.check_key_helper` (*key*, *allow\_unicode\_keys*, *key\_prefix=b''*)  
 Checks key and add key\_prefix.

## pymemcache.client.hash module

```

class pymemcache.client.hash.HashClient (servers,
                                          hasher=<class 'pymem-
cache.client.rendezvous.RendezvousHash'>,
                                          serde=None,      serializer=None,      deseri-
alizer=None,      connect_timeout=None,
                                          timeout=None,      no_delay=False,
                                          socket_module=<module 'socket' from
'/home/docs/.pyenv/versions/3.7.3/lib/python3.7/socket.py'>,
                                          key_prefix=b'',      max_pool_size=None,
                                          lock_generator=None,      retry_attempts=2,
                                          retry_timeout=1,      dead_timeout=60,
                                          use_pooling=False,      ignore_exc=False, al-
low_unicode_keys=False,      default_noreply=True,
                                          encoding='ascii', tls_context=None)

```

Bases: `object`

A client for communicating with a cluster of memcached servers

```

add (key, *args, **kwargs)
add_server (server, port)
append (key, *args, **kwargs)
cas (key, *args, **kwargs)
client_class
    alias of pymemcache.client.base.Client
decr (key, *args, **kwargs)
delete (key, *args, **kwargs)
delete_many (keys, *args, **kwargs)
delete_multi (keys, *args, **kwargs)
flush_all ()
get (key, *args, **kwargs)
get_many (keys, gets=False, *args, **kwargs)
get_multi (keys, gets=False, *args, **kwargs)

```

```
gets (key, *args, **kwargs)  
gets_many (keys, *args, **kwargs)  
gets_multi (keys, *args, **kwargs)  
incr (key, *args, **kwargs)  
prepend (key, *args, **kwargs)  
remove_server (server, port)  
replace (key, *args, **kwargs)  
set (key, *args, **kwargs)  
set_many (values, *args, **kwargs)  
set_multi (values, *args, **kwargs)  
touch (key, *args, **kwargs)
```

## pymemcache.client.murmur3 module

`pymemcache.client.murmur3.murmur3_32` (*data*, *seed*=0)

MurmurHash3 was written by Austin Appleby, and is placed in the public domain. The author hereby disclaims copyright to this source code.

## pymemcache.client.rendezvous module

```
class pymemcache.client.rendezvous.RendezvousHash (nodes=None, seed=0,  
                                                  hash_function=<function mur-  
                                                  mur3_32>)
```

Bases: `object`

Implements the Highest Random Weight (HRW) hashing algorithm most commonly referred to as rendezvous hashing.

Originally developed as part of python-clandestined.

Copyright (c) 2014 Ernest W. Durbin III

```
add_node (node)  
get_node (key)  
remove_node (node)
```

## Module contents

### pymemcache.test package

#### Submodules

### pymemcache.test.conftest module

### pymemcache.test.test\_benchmark module

`pymemcache.test.test_client` module

`pymemcache.test.test_client_hash` module

`pymemcache.test.test_integration` module

`pymemcache.test.test_rendezvous` module

`pymemcache.test.test_serde` module

`pymemcache.test.test_utils` module

`pymemcache.test.utils` module

Useful testing utilities.

This module is considered public API.

```
class pymemcache.test.utils.MockMemcacheClient (server=None,  serde=None,  seri-
                                             alizer=None,  deserializer=None,
                                             connect_timeout=None,  time-
                                             out=None,  no_delay=False,  ig-
                                             nore_exc=False,  socket_module=None,
                                             default_noreply=True,  al-
                                             low_unicode_keys=False,  encod-
                                             ing='ascii',  tls_context=None)
```

Bases: `object`

A (partial) in-memory mock for Clients.

**add** (*key*, *value*, *expire*=0, *noreply*=True, *flags*=None)

**append** (*key*, *value*, *expire*=0, *noreply*=True, *flags*=None)

**cache\_memlimit** (*memlimit*)

**cas** (*key*, *value*, *cas*, *expire*=0, *noreply*=False, *flags*=None)

**check\_key** (*key*)

Checks key and add key\_prefix.

