# memoize Documentation

**Michal Zmuda**

**Jan 03, 2022**

# Contents:


**1 Etymology** **3**

**2 Getting Started** **5**
2.1 Installation . . . 5
2.2 Usage . . . 5

**3 Features** **7**
3.1 Async-first . . . 7
3.2 Tornado & asyncio support . . . 7
3.3 Configurability . . . 7
3.4 Tunable eviction & async refreshing . . . 9
3.5 Dog-piling proofness . . . 9
3.6 Async cache storage . . . 11
3.7 Manual Invalidation . . . 11

**4 Contributing** **13**
4.1 Coverage . . . 13
4.2 Docs . . . 13
4.3 PyPi . . . 14

**5 API docs** **15**
5.1 memoize package . . . 15

**6 Changelog** **23**
6.1 1.1.3 . . . 23
6.2 1.1.2 . . . 23
6.3 1.1.1 . . . 23
6.4 1.1.0 . . . 23
6.5 1.0.4 . . . 24
6.6 1.0.3 . . . 24
6.7 1.0.2 . . . 24
6.8 1.0.1 . . . 24
6.9 1.0.0 . . . 24

**7 Indices and tables** **25**

**Python Module Index** **27**

Index 29

Extended docs (including API docs) available at memoize.readthedocs.io.

**What:** Caching library for asynchronous Python applications.

**Why:** Python deserves library that works in async world (for instance handles dog-piling ) and has a proper, extensible API.

# CHAPTER 1

# Etymology

*In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. (...) The term "memoization" was coined by Donald Michie in 1968 and is derived from the Latin word "memorandum" ("to be remembered"), usually truncated as "memo" in the English language, and thus carries the meaning of "turning [the results of] a function into something to be remembered."* ~ Wikipedia

CHAPTER 2

# Getting Started

## 2.1 Installation

### 2.1.1 Basic Installation

To get you up & running all you need is to install:

```
pip install py-memoize
```

### 2.1.2 Installation of Extras

If you are going to use `memoize` with tornado add a dependency on extra:

```
pip install py-memoize[tornado]
```

To harness the power of ujson (if JSON SerDe is used) install extra:

```
pip install py-memoize[ujson]
```

## 2.2 Usage

Provided examples use default configuration to cache results in memory. For configuration options see *Configurability*.

You can use `memoize` with both asyncio and Tornado - please see the appropriate example:

### 2.2.1 asyncio

To apply default caching configuration use:

```
import asyncio
import random
from memoize.wrapper import memoize


@memoize()
async def expensive_computation():
    return 'expensive-computation-' + str(random.randint(1, 100))


async def main():
    print(await expensive_computation())
    print(await expensive_computation())
    print(await expensive_computation())


if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

### 2.2.2 Tornado

If your project is based on Tornado use:

```
import random

from tornado import gen
from tornado.ioloop import IOLoop

from memoize.wrapper import memoize


@memoize()
@gen.coroutine
def expensive_computation():
    return 'expensive-computation-' + str(random.randint(1, 100))


@gen.coroutine
def main():
    result1 = yield expensive_computation()
    print(result1)
    result2 = yield expensive_computation()
    print(result2)
    result3 = yield expensive_computation()
    print(result3)


if __name__ == "__main__":
    IOLoop.current().run_sync(main)
```

CHAPTER 3

# Features

## 3.1 Async-first

Asynchronous programming is often seen as a huge performance boost in python programming. But with all the benefits it brings there are also new concurrency-related caveats like dog-piling.

This library is built async-oriented from the ground-up, what manifests in, for example, in *Dog-piling proofness* or *Async cache storage*.

## 3.2 Tornado & asyncio support

No matter what are you using, build-in asyncio or its predecessor Tornado *memoize* has you covered as you can use it with both. **This may come handy if you are planning a migration from Tornado to asyncio.**

Under the hood *memoize* detects if you are using *Tornado* or *asyncio* (by checking if *Tornado* is installed and available to import).

If have *Tornado* installed but your application uses *asyncio* IO-loop, set `MEMOIZE_FORCE_ASYNCIO=1` environment variable to force using *asyncio* and ignore *Tornado* instalation.

## 3.3 Configurability

With *memoize* you have under control:

- timeout applied to the cached method;
- key generation strategy (see `memoize.key.KeyExtractor`); already provided strategies use arguments (both positional & keyword) and method name (or reference);
- storage for cached entries/items (see `memoize.storage.CacheStorage`); in-memory storage is already provided; for convenience of implementing new storage adapters some SerDe (`memoize.serde.SerDe`) are provided;

- eviction strategy (see *memoize.eviction.EvictionStrategy*); least-recently-updated strategy is already provided;
- entry builder (see *memoize.entrybuilder.CacheEntryBuilder*) which has control over `update_after` & `expires_after` described in *Tunable eviction & async refreshing*

All of these elements are open for extension (you can implement and plug-in your own). Please contribute!

Example how to customize default config (everything gets overridden):

