cacheout Documentation

Release 0.11.2

Derrick Gilland

Oct 28, 2020

Contents

A caching library for Python.

CHAPTER 1

Links

- Project: https://github.com/dgilland/cacheout
- Documentation: https://cacheout.readthedocs.io
- PyPI: https://pypi.python.org/pypi/cacheout/
- TravisCI: https://travis-ci.org/dgilland/cacheout

CHAPTER 2

Features

- In-memory caching using dictionary backend
- Cache manager for easily accessing multiple cache objects
- Reconfigurable cache settings for runtime setup when using module-level cache objects
- Maximum cache size enforcement
- Default cache TTL (time-to-live) as well as custom TTLs per cache entry
- Bulk set, get, and delete operations
- Bulk get and delete operations filtered by string, regex, or function
- Memoization decorators
- Thread safe
- Multiple cache implementations:
  - FIFO (First In, First Out)
  - LIFO (Last In, First Out)
  - LRU (Least Recently Used)
  - MRU (Most Recently Used)
  - LFU (Least Frequently Used)
  - RR (Random Replacement)

CHAPTER 3

Roadmap

- Layered caching (multi-level caching)
- Cache event listener support (e.g. on-get, on-set, on-delete)
- Cache statistics (e.g. cache hits/misses, cache frequency, etc)

CHAPTER 4

Requirements

- Python >= 3.4

CHAPTER 5

Quickstart

Install using pip:

```
pip install cacheout
```

Let's start with some basic caching by creating a cache object:

```
from cacheout import Cache

cache = Cache()
```

By default the `cache` object will have a maximum size of `256` and default TTL expiration turned off. These values can be set with:

```
cache = Cache(maxsize=256, ttl=0, timer=time.time, default=None)  # defaults
```

Set a cache key using `cache.set()`:

```
cache.set(1, 'foobar')
```

Get the value of a cache key with `cache.get()`:

```
assert cache.get(1) == 'foobar'
```

Get a default value when cache key isn't set:

```
assert cache.get(2) is None
assert cache.get(2, default=False) is False
assert 2 not in cache
```

Provide cache values using a default callable:

```
assert 2 not in cache
assert cache.get(2, default=lambda key: key) == 2
assert cache.get(2) == 2
assert 2 in cache
```

Provide a global default:

```
cache2 = Cache(default=True)
assert cache2.get('missing') is True
assert 'missing' not in cache2

cache3 = Cache(default=lambda key: key)
assert cache3.get('missing') == 'missing'
assert 'missing' in cache3
```

Set the TTL (time-to-live) expiration per entry:

```
cache.set(3, {'data': {}}, ttl=1)
assert cache.get(3) == {'data': {}}
time.sleep(1)
assert cache.get(3) is None
```

Memoize a function where cache keys are generated from the called function parameters:

```
@cache.memoize()
def func(a, b):
    pass
```

Provide a TTL for the memoized function and incorporate argument types into generated cache keys:

```
@cache.memoize(ttl=5, typed=True)
def func(a, b):
    pass

# func(1, 2) has different cache key than func(1.0, 2.0), whereas,
# with "typed=False" (the default), they would have the same key
```

Access the original memoized function:

```
@cache.memoize()
def func(a, b):
    pass

func.uncached(1, 2)
```

Get a copy of the entire cache with `cache.copy()`:

```
assert cache.copy() == {1: 'foobar', 2: ('foo', 'bar', 'baz')}
```

Delete a cache key with `cache.delete()`:

```
cache.delete(1)
assert cache.get(1) is None
```

Clear the entire cache with `cache.clear()`:

```
cache.clear()
assert len(cache) == 0
```

Perform bulk operations with `cache.set_many()`, `cache.get_many()`, and `cache.delete_many()`:

```
cache.set_many({'a': 1, 'b': 2, 'c': 3})
assert cache.get_many(['a', 'b', 'c']) == {'a': 1, 'b': 2, 'c': 3}
```

(continues on next page)

(continued from previous page)

```
cache.delete_many(['a', 'b', 'c'])
assert cache.count() == 0
```

Use complex filtering in `cache.get_many()` and `cache.delete_many()`:

```
import re
cache.set_many({'a_1': 1, 'a_2': 2, '123': 3, 'b': 4})

cache.get_many('a_*') == {'a_1': 1, 'a_2': 2}
cache.get_many(re.compile(r'\d')) == {'123': 3}
cache.get_many(lambda key: '2' in key) == {'a_2': 2, '123': 3}

cache.delete_many('a_*')
assert dict(cache.items()) == {'123': 3, 'b': 4}
```

Reconfigure the cache object after creation with `cache.configure()`:

```
cache.configure(maxsize=1000, ttl=5 * 60)
```

Get keys, values, and items from the cache with `cache.keys()`, `cache.values()`, and `cache.items()`:

```
cache.set_many({'a': 1, 'b': 2, 'c': 3})
assert list(cache.keys()) == ['a', 'b', 'c']
assert list(cache.values()) == [1, 2, 3]
assert list(cache.items()) == [('a', 1), ('b', 2), ('c', 3)]
```

Iterate over cache keys:

```
for key in cache:
    print(key, cache.get(key))
    # 'a' 1
    # 'b' 2
    # 'c' 3
```

Check if key exists with `cache.has()` and `key in cache`:

```
assert cache.has('a')
assert 'a' in cache
```

Manage multiple caches using `CacheManager`:

```
from cacheout import CacheManager

cacheman = CacheManager({'a': {'maxsize': 100},
                         'b': {'maxsize': 200, 'ttl': 900},
                         'c': {})

cacheman['a'].set('key1', 'value1')
value = cacheman['a'].get('key')

cacheman['b'].set('key2', 'value2')
assert cacheman['b'].maxsize == 200
assert cacheman['b'].ttl == 900

cacheman['c'].set('key3', 'value3')
```

(continues on next page)

(continued from previous page)

```
cacheman.clear_all()
for name, cache in cacheman:
    assert name in cacheman
    assert len(cache) == 0
```

For more details, see the full documentation at https://cacheout.readthedocs.io.

CHAPTER 6

Guide

6.1 Installation

cacheout requires Python >= 3.4.