**clear** ()

Method used to clear/reset mock cache

**close** ()

**decr** (*key*, *value*, *noreply*=False)

**delete** (*key*, *noreply*=True)

**delete\_many** (*keys*, *noreply*=True)

**delete\_multi** (*keys*, *noreply*=True)

**flush\_all** (*delay*=0, *noreply*=True)

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

**get\_many** (*keys*)

**get\_multi** (*keys*)

```
incr (key, value, noreply=False)  
prepend (key, value, expire=0, noreply=True, flags=None)  
quit ()  
replace (key, value, expire=0, noreply=True, flags=None)  
set (key, value, expire=0, noreply=True, flags=None)  
set_many (values, expire=0, noreply=True, flags=None)  
set_multi (values, expire=0, noreply=True, flags=None)  
stats (*args)  
touch (key, expire=0, noreply=True)  
version ()
```

## Module contents

### 2.1.2 Submodules

#### pymemcache.exceptions module

**exception** `pymemcache.exceptions.MemcacheClientError`

Bases: `pymemcache.exceptions.MemcacheError`

Raised when memcached fails to parse the arguments to a request, likely due to a malformed key and/or value, a bug in this library, or a version mismatch with memcached.

**exception** `pymemcache.exceptions.MemcacheError`

Bases: `Exception`

Base exception class

**exception** `pymemcache.exceptions.MemcacheIllegalInputError`

Bases: `pymemcache.exceptions.MemcacheClientError`

Raised when a key or value is not legal for Memcache (see the class docs for Client for more details).

**exception** `pymemcache.exceptions.MemcacheServerError`

Bases: `pymemcache.exceptions.MemcacheError`

Raised when memcached reports a failure while processing a request, likely due to a bug or transient issue in memcached.

**exception** `pymemcache.exceptions.MemcacheUnexpectedCloseError`

Bases: `pymemcache.exceptions.MemcacheServerError`

Raised when the connection with memcached closes unexpectedly.

**exception** `pymemcache.exceptions.MemcacheUnknownCommandError`

Bases: `pymemcache.exceptions.MemcacheClientError`

Raised when memcached fails to parse a request, likely due to a bug in this library or a version mismatch with memcached.

**exception** `pymemcache.exceptions.MemcacheUnknownError`

Bases: `pymemcache.exceptions.MemcacheError`

Raised when this library receives a response from memcached that it cannot parse, likely due to a bug in this library or a version mismatch with memcached.

## pymemcache.fallback module

A client for falling back to older memcached servers when performing reads.

It is sometimes necessary to deploy memcached on new servers, or with a different configuration. In these cases, it is undesirable to start up an empty memcached server and point traffic to it, since the cache will be cold, and the backing store will have a large increase in traffic.

This class attempts to solve that problem by providing an interface identical to the Client interface, but which can fall back to older memcached servers when reads to the primary server fail. The approach for upgrading memcached servers or configuration then becomes:

1. Deploy a new host (or fleet) with memcached, possibly with a new configuration.
2. From your application servers, use FallbackClient to write and read from the new cluster, and to read from the old cluster when there is a miss in the new cluster.
3. Wait until the new cache is warm enough to support the load.
4. Switch from FallbackClient to a regular Client library for doing all reads and writes to the new cluster.
5. Take down the old cluster.

