`cachetools` — Extensible memoizing collections and decorators

This module provides various memoizing collections and decorators, including variants of the Python Standard Library's @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.

This module provides multiple cache classes based on different cache algorithms, as well as decorators for easily memoizing function and method calls.

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, `Cache.__setitem__()` calls `self.popitem()` repeatedly until there is enough room for the item to be added.

In general, a cache's size is the total size of its item's values. Therefore, `Cache` provides 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.

Note that the values of a `Cache` are mutable by default, as are e.g. the values of a `dict`. It is the user's responsibility to take care that cached values are not accidentally modified. This is especially important when using a custom *getsizeof* function, since the size of an item's value will only be computed when the item is inserted into the cache.

> **Note:** Please be aware that all these classes are *not* thread-safe. Access to a shared cache from multiple threads must be properly synchronized, e.g. by using one of the memoizing decorators with a suitable *lock* object.

class cachetools.**Cache**(*maxsize*, *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__()`.

property **currsize**

The current size of the cache.

static **getsizeof**(*value*)

Return the size of a cache element's value.

property **maxsize**

The maximum size of the cache.

class cachetools.**FIFOCache**(*maxsize*, *getsizeof=None*)

First In First Out (FIFO) cache implementation.

This class evicts items in the order they were added to make space when necessary.

popitem()

Remove and return the *(key, value)* pair first inserted.

class cachetools.**LFUCache**(*maxsize*, *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*, *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.**MRUCache**(*maxsize*, *getsizeof=None*)

Most Recently Used (MRU) cache implementation.

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

Deprecated since version 5.4.

MRUCache has been deprecated due to lack of use, to reduce maintenance. Please choose another cache implementation that suits your needs.

popitem()

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

class cachetools.**RRCache**(*maxsize*, *choice=random.choice*, *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.

property **choice**

The *choice* function used by the cache.

popitem()

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

class cachetools.**TTLCache**(*maxsize*, *ttl*, *timer=time.monotonic*, *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 no longer accessible, and will be removed eventually. If no expired items are there to remove, the least recently used items will be discarded first to make space when necessary.

By default, the time-to-live is specified in seconds and **`time.monotonic()`** is used to retrieve the current time.

```
cache = TTLCache(maxsize=10, ttl=60)
```

A custom *timer* function can also be supplied, which does not have to return seconds, or even a numeric value. The expression *timer() + ttl* at the time of insertion defines the expiration time of a cache item and must be comparable against later results of *timer()*, but *ttl* does not necessarily have to be a number, either.

```
from datetime import datetime, timedelta

cache = TTLCache(maxsize=10, ttl=timedelta(hours=12), timer=datetime.now)
```

expire(*self*, *time=None*)

Expired items will be removed from a cache only at the next mutating operation, e.g. **`__setitem__()`** or **`__delitem__()`**, and therefore may still claim memory. 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`.

Returns: An iterable of expired *(key, value)* pairs.

popitem()

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

property **timer**

The timer function used by the cache.

property **ttl**

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

class cachetools.**TLRUCache**(*maxsize*, *ttu*, *timer=time.monotonic*, *getsizeof=None*)

Time aware Least Recently Used (TLRU) cache implementation.

Similar to `TTLCache`, this class also associates an expiration time with each item. However, for `TLRUCache` items, expiration time is calculated by a user-provided time-to-use (*ttu*) function, which is passed three arguments at the time of insertion: the new item's key and value, as well as the current value of *timer()*.

```
def my_ttu(_key, value, now):
    # assume value.ttu contains the item's time-to-use in seconds
    # note that the _key argument is ignored in this example
    return now + value.ttu
```

```
cache = TLRUCache(maxsize=10, ttu=my_ttu)
```

The expression *ttu(key, value, timer())* defines the expiration time of a cache item, and must be comparable against later results of *timer()*. As with `TTLCache`, a custom *timer* function can be supplied, which does not have to return a numeric value.

```
from datetime import datetime, timedelta

def datetime_ttu(_key, value, now):
    # assume now to be of type datetime.datetime, and
    # value.hours to contain the item's time-to-use in hours
    return now + timedelta(hours=value.hours)

cache = TLRUCache(maxsize=10, ttu=datetime_ttu, timer=datetime.now)
```

Items that expire because they have exceeded their time-to-use will be no longer accessible, and will be removed eventually. If no expired items are there to remove, the least recently used items will be discarded first to make space when necessary.

expire(*self, time=None*)

Expired items will be removed from a cache only at the next mutating operation, e.g. **`__setitem__()`** or **`__delitem__()`**, and therefore may still claim memory. Calling this method removes all items whose time-to-use 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`.

