cachetools

Release 1.1.6

October 03, 2016

This module provides various memoizing collections and decorators, including variants of the Python 3 Standard Library @lru_cache function decorator.

For the purpose of this module, a *cache* is a mutable mapping of a fixed maximum size. When the cache is full, i.e. by adding another item the cache would exceed its maximum size, the cache must choose which item(s) to discard based on a suitable cache algorithm. In general, a cache's size is the total size of its items, and an item's size is a property or function of its value, e.g. the result of `sys.getsizeof(value)`. For the trivial but common case that each item counts as `1`, a cache's size is equal to the number of its items, or `len(cache)`.

Multiple cache classes based on different caching algorithms are implemented, and decorators for easily memoizing function and method calls are provided, too.

Note: Several features are now marked as deprecated and will be removed in the next major release, *`cachetools`* version 2.0. If you happen to rely on any of these features, it is highly recommended to specify your module dependencies accordingly, for example `cachetools ~= 1.1` when using `setuptools`.

Changed in version 1.1: Moved `functools.lru_cache()` compatible decorators to the *`cachetools.func`* module. For backwards compatibility, they continue to be visible in this module as well.

CHAPTER 1

Cache implementations

This module provides several classes implementing caches using different cache algorithms. All these classes derive from class *Cache*, which in turn derives from `collections.MutableMapping`, and provide `maxsize` and `currsize` properties to retrieve the maximum and current size of the cache. When a cache is full, `setitem()` calls `popitem()` repeatedly until there is enough room for the item to be added.

All cache classes accept an optional *missing* keyword argument in their constructor, which can be used to provide a default *factory function*. If the key *key* is not present, the `cache[key]` operation calls `Cache.__missing__()`, which in turn calls *missing* with *key* as its sole argument. The cache will then store the object returned from `missing(key)` as the new cache value for *key*, possibly discarding other items if the cache is full. This may be used to provide memoization for existing single-argument functions:

```
from cachetools import LRUCache
import urllib.request

def get_pep(num):
    """Retrieve text of a Python Enhancement Proposal"""
    url = 'http://www.python.org/dev/peps/pep-%04d/' % num
    with urllib.request.urlopen(url) as s:
        return s.read()

cache = LRUCache(maxsize=4, missing=get_pep)

for n in 8, 9, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
    try:
        print(n, len(cache[n]))
    except urllib.error.HTTPError:
        print(n, 'Not Found')
print(sorted(cache.keys()))
```

Cache also features a `getsizeof()` method, which returns the size of a given *value*. The default implementation of `getsizeof()` returns `1` irrespective of its argument, making the cache's size equal to the number of its items, or `len(cache)`. For convenience, all cache classes accept an optional named constructor parameter *getsizeof*, which may specify a function of one argument used to retrieve the size of an item's value.

class `cachetools.`**`Cache`**(*maxsize*, *missing=None*, *getsizeof=None*)

Mutable mapping to serve as a simple cache or cache base class.

This class discards arbitrary items using `popitem()` to make space when necessary. Derived classes may override `popitem()` to implement specific caching strategies. If a subclass has to keep track of item access, insertion or deletion, it may additionally need to override `__getitem__()`, `__setitem__()` and `__delitem__()`. If a subclass wants to store meta data with its values, i.e. the *value* argument passed to `Cache.__setitem__()` is different from what the derived class's `__setitem__()` received, it will probably need to override *getsizeof()*, too.

currsize
The current size of the cache.

getsizeof(*value*)
Return the size of a cache element's value.

maxsize
The maximum size of the cache.

class `cachetools.`**LFUCache**(*maxsize*, *missing=None*, *getsizeof=None*)
Least Frequently Used (LFU) cache implementation.

This class counts how often an item is retrieved, and discards the items used least often to make space when necessary.

popitem()
Remove and return the *(key, value)* pair least frequently used.

class `cachetools.`**LRUCache**(*maxsize*, *missing=None*, *getsizeof=None*)
Least Recently Used (LRU) cache implementation.

This class discards the least recently used items first to make space when necessary.

popitem()
Remove and return the *(key, value)* pair least recently used.

class `cachetools.`**RRCache**(*maxsize*, *choice=random.choice*, *missing=None*, *getsizeof=None*)
Random Replacement (RR) cache implementation.

This class randomly selects candidate items and discards them to make space when necessary.

By default, items are selected from the list of cache keys using `random.choice()`. The optional argument *choice* may specify an alternative function that returns an arbitrary element from a non-empty sequence.

choice
The *choice* function used by the cache.

popitem()
Remove and return a random *(key, value)* pair.

class `cachetools.`**TTLCache**(*maxsize*, *ttl*, *timer=time.time*, *missing=None*, *getsizeof=None*)
LRU Cache implementation with per-item time-to-live (TTL) value.