### Best Practices:

- Make sure that the old client has “ignore\_exc” set to True, so that it treats failures like cache misses. That will allow you to take down the old cluster before you switch away from FallbackClient.

```
class pymemcache.fallback.FallbackClient (caches)
```

```
    Bases: object
```

```
    add (key, value, expire=0, noreply=True)
```

```
    append (key, value, expire=0, noreply=True)
```

```
    cas (key, value, cas, expire=0, noreply=True)
```

```
    close ()
```

```
        Close each of the memcached clients
```

```
    decr (key, value, noreply=True)
```

```
    delete (key, noreply=True)
```

```
    flush_all (delay=0, noreply=True)
```

```
    get (key)
```

```
    get_many (keys)
```

```
    gets (key)
```

```
    gets_many (keys)
```

```
    incr (key, value, noreply=True)
```

```
    prepend (key, value, expire=0, noreply=True)
```

```
    quit ()
```

```
    replace (key, value, expire=0, noreply=True)
```

```
    set (key, value, expire=0, noreply=True)
```

```
    stats ()
```

**touch** (*key*, *expire=0*, *noreply=True*)

## pymemcache.pool module

**class** `pymemcache.pool.ObjectPool` (*obj\_creator*, *after\_remove=None*, *max\_size=None*,  
*lock\_generator=None*)

Bases: `object`

A pool of objects that release/creates/destroys as needed.

**clear** ()

**destroy** (*obj*, *silent=True*)

**property free**

**get** ()

**get\_and\_release** (*destroy\_on\_fail=False*)

**release** (*obj*, *silent=True*)

**property used**

## pymemcache.serde module

**class** `pymemcache.serde.LegacyWrappingSerde` (*serializer\_func*, *deserializer\_func*)

Bases: `object`

This class defines how to wrap legacy de/serialization functions into a ‘serde’ object which implements ‘.serialize’ and ‘.deserialize’ methods. It is used automatically by `pymemcache.client.base.Client` when the ‘serializer’ or ‘deserializer’ arguments are given.

The `serializer_func` and `deserializer_func` are expected to be `None` in the case that they are missing.

**class** `pymemcache.serde.PickleSerde` (*pickle\_version=4*)

Bases: `object`

An object which implements the serialization/deserialization protocol for `pymemcache.client.base.Client` and its descendants using the `pickle` module.

Serialization and deserialization are implemented as methods of this class. To implement a custom serialization/deserialization method for `pymemcache`, you should implement the same interface as the one provided by this object – `pymemcache.serde.PickleSerde.serialize()` and `pymemcache.serde.PickleSerde.deserialize()`. Then, pass your custom object to the `pymemcache` client object in place of `PickleSerde`.

For more details on the serialization protocol, see the class documentation for `pymemcache.client.base.Client`

**deserialize** (*key*, *value*, *flags*)

**serialize** (*key*, *value*)

`pymemcache.serde.get_python_memcache_serializer` (*pickle\_version=4*)

Return a serializer using a specific pickle version

`pymemcache.serde.python_memcache_deserializer` (*key*, *value*, *flags*)

`pymemcache.serde.python_memcache_serializer` (*key*, *value*, \*, *pickle\_version=4*)



### 2.1.3 Module contents



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### p

- `pymemcache`, [21](#)
- `pymemcache.client`, [16](#)
- `pymemcache.client.base`, [7](#)
- `pymemcache.client.hash`, [15](#)
- `pymemcache.client.murmur3`, [16](#)
- `pymemcache.client.rendezvous`, [16](#)
- `pymemcache.exceptions`, [18](#)
- `pymemcache.fallback`, [19](#)
- `pymemcache.pool`, [20](#)
- `pymemcache.serde`, [20](#)
- `pymemcache.test`, [18](#)
- `pymemcache.test.utils`, [17](#)



## A

[add\(\)](#) (*pymemcache.client.base.Client* method), 8  
[add\(\)](#) (*pymemcache.client.base.PooledClient* method), 14  
[add\(\)](#) (*pymemcache.client.hash.HashClient* method), 15  
[add\(\)](#) (*pymemcache.fallback.FallbackClient* method), 19  
[add\(\)](#) (*pymemcache.test.utils.MockMemcacheClient* method), 17  
[add\\_node\(\)](#) (*pymemcache.client.rendezvous.RendezvousHash* method), 16  
[add\\_server\(\)](#) (*pymemcache.client.hash.HashClient* method), 15  
[append\(\)](#) (*pymemcache.client.base.Client* method), 9  
[append\(\)](#) (*pymemcache.client.base.PooledClient* method), 14  
[append\(\)](#) (*pymemcache.client.hash.HashClient* method), 15  
[append\(\)](#) (*pymemcache.fallback.FallbackClient* method), 19  
[append\(\)](#) (*pymemcache.test.utils.MockMemcacheClient* method), 17