To install from PyPI:

```
pip install cacheout
```

6.2 Cache

The cache module provides the *Cache* class which is used as the base for all other cache types.

class cacheout.cache.**Cache**(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)

An in-memory, FIFO cache object that supports:

- Maximum number of cache entries
- Global TTL default
- Per cache entry TTL
- TTL first/non-TTL FIFO cache eviction policy

Cache entries are stored in an `OrderedDict` so that key ordering based on the cache type can be maintained without the need for additional list(s). Essentially, the key order of the `OrderedDict` is treated as an "eviction queue" with the convention that entries at the beginning of the queue are "newer" while the entries at the end are "older" (the exact meaning of "newer" and "older" will vary between different cache types). When cache entries need to be evicted, expired entries are removed first followed by the "older" entries (i.e. the ones at the end of the queue).

maxsize

Maximum size of cache dictionary. Defaults to `256`.

Type int, optional

ttl
Default TTL for all cache entries. Defaults to `0` which means that entries do not expire.

Type int, optional

timer
Timer function to use to calculate TTL expiration. Defaults to `time.time`.

Type callable, optional

default
Default value or function to use in *get()* when key is not found. If callable, it will be passed a single argument, `key`, and its return value will be set for that cache key.

Type mixed, optional

add(*key*, *value*, *ttl=None*)
Add cache key/value if it doesn't already exist. Essentially, this method ignores keys that exist which leaves the original TTL in tact.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to add.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses *ttl*.

add_many(*items*, *ttl=None*)
Add multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

clear()
Clear all cache entries.

configure(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Configure cache settings.

This method is meant to support runtime level configurations for global level cache objects.

copy()
Return a copy of the cache.

Returns OrderedDict

delete(*key*)
Delete cache key and return number of entries deleted (`1` or `0`).

Returns `1` if key was deleted, `0` if key didn't exist.

Return type int

delete_expired()
Delete expired cache keys and return number of entries deleted.

Returns Number of entries deleted.

Return type int

delete_many(*iteratee*)
Delete multiple cache keys at once filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.

Returns Number of cache keys deleted.

Return type int

evict()
Perform cache eviction per the cache replacement policy:

- First, remove **all** expired entries.
- Then, remove non-TTL entries using the cache replacement policy.

When removing non-TTL entries, this method will only remove the minimum number of entries to reduce the number of entries below *maxsize*. If *maxsize* is `0`, then only expired entries will be removed.

Returns Number of cache entries evicted.

Return type int

expire_times()
Return cache expirations for TTL keys.

Returns dict

expired(*key*, *expires_on=None*)
Return whether cache key is expired or not.

Parameters

- **key** (*mixed*) – Cache key.
- **expires_on** (*float, optional*) – Timestamp of when the key is considered expired. Defaults to `None` which uses the current value returned from *timer()*.

Returns bool

full()
Return whether the cache is full or not.

Returns bool

get(*key*, *default=None*)
Return the cache value for *key* or *default* or `missing(key)` if it doesn't exist or has expired.

Parameters

- **key** (*mixed*) – Cache key.
- **default** (*mixed, optional*) – Value to return if *key* doesn't exist. If any value other than `None`, then it will take precendence over `missing` and be used as the return value. If *default* is callable, it will function like `missing` and its return value will be set for the cache *key*. Defaults to `None`.

Returns The cached value.

Return type mixed

get_many(*iteratee*, *default=None*)
Return many cache values as a `dict` of key/value pairs filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters

- **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.
- **default** (*mixed, optional*) – Value to return if key doesn't exist. Defaults to `None`.

Returns dict

has(*key*)
Return whether cache key exists and hasn't expired.

Returns bool

items()
Return a `dict_items` view of cache items.

> **Warning:** Returned data is copied from the cache object, but any modifications to mutable values will modify this cache object's data.

Returns dict_items

keys()
Return `dict_keys` view of all cache keys.

> **Note:** Cache is copied from the underlying cache storage before returning.

Returns dict_keys

memoize(*, *ttl=None*, *typed=False*)
Decorator that wraps a function with a memoizing callable and works on both synchronous and asynchronous functions.

Each return value from the function will be cached using the function arguments as the cache key. The cache object can be accessed at `<function>.cache`. The uncached version (i.e. the original function) can be accessed at `<function>.uncached`. Each return value from the function will be cached using the function arguments as the cache key.

Keyword Arguments

- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses *ttl*.
- **typed** (*bool, optional*) – Whether to cache arguments of a different type separately. For example, `<function>(1)` and `<function>(1.0)` would be treated differently. Defaults to `False`.

popitem()
Delete and return next cache item, `(key, value)`, based on cache replacement policy while ignoring expiration times (i.e. the selection of the item to pop is based soley on the cache key ordering).

Returns Two-element tuple of deleted cache `(key, value)`.

Return type tuple

set(*key*, *value*, *ttl=None*)
Set cache key/value and replace any previously set cache key. If the cache key previous existed, setting it will move it to the end of the cache stack which means it would be evicted last.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to set.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses *ttl*.

set_many(*items*, *ttl=None*)
Set multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

size()
Return number of cache entries.

values()
Return `dict_values` view of all cache values.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_values

6.3 FIFO Cache

The fifo module provides the *FIFOCache* (First-In, First-Out) class.

Note: The `FIOCache` is an alias to *Cache* since *Cache* implements FIFO. It is provided as a standard name based on its cache replacement policy.

class cacheout.fifo.**FIFOCache**(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Bases: *cacheout.cache.Cache*

The First In, First Out (FIFO) cache is an alias of *Cache* since *Cache* implements FIFO. It is provided as a standard name based on its cache replacement policy.

add(*key*, *value*, *ttl=None*)
Add cache key/value if it doesn't already exist. Essentially, this method ignores keys that exist which leaves the original TTL in tact.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to add.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

add_many(*items*, *ttl=None*)
Add multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

clear()
Clear all cache entries.

configure(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Configure cache settings.

This method is meant to support runtime level configurations for global level cache objects.

copy()
Return a copy of the cache.

Returns OrderedDict

delete(*key*)
Delete cache key and return number of entries deleted (`1` or `0`).

Returns `1` if key was deleted, `0` if key didn't exist.

Return type int

delete_expired()
Delete expired cache keys and return number of entries deleted.