This class associates a time-to-live value with each item. Items that expire because they have exceeded their time-to-live will be removed automatically. If no expired items are there to remove, the least recently used items will be discarded first to make space when necessary. Trying to access an expired item will raise a `KeyError`.

By default, the time-to-live is specified in seconds, and the `time.time()` function is used to retrieve the current time. A custom *timer* function can be supplied if needed.

expire(*self*, *time=None*)
Remove expired items from the cache.

Since expired items will be "physically" removed from a cache only at the next mutating operation, e.g. `__setitem__()` or `__delitem__()`, to avoid changing the underlying dictionary while iterating over it, expired items may still claim memory although they are no longer accessible. Calling this method removes all items whose time-to-live would have expired by *time*, so garbage collection is free to reuse their memory. If *time* is `None`, this removes all items that have expired by the current value returned by `timer`.

popitem()
Remove and return the *(key, value)* pair least recently used that has not already expired.

`timer`
The timer function used by the cache.

`ttl`
The time-to-live value of the cache's items.

CHAPTER 2

Decorators

The *cachetools* module provides decorators for memoizing function and method calls. This can save time when a function is often called with the same arguments:

```
from cachetools import cached

@cached(cache={})
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)

for i in range(100):
    print('fib(%d) = %d' % (i, fib(i)))
```

@cachetools.**cached**(*cache*, *key=hashkey*, *lock=None*)

Decorator to wrap a function with a memoizing callable that saves results in a cache.

The *cache* argument specifies a cache object to store previous function arguments and return values. Note that *cache* need not be an instance of the cache implementations provided by the *cachetools* module. *cached()* will work with any mutable mapping type, including plain `dict` and `weakref.WeakValueDictionary`.

key specifies a function that will be called with the same positional and keyword arguments as the wrapped function itself, and which has to return a suitable cache key. Since caches are mappings, the object returned by *key* must be hashable. The default is to call *hashkey()*.

If *lock* is not `None`, it must specify an object implementing the context manager protocol. Any access to the cache will then be nested in a `with lock:` statement. This can be used for synchronizing thread access to the cache by providing a `threading.RLock` instance, for example.

Note: The *lock* context manager is used only to guard access to the cache object. The underlying wrapped function will be called outside the *with* statement, and must be thread-safe by itself.

The original underlying function is accessible through the `__wrapped__` attribute of the memoizing wrapper function. This can be used for introspection or for bypassing the cache.

To perform operations on the cache object, for example to clear the cache during runtime, the cache should be assigned to a variable. When a *lock* object is used, any access to the cache from outside the function wrapper should also be performed within an appropriate *with* statement:

```
from threading import RLock
from cachetools import cached, LRUCache

cache = LRUCache(maxsize=100)
lock = RLock()
```

```
@cached(cache, lock=lock)
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)

# make sure access to cache is synchronized
with lock:
    cache.clear()
```

It is also possible to use a single shared cache object with multiple functions. However, care must be taken that different cache keys are generated for each function, even for identical function arguments:

```
from functools import partial
from cachetools import cached, hashkey, LRUCache

cache = LRUCache(maxsize=100)

@cached(cache, key=partial(hashkey, 'fib'))
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)

@cached(cache, key=partial(hashkey, 'fac'))
def fac(n):
    return 1 if n == 0 else n * fac(n - 1)

print(fib(42))
print(fac(42))
print(cache)
```

New in version 1.1.

`@cachetools.`**`cachedmethod`**`(`*cache*, *key=hashkey*, *lock=None*, *typed=False*`)`

Decorator to wrap a class or instance method with a memoizing callable that saves results in a (possibly shared) cache.

The main difference between this and the `cached()` function decorator is that *cache* and *lock* are not passed objects, but functions. Both will be called with `self` (or `cls` for class methods) as their sole argument to retrieve the cache or lock object for the method's respective instance or class.

Note: As with `cached()`, the context manager obtained by calling `lock(self)` will only guard access to the cache itself. It is the user's responsibility to handle concurrent calls to the underlying wrapped method in a multithreaded environment.

If *key* or the optional *typed* keyword argument are set to `True`, the `typedkey()` function is used for generating hash keys. This has been deprecated in favor of specifying `key=typedkey` explicitly.

One advantage of `cachedmethod()` over the `cached()` function decorator is that cache properties such as *maxsize* can be set at runtime:

```
import operator
import urllib.request

from cachetools import LRUCache, cachedmethod

class CachedPEPs(object):

    def __init__(self, cachesize):
        self.cache = LRUCache(maxsize=cachesize)
    @cachedmethod(operator.attrgetter('cache'))
    def get(self, num):
        """Retrieve text of a Python Enhancement Proposal"""
        url = 'http://www.python.org/dev/peps/pep-%04d/' % num
        with urllib.request.urlopen(url) as s:
            return s.read()

peps = CachedPEPs(cachesize=10)
print("PEP #1: %s" % peps.get(1))
```

For backwards compatibility, the default key function used by `cachedmethod()` will generate distinct keys for different methods to ease using a shared cache with multiple methods. This has been deprecated, and relying on this feature is strongly discouraged. When using a shared cache, distinct key functions should be used, as with the `cached()` decorator.