## C

[cache\\_memlimit\(\)](#) (*pymemcache.client.base.Client* method), 9  
[cache\\_memlimit\(\)](#) (*pymemcache.test.utils.MockMemcacheClient* method), 17  
[cas\(\)](#) (*pymemcache.client.base.Client* method), 9  
[cas\(\)](#) (*pymemcache.client.base.PooledClient* method), 14  
[cas\(\)](#) (*pymemcache.client.hash.HashClient* method), 15  
[cas\(\)](#) (*pymemcache.fallback.FallbackClient* method), 19  
[cas\(\)](#) (*pymemcache.test.utils.MockMemcacheClient* method), 17  
[check\\_key\(\)](#) (*pymemcache.client.base.Client* method), 9

[check\\_key\(\)](#) (*pymemcache.client.base.PooledClient* method), 14  
[check\\_key\(\)](#) (*pymemcache.test.utils.MockMemcacheClient* method), 17  
[check\\_key\\_helper\(\)](#) (in module *pymemcache.client.base*), 15  
[clear\(\)](#) (*pymemcache.pool.ObjectPool* method), 20  
[clear\(\)](#) (*pymemcache.test.utils.MockMemcacheClient* method), 17  
[Client](#) (class in *pymemcache.client.base*), 7  
[client\\_class](#) (*pymemcache.client.base.PooledClient* attribute), 14  
[client\\_class](#) (*pymemcache.client.hash.HashClient* attribute), 15  
[close\(\)](#) (*pymemcache.client.base.Client* method), 10  
[close\(\)](#) (*pymemcache.client.base.PooledClient* method), 14  
[close\(\)](#) (*pymemcache.fallback.FallbackClient* method), 19  
[close\(\)](#) (*pymemcache.test.utils.MockMemcacheClient* method), 17

## D

[decr\(\)](#) (*pymemcache.client.base.Client* method), 10  
[decr\(\)](#) (*pymemcache.client.base.PooledClient* method), 14  
[decr\(\)](#) (*pymemcache.client.hash.HashClient* method), 15  
[decr\(\)](#) (*pymemcache.fallback.FallbackClient* method), 19  
[decr\(\)](#) (*pymemcache.test.utils.MockMemcacheClient* method), 17  
[delete\(\)](#) (*pymemcache.client.base.Client* method), 10  
[delete\(\)](#) (*pymemcache.client.base.PooledClient* method), 14  
[delete\(\)](#) (*pymemcache.client.hash.HashClient* method), 15  
[delete\(\)](#) (*pymemcache.fallback.FallbackClient* method), 19  
[delete\(\)](#) (*pymemcache.test.utils.MockMemcacheClient* method), 17