Returns Number of entries deleted.

Return type int

delete_many(*iteratee*)
Delete multiple cache keys at once filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.

Returns Number of cache keys deleted.

Return type int

evict()
Perform cache eviction per the cache replacement policy:

- First, remove **all** expired entries.

- Then, remove non-TTL entries using the cache replacement policy.

When removing non-TTL entries, this method will only remove the minimum number of entries to reduce the number of entries below `maxsize`. If `maxsize` is `0`, then only expired entries will be removed.

Returns Number of cache entries evicted.

Return type int

expire_times()
Return cache expirations for TTL keys.

Returns dict

expired(*key*, *expires_on=None*)
Return whether cache key is expired or not.

Parameters

- **key** (*mixed*) – Cache key.
- **expires_on** (*float, optional*) – Timestamp of when the key is considered expired. Defaults to `None` which uses the current value returned from `timer()`.

Returns bool

full()
Return whether the cache is full or not.

Returns bool

get(*key*, *default=None*)
Return the cache value for *key* or *default* or `missing(key)` if it doesn't exist or has expired.

Parameters

- **key** (*mixed*) – Cache key.
- **default** (*mixed, optional*) – Value to return if *key* doesn't exist. If any value other than `None`, then it will take precendence over `missing` and be used as the return value. If *default* is callable, it will function like `missing` and its return value will be set for the cache *key*. Defaults to `None`.

Returns The cached value.

Return type mixed

get_many(*iteratee*, *default=None*)
Return many cache values as a `dict` of key/value pairs filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters

- **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.
- **default** (*mixed, optional*) – Value to return if key doesn't exist. Defaults to `None`.

Returns dict

has(*key*)
Return whether cache key exists and hasn't expired.

Returns bool

items()
Return a `dict_items` view of cache items.

> **Warning:** Returned data is copied from the cache object, but any modifications to mutable values will modify this cache object's data.

Returns dict_items

keys()
Return `dict_keys` view of all cache keys.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_keys

memoize(*, *ttl=None*, *typed=False*)
Decorator that wraps a function with a memoizing callable and works on both synchronous and asynchronous functions.

Each return value from the function will be cached using the function arguments as the cache key. The cache object can be accessed at `<function>.cache`. The uncached version (i.e. the original function) can be accessed at `<function>.uncached`. Each return value from the function will be cached using the function arguments as the cache key.

Keyword Arguments

- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.
- **typed** (*bool, optional*) – Whether to cache arguments of a different type separately. For example, `<function>(1)` and `<function>(1.0)` would be treated differently. Defaults to `False`.

popitem()
Delete and return next cache item, `(key, value)`, based on cache replacement policy while ignoring expiration times (i.e. the selection of the item to pop is based soley on the cache key ordering).

Returns Two-element tuple of deleted cache `(key, value)`.

Return type tuple

set(*key*, *value*, *ttl=None*)
Set cache key/value and replace any previously set cache key. If the cache key previous existed, setting it will move it to the end of the cache stack which means it would be evicted last.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to set.

- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

set_many(*items*, *ttl=None*)
Set multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

size()
Return number of cache entries.

values()
Return `dict_values` view of all cache values.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_values

6.4 LIFO Cache

The lifo module provides the *LIFOCache* (Last-In, First-Out) class.

class cacheout.lifo.**LIFOCache**(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Bases: *cacheout.cache.Cache*

The Last-In, First-Out (LIFO) cache is like *Cache* but uses a last-in, first-out replacement policy.

The primary difference with *Cache* is that cache entries are evicted from the end of the eviction queue first instead of evicting from the beginning, i.e., the last entry that was added to the cache is the first entry to be removed.

add(*key*, *value*, *ttl=None*)
Add cache key/value if it doesn't already exist. Essentially, this method ignores keys that exist which leaves the original TTL in tact.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to add.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

add_many(*items*, *ttl=None*)
Add multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

clear()
Clear all cache entries.

configure(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Configure cache settings.

This method is meant to support runtime level configurations for global level cache objects.

copy()
Return a copy of the cache.

Returns OrderedDict

delete(*key*)
Delete cache key and return number of entries deleted (`1` or `0`).

Returns `1` if key was deleted, `0` if key didn't exist.

Return type int

delete_expired()
Delete expired cache keys and return number of entries deleted.

Returns Number of entries deleted.

Return type int

delete_many(*iteratee*)
Delete multiple cache keys at once filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.

Returns Number of cache keys deleted.

Return type int

evict()
Perform cache eviction per the cache replacement policy:

- First, remove **all** expired entries.
- Then, remove non-TTL entries using the cache replacement policy.

When removing non-TTL entries, this method will only remove the minimum number of entries to reduce the number of entries below `maxsize`. If `maxsize` is `0`, then only expired entries will be removed.

Returns Number of cache entries evicted.

Return type int

expire_times()
Return cache expirations for TTL keys.

Returns dict

expired(*key*, *expires_on=None*)
Return whether cache key is expired or not.

Parameters

- **key** (*mixed*) – Cache key.

- **expires_on** (*float, optional*) – Timestamp of when the key is considered expired. Defaults to `None` which uses the current value returned from `timer()`.

Returns bool

full()
Return whether the cache is full or not.

Returns bool

get(*key*, *default=None*)
Return the cache value for *key* or *default* or `missing(key)` if it doesn't exist or has expired.

Parameters

- **key** (*mixed*) – Cache key.
- **default** (*mixed, optional*) – Value to return if *key* doesn't exist. If any value other than `None`, then it will take precendence over `missing` and be used as the return value. If *default* is callable, it will function like `missing` and its return value will be set for the cache *key*. Defaults to `None`.

Returns The cached value.

Return type mixed

get_many(*iteratee*, *default=None*)
Return many cache values as a `dict` of key/value pairs filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters

- **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.
- **default** (*mixed, optional*) – Value to return if key doesn't exist. Defaults to `None`.

Returns dict

has(*key*)
Return whether cache key exists and hasn't expired.

Returns bool

items()
Return a `dict_items` view of cache items.

> **Warning:** Returned data is copied from the cache object, but any modifications to mutable values will modify this cache object's data.