New in version 1.1: The optional *key* and *lock* parameters.

Changed in version 1.1: The `__wrapped__` attribute is now set when running Python 2.7, too.

Deprecated since version 1.1: The *typed* argument. Use `key=typedkey` instead.

Deprecated since version 1.1: When using a shared cached for multiple methods, distinct key function should be used.

Deprecated since version 1.1: The wrapper function's `cache` attribute. Use the original function passed as the decorator's *cache* argument to access the cache object.

CHAPTER 3

Key functions

The following functions can be used as key functions with the *cached()* and *cachedmethod()* decorators:

cachetools.**hashkey**(**args*, ***kwargs*)
: Return a cache key for the specified hashable arguments.

  This function returns a tuple instance suitable as a cache key, provided the positional and keywords arguments are hashable.

  New in version 1.1.

cachetools.**typedkey**(**args*, ***kwargs*)
: Return a typed cache key for the specified hashable arguments.

  This function is similar to *hashkey()*, but arguments of different types will yield distinct cache keys. For example, typedkey(3) and typedkey(3.0) will return different results.

  New in version 1.1.

These functions can also be helpful when implementing custom key functions for handling some non-hashable arguments. For example, calling the following function with a dictionary as its *env* argument will raise a TypeError, since dict is not hashable:

```
@cached(LRUCache(maxsize=128))
def foo(x, y, z, env={}):
    pass
```

However, if *env* always holds only hashable values itself, a custom key function can be written that handles the *env* keyword argument specially:

```
def envkey(*args, env={}, **kwargs):
    key = hashkey(*args, **kwargs)
    key += tuple(env.items())
    return key
```

The envkey() function can then be used in decorator declarations like this:

```
@cached(LRUCache(maxsize=128), key=envkey)
```

CHAPTER 4

`cachetools.func` — `functools.lru_cache()` compatible decorators

To ease migration from (or to) Python 3's `functools.lru_cache()`, this module provides several memoizing function decorators with a similar API. All these decorators wrap a function with a memoizing callable that saves up to the *maxsize* most recent calls, using different caching strategies. Note that unlike `functools.lru_cache()`, setting *maxsize* to `None` is not supported.

The wrapped function is instrumented with `cache_info()` and `cache_clear()` functions to provide information about cache performance and clear the cache. See the `functools.lru_cache()` documentation for details.

In addition to *maxsize*, all decorators accept the following optional keyword arguments:

- *typed*, if is set to `True`, will cause function arguments of different types to be cached separately. For example, `f(3)` and `f(3.0)` will be treated as distinct calls with distinct results.
- *getsizeof* specifies a function of one argument that will be applied to each cache value to determine its size. The default value is `None`, which will assign each item an equal size of `1`. This has been deprecated in favor of the new `cachetools.cached()` decorator, which allows passing fully customized cache objects.
- *lock* specifies a function of zero arguments that returns a context manager to lock the cache when necessary. If not specified, `threading.RLock` will be used to synchronize access from multiple threads. The use of *lock* is discouraged, and the *lock* argument has been deprecated.

New in version 1.1: Formerly, the decorators provided by `cachetools.func` were part of the `cachetools` module.

`@cachetools.func.`**`lfu_cache`**`(`*maxsize=128*, *typed=False*, *getsizeof=None*, *lock=threading.RLock*`)`

Decorator that wraps a function with a memoizing callable that saves up to *maxsize* results based on a Least Frequently Used (LFU) algorithm.

Deprecated since version 1.1: The *getsizeof* and *lock* arguments.

`@cachetools.func.`**`lru_cache`**`(`*maxsize=128*, *typed=False*, *getsizeof=None*, *lock=threading.RLock*`)`

Decorator that wraps a function with a memoizing callable that saves up to *maxsize* results based on a Least Recently Used (LRU) algorithm.

Deprecated since version 1.1: The *getsizeof* and *lock* arguments.

`@cachetools.func.`**`rr_cache`**`(`*maxsize=128*, *choice=random.choice*, *typed=False*, *getsizeof=None*, *lock=threading.RLock*`)`

Decorator that wraps a function with a memoizing callable that saves up to *maxsize* results based on a Random Replacement (RR) algorithm.

Deprecated since version 1.1: The *getsizeof* and *lock* arguments.

`@cachetools.func.`**`ttl_cache`**`(`*maxsize=128*, *ttl=600*, *timer=time.time*, *typed=False*, *getsizeof=None*, *lock=threading.RLock*`)`

Decorator to wrap a function with a memoizing callable that saves up to *maxsize* results based on a Least Recently Used (LRU) algorithm with a per-item time-to-live (TTL) value.

Deprecated since version 1.1: The *getsizeof* and *lock* arguments.

Python Module Index

Index