`delete_many()` (*pymemcache.client.base.Client method*), 10

`delete_many()` (*pymemcache.client.base.PooledClient method*), 14

`delete_many()` (*pymemcache.client.hash.HashClient method*), 15

`delete_many()` (*pymemcache.test.utils.MockMemcacheClient method*), 17

`delete_multi()` (*pymemcache.client.base.Client method*), 10

`delete_multi()` (*pymemcache.client.base.PooledClient method*), 14

`delete_multi()` (*pymemcache.client.hash.HashClient method*), 15

`delete_multi()` (*pymemcache.test.utils.MockMemcacheClient method*), 17

`deserialize()` (*pymemcache.serde.PickleSerde method*), 20

`destroy()` (*pymemcache.pool.ObjectPool method*), 20

## F

`FallbackClient` (class in *pymemcache.fallback*), 19

`flush_all()` (*pymemcache.client.base.Client method*), 10

`flush_all()` (*pymemcache.client.base.PooledClient method*), 14

`flush_all()` (*pymemcache.client.hash.HashClient method*), 15

`flush_all()` (*pymemcache.fallback.FallbackClient method*), 19

`flush_all()` (*pymemcache.test.utils.MockMemcacheClient method*), 17

`free()` (*pymemcache.pool.ObjectPool property*), 20

## G

`get()` (*pymemcache.client.base.Client method*), 11

`get()` (*pymemcache.client.base.PooledClient method*), 14

`get()` (*pymemcache.client.hash.HashClient method*), 15

`get()` (*pymemcache.fallback.FallbackClient method*), 19

`get()` (*pymemcache.pool.ObjectPool method*), 20

`get()` (*pymemcache.test.utils.MockMemcacheClient method*), 17

`get_and_release()` (*pymemcache.pool.ObjectPool method*), 20

`get_many()` (*pymemcache.client.base.Client method*), 11

`get_many()` (*pymemcache.client.base.PooledClient method*), 14

`get_many()` (*pymemcache.client.hash.HashClient method*), 15

`get_many()` (*pymemcache.fallback.FallbackClient method*), 19

`get_many()` (*pymemcache.test.utils.MockMemcacheClient method*), 17

`get_multi()` (*pymemcache.client.base.Client method*), 11

`get_multi()` (*pymemcache.client.base.PooledClient method*), 14

`get_multi()` (*pymemcache.client.hash.HashClient method*), 15

`get_multi()` (*pymemcache.test.utils.MockMemcacheClient method*), 17

`get_node()` (*pymemcache.client.rendezvous.RendezvousHash method*), 16

`get_python_memcache_serializer()` (in module *pymemcache.serde*), 20

`gets()` (*pymemcache.client.base.Client method*), 11

`gets()` (*pymemcache.client.base.PooledClient method*), 14

`gets()` (*pymemcache.client.hash.HashClient method*), 15

`gets()` (*pymemcache.fallback.FallbackClient method*), 19

`gets_many()` (*pymemcache.client.base.Client method*), 11

`gets_many()` (*pymemcache.client.base.PooledClient method*), 14

`gets_many()` (*pymemcache.client.hash.HashClient method*), 16

`gets_many()` (*pymemcache.fallback.FallbackClient method*), 19

`gets_multi()` (*pymemcache.client.hash.HashClient method*), 16

## H

`HashClient` (class in *pymemcache.client.hash*), 15

## I

`incr()` (*pymemcache.client.base.Client method*), 11

`incr()` (*pymemcache.client.base.PooledClient method*), 14

`incr()` (*pymemcache.client.hash.HashClient method*), 16

`incr()` (*pymemcache.fallback.FallbackClient method*), 19

`incr()` (*pymemcache.test.utils.MockMemcacheClient method*), 17



## L

LegacyWrappingSerde (class in pymemcache.serde), 20

## M

MemcacheClientError, 18

MemcacheError, 18

MemcacheIllegalInputError, 18

MemcacheServerError, 18

MemcacheUnexpectedCloseError, 18

MemcacheUnknownCommandError, 18

MemcacheUnknownError, 18

MockMemcacheClient (class in pymemcache.test.utils), 17

module

pymemcache, 21

pymemcache.client, 16

pymemcache.client.base, 7

pymemcache.client.hash, 15

pymemcache.client.murmur3, 16

pymemcache.client.rendezvous, 16

pymemcache.exceptions, 18

pymemcache.fallback, 19

pymemcache.pool, 20

pymemcache.serde, 20