Returns dict_items

keys()
Return `dict_keys` view of all cache keys.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_keys

memoize(*, *ttl=None*, *typed=False*)

Decorator that wraps a function with a memoizing callable and works on both synchronous and asynchronous functions.

Each return value from the function will be cached using the function arguments as the cache key. The cache object can be accessed at `<function>.cache`. The uncached version (i.e. the original function) can be accessed at `<function>.uncached`. Each return value from the function will be cached using the function arguments as the cache key.

Keyword Arguments

- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.
- **typed** (*bool, optional*) – Whether to cache arguments of a different type separately. For example, `<function>(1)` and `<function>(1.0)` would be treated differently. Defaults to `False`.

popitem()

Delete and return next cache item, `(key, value)`, based on cache replacement policy while ignoring expiration times (i.e. the selection of the item to pop is based soley on the cache key ordering).

Returns Two-element tuple of deleted cache `(key, value)`.

Return type tuple

set(*key*, *value*, *ttl=None*)

Set cache key/value and replace any previously set cache key. If the cache key previous existed, setting it will move it to the end of the cache stack which means it would be evicted last.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to set.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

set_many(*items*, *ttl=None*)

Set multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

size()

Return number of cache entries.

values()

Return `dict_values` view of all cache values.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_values

6.5 LRU Cache

The lru module provides the *LRUCache* (Least Recently Used) class.

class cacheout.lru.**LRUCache**(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Bases: *cacheout.cache.Cache*

Like *Cache* but uses a least-recently-used eviction policy.

The primary difference with *Cache* is that cache entries are moved to the end of the eviction queue when both *get()* and *set()* are called (as opposed to *Cache* that only moves entries on `set()`.

add(*key*, *value*, *ttl=None*)
Add cache key/value if it doesn't already exist. Essentially, this method ignores keys that exist which leaves the original TTL in tact.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to add.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

add_many(*items*, *ttl=None*)
Add multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

clear()
Clear all cache entries.

configure(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Configure cache settings.

This method is meant to support runtime level configurations for global level cache objects.

copy()
Return a copy of the cache.

Returns OrderedDict

delete(*key*)
Delete cache key and return number of entries deleted (`1` or `0`).

Returns `1` if key was deleted, `0` if key didn't exist.

Return type int

delete_expired()
Delete expired cache keys and return number of entries deleted.

Returns Number of entries deleted.

Return type int

delete_many(*iteratee*)
Delete multiple cache keys at once filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.

Returns Number of cache keys deleted.

Return type int

evict()
Perform cache eviction per the cache replacement policy:

- First, remove **all** expired entries.
- Then, remove non-TTL entries using the cache replacement policy.

When removing non-TTL entries, this method will only remove the minimum number of entries to reduce the number of entries below `maxsize`. If `maxsize` is `0`, then only expired entries will be removed.

Returns Number of cache entries evicted.

Return type int

expire_times()
Return cache expirations for TTL keys.

Returns dict

expired(*key*, *expires_on=None*)
Return whether cache key is expired or not.

Parameters

- **key** (*mixed*) – Cache key.
- **expires_on** (*float, optional*) – Timestamp of when the key is considered expired. Defaults to `None` which uses the current value returned from `timer()`.

Returns bool

full()
Return whether the cache is full or not.

Returns bool

get(*key*, *default=None*)
Return the cache value for *key* or *default* or `missing(key)` if it doesn't exist or has expired.

Parameters

- **key** (*mixed*) – Cache key.
- **default** (*mixed, optional*) – Value to return if *key* doesn't exist. If any value other than `None`, then it will take precendence over `missing` and be used as the return value. If *default* is callable, it will function like `missing` and its return value will be set for the cache *key*. Defaults to `None`.

Returns The cached value.

Return type mixed

get_many(*iteratee*, *default=None*)
Return many cache values as a `dict` of key/value pairs filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters

- **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.
- **default** (*mixed, optional*) – Value to return if key doesn't exist. Defaults to `None`.

Returns dict

has(*key*)
Return whether cache key exists and hasn't expired.

Returns bool

items()
Return a `dict_items` view of cache items.

> **Warning:** Returned data is copied from the cache object, but any modifications to mutable values will modify this cache object's data.

Returns dict_items

keys()
Return `dict_keys` view of all cache keys.

> **Note:** Cache is copied from the underlying cache storage before returning.

Returns dict_keys

memoize(*, *ttl=None*, *typed=False*)
Decorator that wraps a function with a memoizing callable and works on both synchronous and asynchronous functions.

Each return value from the function will be cached using the function arguments as the cache key. The cache object can be accessed at `<function>.cache`. The uncached version (i.e. the original function) can be accessed at `<function>.uncached`. Each return value from the function will be cached using the function arguments as the cache key.

Keyword Arguments

- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.
- **typed** (*bool, optional*) – Whether to cache arguments of a different type separately. For example, `<function>(1)` and `<function>(1.0)` would be treated differently. Defaults to `False`.

popitem()
Delete and return next cache item, `(key, value)`, based on cache replacement policy while ignoring expiration times (i.e. the selection of the item to pop is based soley on the cache key ordering).

Returns Two-element tuple of deleted cache `(key, value)`.

Return type tuple

set(*key*, *value*, *ttl=None*)
Set cache key/value and replace any previously set cache key. If the cache key previous existed, setting it will move it to the end of the cache stack which means it would be evicted last.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to set.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

set_many(*items*, *ttl=None*)
Set multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

size()
Return number of cache entries.

values()
Return `dict_values` view of all cache values.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_values

6.6 MRU Cache

The mru module provides the *MRUCache* (Most Recently Used) class.

class cacheout.mru.**MRUCache**(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Bases: *cacheout.lru.LRUCache*

The Most Recently Used (MRU) cache is like *Cache* and *LRUCache* but uses a most-recently-used replacement policy.

The primary difference with *Cache* is that cache entries are moved to the end of the eviction queue when both *get()* and *set()* are called (as opposed to *Cache* that only moves entries on `set()`.

The primary difference with *LRUCache* is that cache entries are evicted from the end of the eviction queue first instead of evicting from the beginning.

add(*key*, *value*, *ttl=None*)
Add cache key/value if it doesn't already exist. Essentially, this method ignores keys that exist which leaves the original TTL in tact.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to add.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

add_many(*items*, *ttl=None*)
Add multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

clear()
Clear all cache entries.

configure(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Configure cache settings.