```
from datetime import timedelta

from memoize.configuration import MutableCacheConfiguration,␣
↪DefaultInMemoryCacheConfiguration
from memoize.entrybuilder import ProvidedLifeSpanCacheEntryBuilder
from memoize.eviction import LeastRecentlyUpdatedEvictionStrategy
from memoize.key import EncodedMethodNameAndArgsKeyExtractor
from memoize.storage import LocalInMemoryCacheStorage
from memoize.wrapper import memoize


@memoize(configuration=MutableCacheConfiguration
         .initialized_with(DefaultInMemoryCacheConfiguration())
         .set_method_timeout(value=timedelta(minutes=2))
         .set_entry_builder(ProvidedLifeSpanCacheEntryBuilder(update_
↪after=timedelta(minutes=2),
                                                              expire_
↪after=timedelta(minutes=5)))
         .set_eviction_strategy(LeastRecentlyUpdatedEvictionStrategy(capacity=2048))
         .set_key_extractor(EncodedMethodNameAndArgsKeyExtractor(skip_first_arg_as_
↪self=False))
         .set_storage(LocalInMemoryCacheStorage())
         )
async def cached():
    return 'dummy'
```

Still, you can use default configuration which:

- sets timeout for underlying method to 2 minutes;
- uses in-memory storage;
- uses method instance & arguments to infer cache key;
- stores up to 4096 elements in cache and evicts entries according to least recently updated policy;
- refreshes elements after 10 minutes & ignores unrefreshed elements after 30 minutes.

If that satisfies you, just use default config:

```
from memoize.configuration import DefaultInMemoryCacheConfiguration
from memoize.wrapper import memoize


@memoize(configuration=DefaultInMemoryCacheConfiguration())
async def cached():
    return 'dummy'
```

Also, if you want to stick to the building blocks of the default configuration, but need to adjust some basic params:

```
from datetime import timedelta

from memoize.configuration import DefaultInMemoryCacheConfiguration
from memoize.wrapper import memoize


@memoize(configuration=DefaultInMemoryCacheConfiguration(capacity=4096, method_
↪timeout=timedelta(minutes=2),
                                                         update_
↪after=timedelta(minutes=10),
                                                         expire_
↪after=timedelta(minutes=30)))
async def cached():
    return 'dummy'
```

## 3.4 Tunable eviction & async refreshing

Sometimes caching libraries allow providing TTL only. This may result in a scenario where when the cache entry expires latency is increased as the new value needs to be recomputed. To mitigate this periodic extra latency multiple delays are often used. In the case of *memoize* there are two (see `memoize.entrybuilder.ProvidedLifeSpanCacheEntryBuilder`):

- `update_after` defines delay after which background/async update is executed;
- `expire_after` defines delay after which entry is considered outdated and invalid.

This allows refreshing cached value in the background without any observable latency. Moreover, if some of those background refreshes fail they will be retried still in the background. Due to this beneficial feature, it is recommended to `update_after` be significantly shorter than `expire_after`.

## 3.5 Dog-piling proofness

If some resource is accessed asynchronously dog-piling may occur. Caches designed for synchronous python code (like built-in LRU) will allow multiple concurrent tasks to observe a miss for the same resource and will proceed to flood underlying/cached backend with requests for the same resource.

As it breaks the purpose of caching (as backend effectively sometimes is not protected with cache) *memoize* has built-in dog-piling protection.

Under the hood, concurrent requests for the same resource (cache key) get collapsed to a single request to the backend. When the resource is fetched all requesters obtain the result. On failure, all requesters get an exception (same happens on timeout).

An example of what it all is about:

```
import asyncio
from datetime import timedelta

from aiocache import cached, SimpleMemoryCache  # version 0.11.1 (latest) used as␣
↪example of other cache implementation

from memoize.configuration import DefaultInMemoryCacheConfiguration
from memoize.wrapper import memoize
```

(continues on next page)

(continued from previous page)

```
# scenario configuration
concurrent_requests = 5
request_batches_execution_count = 50
cached_value_ttl_ms = 200
delay_between_request_batches_ms = 70

# results/statistics
unique_calls_under_memoize = 0
unique_calls_under_different_cache = 0


@memoize(configuration=DefaultInMemoryCacheConfiguration(update_
↪after=timedelta(milliseconds=cached_value_ttl_ms)))
async def cached_with_memoize():
    global unique_calls_under_memoize
    unique_calls_under_memoize += 1
    await asyncio.sleep(0.01)
    return unique_calls_under_memoize