pymemcache.test, 18

pymemcache.test.utils, 17

murmur3\_32() (in module pymemcache.client.murmur3), 16

## O

ObjectPool (class in pymemcache.pool), 20

## P

PickleSerde (class in pymemcache.serde), 20

PooledClient (class in pymemcache.client.base), 13

prepend() (pymemcache.client.base.Client method), 12

prepend() (pymemcache.client.base.PooledClient method), 14

prepend() (pymemcache.client.hash.HashClient method), 16

prepend() (pymemcache.fallback.FallbackClient method), 19

prepend() (pymemcache.test.utils.MockMemcacheClient method), 18

pymemcache

module, 21

pymemcache.client

module, 16

pymemcache.client.base

module, 7

pymemcache.client.hash

module, 15

pymemcache.client.murmur3  
module, 16

pymemcache.client.rendezvous  
module, 16

pymemcache.exceptions  
module, 18

pymemcache.fallback  
module, 19

pymemcache.pool  
module, 20

pymemcache.serde  
module, 20

pymemcache.test  
module, 18

pymemcache.test.utils  
module, 17

python\_memcache\_deserializer() (in module pymemcache.serde), 20

python\_memcache\_serializer() (in module pymemcache.serde), 20

## Q

quit() (pymemcache.client.base.Client method), 12

quit() (pymemcache.client.base.PooledClient method), 14

quit() (pymemcache.fallback.FallbackClient method), 19

quit() (pymemcache.test.utils.MockMemcacheClient method), 18

## R

release() (pymemcache.pool.ObjectPool method), 20

remove\_node() (pymemcache.client.rendezvous.RendezvousHash method), 16

remove\_server() (pymemcache.client.hash.HashClient method), 16

RendezvousHash (class in pymemcache.client.rendezvous), 16

replace() (pymemcache.client.base.Client method), 12

replace() (pymemcache.client.base.PooledClient method), 15

replace() (pymemcache.client.hash.HashClient method), 16

replace() (pymemcache.fallback.FallbackClient method), 19

replace() (pymemcache.test.utils.MockMemcacheClient method), 18

## S

serialize() (pymemcache.serde.PickleSerde method), 20

[set \(\) \(pymemcache.client.base.Client method\), 12](#)  
[set \(\) \(pymemcache.client.base.PooledClient method\), 15](#)  
[set \(\) \(pymemcache.client.hash.HashClient method\), 16](#)  
[set \(\) \(pymemcache.fallback.FallbackClient method\), 19](#)  
[set \(\) \(pymemcache.test.utils.MockMemcacheClient method\), 18](#)  
[set\\_many \(\) \(pymemcache.client.base.Client method\), 12](#)  
[set\\_many \(\) \(pymemcache.client.base.PooledClient method\), 15](#)  
[set\\_many \(\) \(pymemcache.client.hash.HashClient method\), 16](#)  
[set\\_many \(\) \(pymemcache.test.utils.MockMemcacheClient method\), 18](#)  
[set\\_multi \(\) \(pymemcache.client.base.Client method\), 13](#)  
[set\\_multi \(\) \(pymemcache.client.base.PooledClient method\), 15](#)  
[set\\_multi \(\) \(pymemcache.client.hash.HashClient method\), 16](#)  
[set\\_multi \(\) \(pymemcache.test.utils.MockMemcacheClient method\), 18](#)  
[stats \(\) \(pymemcache.client.base.Client method\), 13](#)  
[stats \(\) \(pymemcache.client.base.PooledClient method\), 15](#)  
[stats \(\) \(pymemcache.fallback.FallbackClient method\), 19](#)  
[stats \(\) \(pymemcache.test.utils.MockMemcacheClient method\), 18](#)

## T

[touch \(\) \(pymemcache.client.base.Client method\), 13](#)  
[touch \(\) \(pymemcache.client.base.PooledClient method\), 15](#)  
[touch \(\) \(pymemcache.client.hash.HashClient method\), 16](#)  
[touch \(\) \(pymemcache.fallback.FallbackClient method\), 19](#)  
[touch \(\) \(pymemcache.test.utils.MockMemcacheClient method\), 18](#)

## U

[used \(\) \(pymemcache.pool.ObjectPool property\), 20](#)

## V

[version \(\) \(pymemcache.client.base.Client method\), 13](#)  
[version \(\) \(pymemcache.client.base.PooledClient method\), 15](#)

[version \(\) \(pymemcache.test.utils.MockMemcacheClient method\), 18](#)