This method is meant to support runtime level configurations for global level cache objects.

copy()
Return a copy of the cache.

Returns OrderedDict

delete(*key*)
Delete cache key and return number of entries deleted (`1` or `0`).

Returns `1` if key was deleted, `0` if key didn't exist.

Return type int

delete_expired()
Delete expired cache keys and return number of entries deleted.

Returns Number of entries deleted.

Return type int

delete_many(*iteratee*)
Delete multiple cache keys at once filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.

Returns Number of cache keys deleted.

Return type int

evict()
Perform cache eviction per the cache replacement policy:

- First, remove **all** expired entries.

- Then, remove non-TTL entries using the cache replacement policy.

When removing non-TTL entries, this method will only remove the minimum number of entries to reduce the number of entries below `maxsize`. If `maxsize` is `0`, then only expired entries will be removed.

Returns Number of cache entries evicted.

Return type int

expire_times()
Return cache expirations for TTL keys.

Returns dict

expired(*key*, *expires_on=None*)
Return whether cache key is expired or not.

Parameters

- **key** (*mixed*) – Cache key.
- **expires_on** (*float, optional*) – Timestamp of when the key is considered expired. Defaults to `None` which uses the current value returned from `timer()`.

Returns bool

full()
Return whether the cache is full or not.

Returns bool

get(*key*, *default=None*)
Return the cache value for *key* or *default* or `missing(key)` if it doesn't exist or has expired.

Parameters

- **key** (*mixed*) – Cache key.
- **default** (*mixed, optional*) – Value to return if *key* doesn't exist. If any value other than `None`, then it will take precendence over `missing` and be used as the return value. If *default* is callable, it will function like `missing` and its return value will be set for the cache *key*. Defaults to `None`.

Returns The cached value.

Return type mixed

get_many(*iteratee*, *default=None*)
Return many cache values as a `dict` of key/value pairs filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters

- **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.
- **default** (*mixed, optional*) – Value to return if key doesn't exist. Defaults to `None`.

Returns dict

has(*key*)
Return whether cache key exists and hasn't expired.

Returns bool

items()
Return a `dict_items` view of cache items.

> **Warning:** Returned data is copied from the cache object, but any modifications to mutable values will modify this cache object's data.

Returns dict_items

keys()
Return `dict_keys` view of all cache keys.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_keys

memoize(*, *ttl=None*, *typed=False*)
Decorator that wraps a function with a memoizing callable and works on both synchronous and asynchronous functions.

Each return value from the function will be cached using the function arguments as the cache key. The cache object can be accessed at `<function>.cache`. The uncached version (i.e. the original function) can be accessed at `<function>.uncached`. Each return value from the function will be cached using the function arguments as the cache key.

Keyword Arguments

- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.
- **typed** (*bool, optional*) – Whether to cache arguments of a different type separately. For example, `<function>(1)` and `<function>(1.0)` would be treated differently. Defaults to `False`.

popitem()
Delete and return next cache item, `(key, value)`, based on cache replacement policy while ignoring expiration times (i.e. the selection of the item to pop is based soley on the cache key ordering).

Returns Two-element tuple of deleted cache `(key, value)`.

Return type tuple

set(*key*, *value*, *ttl=None*)
Set cache key/value and replace any previously set cache key. If the cache key previous existed, setting it will move it to the end of the cache stack which means it would be evicted last.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to set.

- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

set_many(*items*, *ttl=None*)
Set multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

size()
Return number of cache entries.

values()
Return `dict_values` view of all cache values.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_values

6.7 LFU Cache

The lfu module provides the *LFUCache* (Least Frequently Used) class.

class cacheout.lfu.**LFUCache**(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Bases: *cacheout.cache.Cache*

The Least Frequently Used (LFU) cache is like *Cache* but uses a least-frequently-used eviction policy.

The primary difference with *Cache* is that access to cache entries (i.e. calls to *get()* and *set()*) are tracked; each call to *get()* will increment the cache key's access count while calls to *set()* will reset the counter. During cache eviction, the entry with the lowest access count is removed first.

add(*key*, *value*, *ttl=None*)
Add cache key/value if it doesn't already exist. Essentially, this method ignores keys that exist which leaves the original TTL in tact.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to add.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

add_many(*items*, *ttl=None*)
Add multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

clear()
Clear all cache entries.

configure(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Configure cache settings.

This method is meant to support runtime level configurations for global level cache objects.

copy()
Return a copy of the cache.

Returns OrderedDict

delete(*key*)
Delete cache key and return number of entries deleted (`1` or `0`).

Returns `1` if key was deleted, `0` if key didn't exist.

Return type int

delete_expired()
Delete expired cache keys and return number of entries deleted.

Returns Number of entries deleted.

Return type int

delete_many(*iteratee*)
Delete multiple cache keys at once filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.

Returns Number of cache keys deleted.

Return type int

evict()
Perform cache eviction per the cache replacement policy:

- First, remove **all** expired entries.
- Then, remove non-TTL entries using the cache replacement policy.

When removing non-TTL entries, this method will only remove the minimum number of entries to reduce the number of entries below `maxsize`. If `maxsize` is `0`, then only expired entries will be removed.

Returns Number of cache entries evicted.

Return type int

expire_times()
Return cache expirations for TTL keys.

Returns dict

expired(*key*, *expires_on=None*)
Return whether cache key is expired or not.

Parameters

- **key** (*mixed*) – Cache key.

- **expires_on** (*float, optional*) – Timestamp of when the key is considered expired. Defaults to `None` which uses the current value returned from `timer()`.

Returns bool

full()
Return whether the cache is full or not.

Returns bool

get(*key*, *default=None*)
Return the cache value for *key* or *default* or `missing(key)` if it doesn't exist or has expired.

Parameters

- **key** (*mixed*) – Cache key.
- **default** (*mixed, optional*) – Value to return if *key* doesn't exist. If any value other than `None`, then it will take precendence over `missing` and be used as the return value. If *default* is callable, it will function like `missing` and its return value will be set for the cache *key*. Defaults to `None`.

Returns The cached value.