@cached(ttl=cached_value_ttl_ms / 1000, cache=SimpleMemoryCache)
async def cached_with_different_cache():
    global unique_calls_under_different_cache
    unique_calls_under_different_cache += 1
    await asyncio.sleep(0.01)
    return unique_calls_under_different_cache


async def main():
    for i in range(request_batches_execution_count):
        await asyncio.gather(*[x() for x in [cached_with_memoize] * concurrent_
↪requests])
        await asyncio.gather(*[x() for x in [cached_with_different_cache] *␣
↪concurrent_requests])
        await asyncio.sleep(delay_between_request_batches_ms / 1000)

    print("Memoize generated {} unique backend calls".format(unique_calls_under_
↪memoize))
    print("Other cache generated {} unique backend calls".format(unique_calls_under_
↪different_cache))
    predicted = (delay_between_request_batches_ms * request_batches_execution_count) /
↪/ cached_value_ttl_ms
    print("Predicted (according to TTL) {} unique backend calls".format(predicted))

    # Printed:
    # Memoize generated 17 unique backend calls
    # Other cache generated 85 unique backend calls
    # Predicted (according to TTL) 17 unique backend calls

if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

## 3.6 Async cache storage

Interface for cache storage allows you to fully harness benefits of asynchronous programming (see interface of `memoize.storage.CacheStorage`).

Currently *memoize* provides only in-memory storage for cache values (internally at *RASP* we have others). If you want (for instance) Redis integration, you need to implement one (please contribute!) but *memoize* will optimally use your async implementation from the start.

## 3.7 Manual Invalidation

You could also invalidate entries manually. To do so you need to create instance of `memoize.invalidation.InvalidationSupport`) and pass it alongside cache configuration. Then you could just pass args and kwargs for which you want to invalidate entry.

```
from memoize.configuration import DefaultInMemoryCacheConfiguration
from memoize.invalidation import InvalidationSupport


import asyncio
import random
from memoize.wrapper import memoize

invalidation = InvalidationSupport()


@memoize(configuration=DefaultInMemoryCacheConfiguration(), invalidation=invalidation)
async def expensive_computation(*args, **kwargs):
    return 'expensive-computation-' + str(random.randint(1, 100))


async def main():
    print(await expensive_computation('arg1', kwarg='kwarg1'))
    print(await expensive_computation('arg1', kwarg='kwarg1'))

    print("Invalidation #1")
    await invalidation.invalidate_for_arguments(('arg1',), {'kwarg': 'kwarg1'})

    print(await expensive_computation('arg1', kwarg='kwarg1'))
    print(await expensive_computation('arg1', kwarg='kwarg1'))

    print("Invalidation #2")
    await invalidation.invalidate_for_arguments(('arg1',), {'kwarg': 'kwarg1'})

    print(await expensive_computation('arg1', kwarg='kwarg1'))

    # Sample output:
    #
    # expensive - computation - 98
    # expensive - computation - 98
    # Invalidation  # 1
    # expensive - computation - 73
    # expensive - computation - 73
    # Invalidation  # 2
    # expensive - computation - 59
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

CHAPTER 4

# Contributing

## 4.1 Coverage

All test cases/scenarios & coverage are executed by tox:

```
tox
```

For users who use multiple interpreters:

```
# python3.8 is just an example
python3.8 -m tox
```

To run testing scenarios in IDE of your choice please see `tox.ini` (especially take a look how `MEMOIZE_FORCE_ASYNCIO` is used).

## 4.2 Docs

Before submitting pull request please update modules apidocs & ensure documentation generates properly. The following commands will do both (watch for errors & warnings).

```
pip install sphinx sphinx_rtd_theme
cd docs
rm -rf source/*
sphinx-apidoc --doc-project "API docs" -o source/ ../memoize
make html
```

## 4.3 PyPi

```
# build dist
python3 -m pip install --user --upgrade setuptools wheel
python3 setup.py sdist bdist_wheel

# try package
python3 -m pip install --user --upgrade twine
python3 -m twine check dist/*

# actual upload will be done by Travis
```

CHAPTER 5

# API docs

## 5.1 memoize package

### 5.1.1 Submodules

### 5.1.2 memoize.coerced module

[Internal use only] Some functions work differently way depending if asyncio or Tornado is used - this is resolved here.

### 5.1.3 memoize.configuration module

[API] Provides interface (and built-in implementations) of full cache configuration.

**class** `memoize.configuration.`**`CacheConfiguration`**

Bases: `object`

Provides configuration for cache.

**`configured`**`()` → bool

Cache will raise NotConfiguredCacheCalledException if this returns false. May be useful if when cache is reconfigured in runtime.

**`entry_builder`**`()` → memoize.entrybuilder.CacheEntryBuilder

Determines which CacheEntryBuilder is to be used by cache.

**`eviction_strategy`**`()` → memoize.eviction.EvictionStrategy

Determines which EvictionStrategy is to be used by cache.

**`key_extractor`**`()` → memoize.key.KeyExtractor

Determines which KeyExtractor is to be used by cache.

**`method_timeout`**`()` → datetime.timedelta

Defines how much time wrapped method can take to complete.

**storage**() → memoize.storage.CacheStorage
Determines which CacheStorage is to be used by cache.

**class** memoize.configuration.**DefaultInMemoryCacheConfiguration**(*capacity: int = 4096, method_timeout: datetime.timedelta = datetime.timedelta(seconds=120), update_after: datetime.timedelta = datetime.timedelta(seconds=600), expire_after: datetime.timedelta = datetime.timedelta(seconds=1800)*)

Bases: *memoize.configuration.CacheConfiguration*

Default parameters that describe in-memory cache. Be ware that parameters used do not suit every case.

**configured**() → bool
Cache will raise NotConfiguredCacheCalledException if this returns false. May be useful if when cache is reconfigured in runtime.

**entry_builder**() → memoize.entrybuilder.ProvidedLifeSpanCacheEntryBuilder
Determines which CacheEntryBuilder is to be used by cache.

**eviction_strategy**() → memoize.eviction.LeastRecentlyUpdatedEvictionStrategy
Determines which EvictionStrategy is to be used by cache.

**key_extractor**() → memoize.key.EncodedMethodReferenceAndArgsKeyExtractor
Determines which KeyExtractor is to be used by cache.

**method_timeout**() → datetime.timedelta
Defines how much time wrapped method can take to complete.

**storage**() → memoize.storage.LocalInMemoryCacheStorage
Determines which CacheStorage is to be used by cache.

**class** memoize.configuration.**MutableCacheConfiguration**(*configured: bool, storage: memoize.storage.CacheStorage, key_extractor: memoize.key.KeyExtractor, eviction_strategy: memoize.eviction.EvictionStrategy, entry_builder: memoize.entrybuilder.CacheEntryBuilder, method_timeout: datetime.timedelta*)

Bases: *memoize.configuration.CacheConfiguration*

Mutable configuration which can be change at runtime. May be also used to customize existing configuration (for example a default one, which is immutable).

**configured**() → bool
Cache will raise NotConfiguredCacheCalledException if this returns false. May be useful if when cache is reconfigured in runtime.

**entry_builder**() → memoize.entrybuilder.CacheEntryBuilder
Determines which CacheEntryBuilder is to be used by cache.

**eviction_strategy**() → memoize.eviction.EvictionStrategy
Determines which EvictionStrategy is to be used by cache.

**static initialized_with**(*configuration: memoize.configuration.CacheConfiguration*) → memoize.configuration.MutableCacheConfiguration

**key_extractor**() → memoize.key.KeyExtractor
Determines which KeyExtractor is to be used by cache.

**method_timeout**() → datetime.timedelta
Defines how much time wrapped method can take to complete.

**set_configured**(*value: bool*) → memoize.configuration.MutableCacheConfiguration

**set_entry_builder**(*value: memoize.entrybuilder.CacheEntryBuilder*) → memoize.configuration.MutableCacheConfiguration

**set_eviction_strategy**(*value: memoize.eviction.EvictionStrategy*) → memoize.configuration.MutableCacheConfiguration

**set_key_extractor**(*value: memoize.key.KeyExtractor*) → memoize.configuration.MutableCacheConfiguration

**set_method_timeout**(*value: datetime.timedelta*) → memoize.configuration.MutableCacheConfiguration

**set_storage**(*value: memoize.storage.CacheStorage*) → memoize.configuration.MutableCacheConfiguration

**storage**() → memoize.storage.CacheStorage
Determines which CacheStorage is to be used by cache.

**exception** memoize.configuration.**NotConfiguredCacheCalledException**
Bases: `Exception`

### 5.1.4 memoize.entry module

[Internal use only] Contains implementation of cache entry.

**class** memoize.entry.**CacheEntry**(*created: datetime.datetime*, *update_after: datetime.datetime*, *expires_after: datetime.datetime*, *value: Any*)
Bases: `object`