Return type mixed

get_many(*iteratee*, *default=None*)
Return many cache values as a `dict` of key/value pairs filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters

- **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.
- **default** (*mixed, optional*) – Value to return if key doesn't exist. Defaults to `None`.

Returns dict

has(*key*)
Return whether cache key exists and hasn't expired.

Returns bool

items()
Return a `dict_items` view of cache items.

> **Warning:** Returned data is copied from the cache object, but any modifications to mutable values will modify this cache object's data.

Returns dict_items

keys()
Return `dict_keys` view of all cache keys.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_keys

memoize(*, *ttl=None*, *typed=False*)
Decorator that wraps a function with a memoizing callable and works on both synchronous and asynchronous functions.

Each return value from the function will be cached using the function arguments as the cache key. The cache object can be accessed at `<function>.cache`. The uncached version (i.e. the original function) can be accessed at `<function>.uncached`. Each return value from the function will be cached using the function arguments as the cache key.

Keyword Arguments

- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.
- **typed** (*bool, optional*) – Whether to cache arguments of a different type separately. For example, `<function>(1)` and `<function>(1.0)` would be treated differently. Defaults to `False`.

popitem()
Delete and return next cache item, `(key, value)`, based on cache replacement policy while ignoring expiration times (i.e. the selection of the item to pop is based soley on the cache key ordering).

Returns Two-element tuple of deleted cache `(key, value)`.

Return type tuple

set(*key*, *value*, *ttl=None*)
Set cache key/value and replace any previously set cache key. If the cache key previous existed, setting it will move it to the end of the cache stack which means it would be evicted last.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to set.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

set_many(*items*, *ttl=None*)
Set multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

size()
Return number of cache entries.

values()
Return `dict_values` view of all cache values.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_values

6.8 RR Cache

The rr module provides the *RRCache* (Random Replacement) class.

class cacheout.rr.**RRCache**(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Bases: *cacheout.cache.Cache*

The Random Replacment (RR) cache is like *Cache* but uses a random eviction policy where keys are evicted in a random order.

add(*key*, *value*, *ttl=None*)
Add cache key/value if it doesn't already exist. Essentially, this method ignores keys that exist which leaves the original TTL in tact.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to add.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

add_many(*items*, *ttl=None*)
Add multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

clear()
Clear all cache entries.

configure(*maxsize=None*, *ttl=None*, *timer=None*, *default=None*)
Configure cache settings.

This method is meant to support runtime level configurations for global level cache objects.

copy()
Return a copy of the cache.

Returns OrderedDict

delete(*key*)
Delete cache key and return number of entries deleted (`1` or `0`).

Returns `1` if key was deleted, `0` if key didn't exist.

Return type int

delete_expired()
Delete expired cache keys and return number of entries deleted.

Returns Number of entries deleted.

Return type int

delete_many(*iteratee*)
Delete multiple cache keys at once filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.

Returns Number of cache keys deleted.

Return type int

evict()
Perform cache eviction per the cache replacement policy:

- First, remove **all** expired entries.
- Then, remove non-TTL entries using the cache replacement policy.

When removing non-TTL entries, this method will only remove the minimum number of entries to reduce the number of entries below `maxsize`. If `maxsize` is `0`, then only expired entries will be removed.

Returns Number of cache entries evicted.

Return type int

expire_times()
Return cache expirations for TTL keys.

Returns dict

expired(*key*, *expires_on=None*)
Return whether cache key is expired or not.

Parameters

- **key** (*mixed*) – Cache key.
- **expires_on** (*float, optional*) – Timestamp of when the key is considered expired. Defaults to `None` which uses the current value returned from `timer()`.

Returns bool

full()
Return whether the cache is full or not.

Returns bool

get(*key*, *default=None*)
Return the cache value for *key* or *default* or `missing(key)` if it doesn't exist or has expired.

Parameters

- **key** (*mixed*) – Cache key.
- **default** (*mixed, optional*) – Value to return if *key* doesn't exist. If any value other than `None`, then it will take precendence over `missing` and be used as the return value. If *default* is callable, it will function like `missing` and its return value will be set for the cache *key*. Defaults to `None`.

Returns The cached value.

Return type mixed

get_many(*iteratee*, *default=None*)
Return many cache values as a `dict` of key/value pairs filtered by an *iteratee* that can be one of:

- `list` - List of cache keys.
- `str` - Search string that supports Unix shell-style wildcards.
- `re.compile()` - Compiled regular expression.
- `function` - Function that returns whether a key matches. Invoked with `iteratee(key)`.

Parameters

- **iteratee** (*list|str|Pattern|callable*) – Iteratee to filter by.
- **default** (*mixed, optional*) – Value to return if key doesn't exist. Defaults to `None`.

Returns dict

has(*key*)
Return whether cache key exists and hasn't expired.

Returns bool

items()
Return a `dict_items` view of cache items.

> **Warning:** Returned data is copied from the cache object, but any modifications to mutable values will modify this cache object's data.

Returns dict_items

keys()
Return `dict_keys` view of all cache keys.

> **Note:** Cache is copied from the underlying cache storage before returning.

Returns dict_keys

memoize(*, *ttl=None*, *typed=False*)
Decorator that wraps a function with a memoizing callable and works on both synchronous and asynchronous functions.

Each return value from the function will be cached using the function arguments as the cache key. The cache object can be accessed at `<function>.cache`. The uncached version (i.e. the original function) can be accessed at `<function>.uncached`. Each return value from the function will be cached using the function arguments as the cache key.

Keyword Arguments

- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.
- **typed** (*bool, optional*) – Whether to cache arguments of a different type separately. For example, `<function>(1)` and `<function>(1.0)` would be treated differently. Defaults to `False`.

popitem()
Delete and return next cache item, `(key, value)`, based on cache replacement policy while ignoring expiration times (i.e. the selection of the item to pop is based soley on the cache key ordering).

Returns Two-element tuple of deleted cache `(key, value)`.

Return type tuple

set(*key*, *value*, *ttl=None*)
Set cache key/value and replace any previously set cache key. If the cache key previous existed, setting it will move it to the end of the cache stack which means it would be evicted last.

Note: Cache key must be hashable.