Implementation of cache entry used internally

### 5.1.5 memoize.entrybuilder module

[API] Provides interface (and built-in implementations) how cache entries should be constructed (responsibility of expiry & update after times lies here). This interface is used in cache configuration.

**class** memoize.entrybuilder.**CacheEntryBuilder**
Bases: `object`

**build**(*key: str*, *value: Any*) → memoize.entry.CacheEntry
Constructs cache entry object (sets creation time, can transform value, governs update/expiration times).

**class** memoize.entrybuilder.**ProvidedLifeSpanCacheEntryBuilder**(*update_after: datetime.timedelta = datetime.timedelta(seconds=600), expire_after: datetime.timedelta = datetime.timedelta(seconds=1800)*)

Bases: *memoize.entrybuilder.CacheEntryBuilder*

CacheEntryBuilder which uses constant delays independent form values that are cached

**build**(*key: str*, *value: Any*) → memoize.entry.CacheEntry

Constructs cache entry object (sets creation time, can transform value, governs update/expiration times).

**update_timeouts**(*update_after: datetime.timedelta*, *expire_after: datetime.timedelta*) → None

### 5.1.6 memoize.eviction module

[API] Provides interface (and built-in implementations) how cache entries should be constructed (responsibility of expiry & update after times lies here). This interface is used in cache configuration.

**class** memoize.eviction.**EvictionStrategy**

Bases: object

**mark_read**(*key: str*) → None

Informs strategy that entry related to given key was read by current client.

**mark_released**(*key: str*) → None

Informs strategy that entry related to given key was deemed non-essential by current client.

**mark_written**(*key: str*, *entry: memoize.entry.CacheEntry*) → None

Informs strategy that entry related to given key was updated by current client.

**next_to_release**() → Optional[str]

Returns element that should be released by the current client according to this strategy (or None).

**class** memoize.eviction.**LeastRecentlyUpdatedEvictionStrategy**(*capacity=4096*)

Bases: *memoize.eviction.EvictionStrategy*

**mark_read**(*key: str*) → None

Informs strategy that entry related to given key was read by current client.

**mark_released**(*key: str*) → None

Informs strategy that entry related to given key was deemed non-essential by current client.

**mark_written**(*key: str*, *entry: memoize.entry.CacheEntry*) → None

Informs strategy that entry related to given key was updated by current client.

**next_to_release**() → Optional[str]

Returns element that should be released by the current client according to this strategy (or None).

**class** memoize.eviction.**NoEvictionStrategy**

Bases: *memoize.eviction.EvictionStrategy*

Strategy to be used when delegating eviction to cache itself. This strategy performs no actions.

**mark_read**(*key: str*) → None

Informs strategy that entry related to given key was read by current client.

**mark_released**(*key: str*) → None

Informs strategy that entry related to given key was deemed non-essential by current client.

**mark_written**(*key: str*, *entry: memoize.entry.CacheEntry*) → None
Informs strategy that entry related to given key was updated by current client.

**next_to_release**() → Optional[str]
Returns element that should be released by the current client according to this strategy (or None).

### 5.1.7 memoize.exceptions module

[API] Contains exceptions that may be exposed to the library client.

**exception** memoize.exceptions.**CachedMethodFailedException**
Bases: Exception

### 5.1.8 memoize.key module

[API] Provides interface (and built-in implementations) how cache keys are constructed. This interface is used in cache configuration.

**class** memoize.key.**EncodedMethodNameAndArgsKeyExtractor**(*skip_first_arg_as_self=False*)
Bases: *memoize.key.KeyExtractor*

Encodes method name, args & kwargs to string and uses that as cache entry key. This KeyExtractor is class-centric and creates same keys for all objects of the same type.

Note: If wrapped function is a method (has 'self' as first positional arg) you may want to exclude 'self' from key by setting 'skip_first_arg_as_self' flag. For static methods of ordinary functions flag should be set to 'False'.

Warning: uses method name only, so be cautious and do not wrap methods of different classes with the same names while using same store and 'skip_first_arg_as_self' set to False.

**format_key**(*method_reference, call_args: Tuple[Any, ...], call_kwargs: Dict[str, Any]*) → str
Using wrapped method object, call args and call keyword args, prepare cache entry key.

**class** memoize.key.**EncodedMethodReferenceAndArgsKeyExtractor**
Bases: *memoize.key.KeyExtractor*

Encodes method reference, args & kwargs to string and uses that as cache entry key. This KeyExtractor is object-centric and creates different keys for different objects of the same type (so when you create new objects - for instance after app restart - old entries in external store like Redis will be unreachable).

**format_key**(*method_reference, call_args: Tuple[Any, ...], call_kwargs: Dict[str, Any]*) → str
Using wrapped method object, call args and call keyword args, prepare cache entry key.

**class** memoize.key.**KeyExtractor**
Bases: object

Provides logic of cache key construction.

**format_key**(*method_reference, call_args: Tuple[Any, ...], call_kwargs: Dict[str, Any]*) → str
Using wrapped method object, call args and call keyword args, prepare cache entry key.

### 5.1.9 memoize.memoize_configuration module

[API] Provides global config of the library. Translates environment variables into values used internally by the library.

### 5.1.10 memoize.serde module

[API] Provides interface (and built-in implementations) of SerDe that may be used to implement cache storage.

**class** memoize.serde.**EncodingSerDe**(*serde: memoize.serde.SerDe*, *binary_encoding: str = 'zip'*)

Bases: *memoize.serde.SerDe*

Applies extra encoding to the data (for instance compression when 'zip' or 'bz2' codec used).

**deserialize**(*value: bytes*) → memoize.entry.CacheEntry

**serialize**(*value: memoize.entry.CacheEntry*) → bytes

**class** memoize.serde.**JsonSerDe**(*string_encoding: str = 'utf-8', value_to_reversible_repr: Callable[[Any], Any] = <function JsonSerDe.<lambda>>, reversible_repr_to_value: Callable[[Any], Any] = <function JsonSerDe.<lambda>>*)

Bases: *memoize.serde.SerDe*

Uses encoded json string as binary representation. Value of cached type should consist of types which are json-reversible (json.loads(json.dumps(v)) is equal to v) or one should provide (by constructor) functions converting values to/from such representation.

**deserialize**(*data: bytes*) → memoize.entry.CacheEntry

**serialize**(*entry: memoize.entry.CacheEntry*) → bytes

**class** memoize.serde.**PickleSerDe**(*pickle_protocol=4*)

Bases: *memoize.serde.SerDe*

Uses encoded pickles as binary representation.

**deserialize**(*data: bytes*) → memoize.entry.CacheEntry

**serialize**(*entry: memoize.entry.CacheEntry*) → bytes

**class** memoize.serde.**SerDe**

Bases: object

Responsible for (de)serialization of values handled by CacheStorage (if SerDes are supported).

**deserialize**(*data: bytes*) → memoize.entry.CacheEntry

**serialize**(*entry: memoize.entry.CacheEntry*) → bytes

### 5.1.11 memoize.statuses module

[Internal use only] Encapsulates update state management.

**class** memoize.statuses.**UpdateStatuses**(*update_lock_timeout: datetime.timedelta = datetime.timedelta(seconds=300)*)

Bases: object

**await_updated**(*key: str*) → Awaitable[Optional[memoize.entry.CacheEntry]]

Waits (asynchronously) until update in progress has benn finished. Returns updated entry or None if update failed/timed-out. Should be called only if 'is_being_updated' returned True (and since then IO-loop has not been lost).

**is_being_updated**(*key: str*) → bool

Checks if update for given key is in progress. Obtained info is valid until control gets back to IO-loop.

**mark_being_updated**(*key: str*) → None
Informs that update has been started. Should be called only if 'is_being_updated' returned False (and since then IO-loop has not been lost).. Calls to 'is_being_updated' will return True until 'mark_updated' will be called.

**mark_update_aborted**(*key: str*) → None
Informs that update failed to complete. Calls to 'is_being_updated' will return False until 'mark_being_updated' will be called.

**mark_updated**(*key: str*, *entry: memoize.entry.CacheEntry*) → None
Informs that update has been finished. Calls to 'is_being_updated' will return False until 'mark_being_updated' will be called.

### 5.1.12 memoize.storage module

[API] Provides interface (and built-in implementations) of storage for cache entries. This interface is used in cache configuration.

**class** memoize.storage.**CacheStorage**
Bases: object

**get**(*key: str*) → Optional[memoize.entry.CacheEntry]
Request value for given key. If currently there is no such value, returns None. Has to be async.

**offer**(*key: str*, *entry: memoize.entry.CacheEntry*) → None
Offer entry to be stored. If storage already has more relevant data, offer may be declined. Has to be async.

**release**(*key: str*) → None
Declare that current client does not need entry determined by given key. Has to be async.

**class** memoize.storage.**LocalInMemoryCacheStorage**
Bases: *memoize.storage.CacheStorage*

Implementation that stores all entries as-is in a dictionary residing solely in memory.

**get**(*key: str*) → Optional[memoize.entry.CacheEntry]
Request value for given key. If currently there is no such value, returns None. Has to be async.