Parameters

- **key** (*mixed*) – Cache key to set.
- **value** (*mixed*) – Cache value.
- **ttl** (*int, optional*) – TTL value. Defaults to `None` which uses `ttl`.

set_many(*items*, *ttl=None*)
Set multiple cache keys at once.

Parameters **items** (*dict*) – Mapping of cache key/values to set.

size()
Return number of cache entries.

values()
Return `dict_values` view of all cache values.

Note: Cache is copied from the underlying cache storage before returning.

Returns dict_values

6.9 Memoization

The memoization modules provides standalone memoiziation decorators that create an independent cache object for each decorated function.

cacheout.memoization.**fifo_memoize**(*maxsize=128*, *ttl=0*, *typed=False*)
Like *memoize()* except it uses *FIFOCache*.

cacheout.memoization.**lfu_memoize**(*maxsize=128*, *ttl=0*, *typed=False*)
Like *memoize()* except it uses *LFUCache*.

cacheout.memoization.**lifo_memoize**(*maxsize=128*, *ttl=0*, *typed=False*)
Like *memoize()* except it uses *LIFOCache*.

cacheout.memoization.**lru_memoize**(*maxsize=128*, *ttl=0*, *typed=False*)
Like *memoize()* except it uses *LRUCache*.

cacheout.memoization.**memoize**(*maxsize=128*, *ttl=0*, *typed=False*)
Decorator that wraps a function with a memoizing callable and works on both synchronous and asynchronous functions.

A cache object will be created for each memoized function using *Cache* and the arguments provided to this decorator followed by an immediate call to *Cache.memoize()* to wrap the function. The cache object can be accessed at `<function>.cache`. The uncached version (i.e. the original function) can be accessed at `<function>.uncached`. Each return value from the function will be cached using the function arguments as the cache key.

Parameters

- **maxsize** (*int, optional*) – Maximum size of cache dictionary. Defaults to `128`.
- **ttl** (*int, optional*) – Default TTL for all cache entries. Defaults to `0` which means that entries do not expire.
- **typed** (*bool, optional*) – Whether to cache arguments of a different type separately. For example, `<function>(1)` and `<function>(1.0)` would be treated differently. Defaults to `False`.

cacheout.memoization.**mru_memoize**(*maxsize=128*, *ttl=0*, *typed=False*)
Like *memoize()* except it uses *MRUCache*.

cacheout.memoization.**rr_memoize**(*maxsize=128*, *ttl=0*, *typed=False*)
Like *memoize()* except it uses *RRCache*.

6.10 Cache Manager

The manager module provides the *CacheManager* class.

class cacheout.manager.**CacheManager**(*settings=None*, *cache_class=<class 'cacheout.cache.Cache'>*)
The cache manager provides an interface for accessing multiple caches indexed by name.

Each named cache is a separate cache instance with its own configuration. Named caches can be configured during initialization or later using the *setup()* (bulk configuration) or *configure()* (individual configuration) methods.

Example

```
>>> # Configure bulk caches during initialization
>>> caches = CacheManager({"A": {"maxsize": 100}, "B": {"ttl": 90}})
>>> assert "A" in caches
>>> assert "B" in caches
```

```
>>> # Replace bulk caches after initializtion
>>> class MyCache(Cache): pass
>>> caches.setup({"C": {"cache_class": MyCache}, "D": {}})
>>> assert "A" not in caches
>>> assert "B" not in caches
>>> assert "C" in caches
>>> assert isinstance(caches["C"], MyCache)
>>> assert "D" in caches
>>> assert isinstance(caches["D"], Cache)
```

```
>>> # Configure individual cache after initializtaion
>>> caches.configure("E", **{"cache_class": MyCache})
>>> assert isinstance(caches["E"], MyCache)
```

```
>>> # Replace a cache entity
>>> caches.register("E", Cache())
>>> assert isinstance(caches["E"], Cache)
```

```
>>> # Access caches
>>> caches["C"].set("key1", "value1")
>>> caches["D"].set("key2", "value2")
```

Parameters

- **settings** (*dict, optional*) – A `dict` indexed by each cache name that should be created and configured. Defaults to `None` which doesn't configure anything.
- **cache_class** (*callable, optional*) – A factory function used when creating a cache.

cache_names()
Return list of names of cache entities.

Returns list

caches()
Return list of cache instances.

Returns list

clear_all()
Clear all caches.

configure(*name*, ***options*)
Configure cache identified by *name*.

Note: If no cache has been configured for *name*, then it will be created.

Keyword Arguments ****options** – Cache options.

register(*name*, *cache*)
Register a named cache instance.

setup(*settings=None*)
Set up named cache instances using configuration defined in *settings*.

The *settings* should contain key/values corresponding to the cache name and its cache options, respectively. Named caches are then accessible using index access on the cache handler object.

Warning: Calling this method will destroy **all** previously configured named caches and replace them with what is defined in *settings*.

Parameters **settings** (*dict, optional*) – A `dict` indexed by each cache name that contains the options for each named cache.

6.11 Developer Guide

This guide provides an overview of the tooling this project uses and how to execute developer workflows using the developer CLI.

6.11.1 Python Environments

This Python project is tested against different Python versions. For local development, it is a good idea to have those versions installed so that tests can be run against each.

There are libraries that can help with this. Which tools to use is largely a matter of preference, but below are a few recommendations.

For managing multiple Python versions:

- pyenv
- OS package manager (e.g. apt, yum, homebrew, etc)
- Build from source

For managing Python virtualenvs:

- pyenv-virtualenv
- pew
- python-venv

6.11.2 Tooling

The following tools are used by this project:

Tool	Description	Configuration
black	Code formatter	`pyproject.toml`
isort	Import statement formatter	`setup.cfg`
docformatter	Docstring formatter	`setup.cfg`
flake8	Code linter	`setup.cfg`
pylint	Code linter	`pylintrc`
pytest	Test framework	`setup.cfg`
tox	Test environment manager	`tox.ini`
invoke	CLI task execution library	`tasks.py`

6.11.3 Workflows

The following workflows use developer CLI commands via invoke and are defined in `tasks.py`.