**offer**(*key: str*, *entry: memoize.entry.CacheEntry*) → None
Offer entry to be stored. If storage already has more relevant data, offer may be declined. Has to be async.

**release**(*key: str*) → None
Declare that current client does not need entry determined by given key. Has to be async.

### 5.1.13 memoize.wrapper module

[API] Provides an entry point to the library - a wrapper that is used to cache entries.

memoize.wrapper.**memoize**(*method: Optional[Callable] = None*, *configuration: memoize.configuration.CacheConfiguration = None*, *invalidation: memoize.invalidation.InvalidationSupport = None*)
Wraps function with memoization.

If entry reaches time it should be updated, refresh is performed in background, but current entry is still valid and may be returned. Once expiration time is reached, refresh is blocking and current entry is considered invalid.

Note: If wrapped method times out after *method_timeout* (see configuration) the cache will not be populated and a failure occurs.

Note: If wrapped method throws an exception the cache will not be populated and failure occurs.

Note: Failures are indicated by designated exceptions (not original ones).

To force refreshing immediately upon call to a cached method, set 'force_refresh_memoized' keyword flag, so the method will block until it's cache is refreshed.

Warning: Leaving default configuration is a bad idea as it may not fit your data (may cause OOMs or cache for an inappropriate time).

**Parameters**

- **method** (*function*) – function to be decorated
- **configuration** (*CacheConfiguration*) – cache configuration; default: DefaultInMemoryCacheConfiguration
- **invalidation** (*InvalidationSupport*) – pass created instance of InvalidationSupport to have it configured

**Raises** CachedMethodFailedException upon call: if cached method timed-out or thrown an exception

**Raises** NotConfiguredCacheCalledException upon call: if provided configuration is not ready

### 5.1.14 Module contents

CHAPTER 6

# Changelog

## 6.1 1.1.3

- Fixed declared supported python versions (*classifiers* in *setup.py*).

## 6.2 1.1.2

- **Added support for Python 3.10:**
  - Applied workaround for testing setup (*MutableMapping* alias required by *tornado.testing.gen_test*);
  - Updated *mypy* and *coverage* setup to Python 3.10.
- Added support for Python 3.11 (provisional as only alpha releases of 3.11 were tested).

## 6.3 1.1.1

- Fixed parallel queries returning expired entries (while expired entry was being refreshed).

## 6.4 1.1.0

- Added support for manual invalidation.
- Added customization of basic params for DefaultInMemoryCacheConfiguration.
- Included initial mypy checking (tornado/asyncio interoperability forces some ignores).

## 6.5 1.0.4

- Added support for Python 3.9.

## 6.6 1.0.3

- Fixed unhandled KeyError in UpdateStatuses.

## 6.7 1.0.2

- Added support for Python 3.8.

## 6.8 1.0.1

- Removed unintended dependency on tornado (it prevented asyncio mode working without tornado installed).

## 6.9 1.0.0

- Initial public release.

CHAPTER 7

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## m

memoize, 22
memoize.coerced, 15
memoize.configuration, 15
memoize.entry, 17
memoize.entrybuilder, 17
memoize.eviction, 18
memoize.exceptions, 19
memoize.key, 19
memoize.memoize_configuration, 19
memoize.serde, 20
memoize.statuses, 20
memoize.storage, 21
memoize.wrapper, 21

# Index


## A

`await_updated()` (*memoize.statuses.UpdateStatuses method*), 20

## B

`build()` (*memoize.entrybuilder.CacheEntryBuilder method*), 17
`build()` (*memoize.entrybuilder.ProvidedLifeSpanCacheEntryBuilder method*), 18

## C

`CacheConfiguration` (*class in memoize.configuration*), 15
`CachedMethodFailedException`, 19
`CacheEntry` (*class in memoize.entry*), 17
`CacheEntryBuilder` (*class in memoize.entrybuilder*), 17
`CacheStorage` (*class in memoize.storage*), 21
`configured()` (*memoize.configuration.CacheConfiguration method*), 15
`configured()` (*memoize.configuration.DefaultInMemoryCacheConfiguration method*), 16
`configured()` (*memoize.configuration.MutableCacheConfiguration method*), 16

## D

`DefaultInMemoryCacheConfiguration` (*class in memoize.configuration*), 16
`deserialize()` (*memoize.serde.EncodingSerDe method*), 20
`deserialize()` (*memoize.serde.JsonSerDe method*), 20
`deserialize()` (*memoize.serde.PickleSerDe method*), 20
`deserialize()` (*memoize.serde.SerDe method*), 20

## E

`EncodedMethodNameAndArgsKeyExtractor` (*class in memoize.key*), 19
`EncodedMethodReferenceAndArgsKeyExtractor` (*class in memoize.key*), 19
`EncodingSerDe` (*class in memoize.serde*), 20
`entry_builder()` (*memoize.configuration.CacheConfiguration method*), 15
`entry_builder()` (*memoize.configuration.DefaultInMemoryCacheConfiguration method*), 16
`entry_builder()` (*memoize.configuration.MutableCacheConfiguration method*), 17
`eviction_strategy()` (*memoize.configuration.CacheConfiguration method*), 15
`eviction_strategy()` (*memoize.configuration.DefaultInMemoryCacheConfiguration method*), 16
`eviction_strategy()` (*memoize.configuration.MutableCacheConfiguration method*), 17
`EvictionStrategy` (*class in memoize.eviction*), 18

## F

`format_key()` (*memoize.key.EncodedMethodNameAndArgsKeyExtractor method*), 19
`format_key()` (*memoize.key.EncodedMethodReferenceAndArgsKeyExtractor method*), 19
`format_key()` (*memoize.key.KeyExtractor method*), 19

## G

`get()` (*memoize.storage.CacheStorage method*), 21
`get()` (*memoize.storage.LocalInMemoryCacheStorage method*), 21

## I


initialized_with() (*memoize.configuration.MutableCacheConfiguration static method*), 17
is_being_updated() (*memoize.statuses.UpdateStatuses method*), 20


## J


JsonSerDe (*class in memoize.serde*), 20


## K


key_extractor() (*memoize.configuration.CacheConfiguration method*), 15
key_extractor() (*memoize.configuration.DefaultInMemoryCacheConfiguration method*), 16
key_extractor() (*memoize.configuration.MutableCacheConfiguration method*), 17
KeyExtractor (*class in memoize.key*), 19


## L


LeastRecentlyUpdatedEvictionStrategy (*class in memoize.eviction*), 18
LocalInMemoryCacheStorage (*class in memoize.storage*), 21


## M


mark_being_updated() (*memoize.statuses.UpdateStatuses method*), 20
mark_read() (*memoize.eviction.EvictionStrategy method*), 18
mark_read() (*memoize.eviction.LeastRecentlyUpdatedEvictionStrategy method*), 18
mark_read() (*memoize.eviction.NoEvictionStrategy method*), 18
mark_released() (*memoize.eviction.EvictionStrategy method*), 18
mark_released() (*memoize.eviction.LeastRecentlyUpdatedEvictionStrategy method*), 18
mark_released() (*memoize.eviction.NoEvictionStrategy method*), 18
mark_update_aborted() (*memoize.statuses.UpdateStatuses method*), 21
mark_updated() (*memoize.statuses.UpdateStatuses method*), 21
mark_written() (*memoize.eviction.EvictionStrategy method*), 18
mark_written() (*memoize.eviction.LeastRecentlyUpdatedEvictionStrategy method*), 18
mark_written() (*memoize.eviction.NoEvictionStrategy method*), 18
memoize (*module*), 22
memoize() (*in module memoize.wrapper*), 21
memoize.coerced (*module*), 15
memoize.configuration (*module*), 15
memoize.entry (*module*), 17
memoize.entrybuilder (*module*), 17
memoize.eviction (*module*), 18
memoize.exceptions (*module*), 19
memoize.key (*module*), 19
memoize.memoize_configuration (*module*), 19
memoize.serde (*module*), 20
memoize.statuses (*module*), 20
memoize.storage (*module*), 21
memoize.wrapper (*module*), 21
method_timeout() (*memoize.configuration.CacheConfiguration method*), 15
method_timeout() (*memoize.configuration.DefaultInMemoryCacheConfiguration method*), 16
method_timeout() (*memoize.configuration.MutableCacheConfiguration method*), 17
MutableCacheConfiguration (*class in memoize.configuration*), 16


## N


next_to_release() (*memoize.eviction.EvictionStrategy method*), 18
next_to_release() (*memoize.eviction.LeastRecentlyUpdatedEvictionStrategy method*), 18
next_to_release() (*memoize.eviction.NoEvictionStrategy method*), 19
NoEvictionStrategy (*class in memoize.eviction*), 18
NotConfiguredCacheCalledException, 17


## O


offer() (*memoize.storage.CacheStorage method*), 21
offer() (*memoize.storage.LocalInMemoryCacheStorage method*), 21


## P


PickleSerDe (*class in memoize.serde*), 20
ProvidedLifeSpanCacheEntryBuilder (*class in memoize.entrybuilder*), 17

## R


`release()` (*memoize.storage.CacheStorage method*), 21
`release()` (*memoize.storage.LocalInMemoryCacheStorage method*), 21


## S


`SerDe` (*class in memoize.serde*), 20
`serialize()` (*memoize.serde.EncodingSerDe method*), 20
`serialize()` (*memoize.serde.JsonSerDe method*), 20
`serialize()` (*memoize.serde.PickleSerDe method*), 20
`serialize()` (*memoize.serde.SerDe method*), 20
`set_configured()` (*memoize.configuration.MutableCacheConfiguration method*), 17
`set_entry_builder()` (*memoize.configuration.MutableCacheConfiguration method*), 17
`set_eviction_strategy()` (*memoize.configuration.MutableCacheConfiguration method*), 17
`set_key_extractor()` (*memoize.configuration.MutableCacheConfiguration method*), 17
`set_method_timeout()` (*memoize.configuration.MutableCacheConfiguration method*), 17
`set_storage()` (*memoize.configuration.MutableCacheConfiguration method*), 17
`storage()` (*memoize.configuration.CacheConfiguration method*), 15
`storage()` (*memoize.configuration.DefaultInMemoryCacheConfiguration method*), 16
`storage()` (*memoize.configuration.MutableCacheConfiguration method*), 17


## U


`update_timeouts()` (*memoize.entrybuilder.ProvidedLifeSpanCacheEntryBuilder method*), 18
`UpdateStatuses` (*class in memoize.statuses*), 20