Autoformat Code

To run all autoformatters:

```
inv fmt
```

This is the same as running each autoformatter individually:

```
inv black
inv isort
inv docformatter
```

Lint

To run all linters:

```
inv lint
```

This is the same as running each linter individually:

```
inv flake8
inv pylint
```

Test

To run all unit tests:

```
inv unit
```

To run unit tests and builds:

```
inv test
```

Test on All Supported Python Versions

To run tests on all supported Python versions:

```
tox
```

This requires that the supported versions are available on the PATH.

Build Package

To build the package:

```
inv build
```

This will output the source and binary distributions under `dist/`.

Build Docs

To build documentation:

```
inv docs
```

This will output the documentation under `docs/_build/`.

Serve Docs

To serve docs over HTTP:

```
inv docs -s|--server [-b|--bind 127.0.0.1] [-p|--port 8000]

inv docs -s
inv docs -s -p 8080
inv docs -s -b 0.0.0.0 -p 8080
```

Delete Build Files

To remove all build and temporary files:

```
inv clean
```

This will remove Python bytecode files, egg files, build output folders, caches, and tox folders.

Release Package

To release a new version of the package to https://pypi.org:

```
inv release
```

6.11.4 CI/CD

This project uses Github Actions for CI/CD:

- https://github.com/dgilland/fnc/actions

CHAPTER 7

Project Info

7.1 License

7.2 Versioning

This project follows Semantic Versioning with the following caveats:

- Only the public API (i.e. the objects imported into the cacheout module) will maintain backwards compatibility between MINOR version bumps.
- Objects within any other parts of the library are not guaranteed to not break between MINOR version bumps.

With that in mind, it is recommended to only use or import objects from the main module, cacheout.

7.3 Changelog

7.3.1 v0.11.2 (2019-09-30)

- Fix bug in `LFUCache` that would result cache growing beyond `maxsize` limit.

7.3.2 v0.11.1 (2019-01-09)

- Fix issue with asyncio support in memoization decorators that caused a `RuntimeError: await wasn't used with future` when certain types of async functions were used inside the memoized function.

7.3.3 v0.11.0 (2018-10-19)

- Add asyncio support to memoization decorators so they can decorate coroutines.

7.3.4 v0.10.3 (2018-08-01)

- Expose `typed` argument of underlying `*Cache.memoize()` in `memoize()` and `*_memoize()` decorators.

7.3.5 v0.10.2 (2018-07-31)

- Fix bug in `LRUCache.get()` where supplying a `default` value would result in a `KeyError`.

7.3.6 v0.10.1 (2018-07-15)

- Support Python 3.7.

7.3.7 v0.10.0 (2018-04-03)

- Modify behavior of `default` argument to `Cache.get()` so that if `default` is a callable and the cache key is missing, then it will be called and its return value will be used as the value for cache key and subsequently be set as the value for the key in the cache. (**breaking change**)
- Add `default` argument to `Cache()` that can be used to override the value for `default` in `Cache.get()`.

7.3.8 v0.9.0 (2018-03-31)

- Merge functionality of `Cache.get_many_by()` into `Cache.get_many()` and remove `Cache.get_many_by()`. (**breaking change**).
- Merge functionality of `Cache.delete_many_by()` into `Cache.delete_many()` and remove `Cache.delete_many_by()`. (**breaking change**).

7.3.9 v0.8.0 (2018-03-30)

- Add `Cache.get_many_by()`.
- Add `Cache.delete_many_by()`.
- Make `Cache.keys()` and `Cache.values()` return dictionary view objects instead of yielding items. (**breaking change**)

7.3.10 v0.7.0 (2018-02-22)

- Changed default cache `maxsize` from `300` to `256`. (**breaking change**)
- Add `Cache.memoize()` decorator.
- Add standalone memoization decorators:
  - `memoize`
  - `fifo_memoize`
  - `lfu_memoize`
  - `lifo_memoize`
  - `lru_memoize`
  - `mru_memoize`
  - `rr_memoize`

7.3.11 v0.6.0 (2018-02-05)

- Add `LIFOCache`
- Add `FIFOCache` as an alias of `Cache`.

7.3.12 v0.5.0 (2018-02-04)

- Add `LFUCache`
- Delete expired items before popping an item in `Cache.popitem()`.

7.3.13 v0.4.0 (2018-02-02)

- Add `MRUCache`
- Add `RRCache`
- Add `Cache.popitem()`.
- Rename `Cache.expirations()` to `Cache.expire_times()`. (**breaking change**)
- Rename `Cache.count()` to `Cache.size()`. (**breaking change**)
- Remove `minimum` arguement from `Cache.evict()`. (**breaking change**)

7.3.14 v0.3.0 (2018-01-31)

- Add `LRUCache`.
- Add `CacheManager.__repr__()`.
- Make threading lock usage in `Cache` more fine-grained and eliminate redundant locking.
- Fix missing thread-safety in `Cache.__len__()` and `Cache.__contains__()`.

7.3.15 v0.2.0 (2018-01-30)

- Rename `Cache.setup()` to `Cache.configure()`. (**breaking change**)
- Add `CacheManager` class.

7.3.16 v0.1.0 (2018-01-28)

- Add `Cache` class.

7.4 Authors

7.4.1 Lead

- Derrick Gilland, dgilland@gmail.com, dgilland@github

7.4.2 Contributors

None

7.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/dgilland/cacheout.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" or "help wanted" is open to whoever wants to implement it.

Write Documentation

cacheout could always use more documentation, whether as part of the official cacheout docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/dgilland/cacheout.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.5.2 Get Started!

Ready to contribute? Here's how to set up `cacheout` for local development.

1. Fork the `cacheout` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_username_here/cacheout.git
```

3. Install Python dependencies into a virtualenv:

```
$ cd cacheout
$ pip install -r requirements.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. Autoformat code using black:

```
$ tox -e black
```

6. When you're done making changes, check that your changes pass linting and all unit tests by testing with tox across all supported Python versions:

```
$ tox
```

7. Add yourself to `AUTHORS.rst`.
8. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

9. Submit a pull request through GitHub.

7.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the README.rst.
3. The pull request should work for all versions Python that this project supports. Check https://travis-ci.org/dgilland/cacheout/pull_requests and make sure that the all environments pass.

CHAPTER 8

Indices and Tables

- genindex
- modindex
- search

Python Module Index

C

Index

E

F

G

H

I

K

L

M

P

R

S

T

V