Returns: An iterable of expired *(key, value)* pairs.

popitem()

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

property **timer**

The timer function used by the cache.

property **ttu**

The local time-to-use function used by the cache.

Extending cache classes

Sometimes it may be desirable to notice when and what cache items are evicted, i.e. removed from a cache to make room for new items. Since all cache implementations call **`popitem()`** to evict items from the cache, this can be achieved by overriding this method in a subclass:

```
>>> class MyCache(LRUCache):
...     def popitem(self):
...         key, value = super().popitem()
...         print('Key "%s" evicted with value "%s"' % (key, value))
...         return key, value

>>> c = MyCache(maxsize=2)
>>> c['a'] = 1
>>> c['b'] = 2
>>> c['c'] = 3
Key "a" evicted with value "1"
```


With `TTLCache` and `TLRUCache`, items may also be removed after they expire. In this case, **`popitem()`** will *not* be called, but **`expire()`** will be called from the next mutating operation and will return an iterable of the

expired *(key, value)* pairs. By overrding **`expire()`**, a subclass will be able to track expired items:

```
>>> class ExpCache(TTLCache):
...     def expire(self, time=None):
...         items = super().expire(time)
...         print(f"Expired items: {items}")
...         return items

>>> c = ExpCache(maxsize=10, ttl=1.0)
>>> c['a'] = 1
Expired items: []
>>> c['b'] = 2
Expired items: []
>>> time.sleep(1.5)
>>> c['c'] = 3
Expired items: [('a', 1), ('b', 2)]
```

Similar to the standard library's **`collections.defaultdict`**, subclasses of `Cache` may implement a **`__missing__()`** method which is called by **`Cache.__getitem__()`** if the requested key is not found:

```
>>> class PepStore(LRUCache):
...     def __missing__(self, key):
...         """Retrieve text of a Python Enhancement Proposal"""
...         url = 'http://www.python.org/dev/peps/pep-%04d/' % key
...         with urllib.request.urlopen(url) as s:
...             pep = s.read()
...             self[key] = pep  # store text in cache
...             return pep

>>> peps = PepStore(maxsize=4)
>>> for n in 8, 9, 290, 308, 320, 8, 218, 320, 279, 289, 320:
...     pep = peps[n]
>>> print(sorted(peps.keys()))
[218, 279, 289, 320]
```

Note, though, that such a class does not really behave like a *cache* any more, and will lead to surprising results when used with any of the memoizing decorators described below. However, it may be useful in its own right.

Memoizing 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:

```
>>> @cached(cache={})
... def fib(n):
...     'Compute the nth number in the Fibonacci sequence'
...     return n if n < 2 else fib(n - 1) + fib(n - 2)

>>> fib(42)
267914296
```

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

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 plai

`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 `cachetools.keys.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.Lock` 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 decorator's *cache*, *key* and *lock* parameters are also available as `cache`, `cache_key` and `cache_lock` attributes of the memoizing wrapper function. These can be used for clearing the cache or invalidating individual cache items, for example.

```
from threading import Lock

# 640K should be enough for anyone...
@cached(cache=LRUCache(maxsize=640*1024, getsizeof=len), lock=Lock())
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()

# make sure access to cache is synchronized
with get_pep.cache_lock:
    get_pep.cache.clear()

# always use the key function for accessing cache items
with get_pep.cache_lock:
    get_pep.cache.pop(get_pep.cache_key(42), None)
```

For the common use case of clearing or invalidating the cache, the decorator also provides a `cache_clear()` function which takes care of locking automatically, if needed:

```
# no need for get_pep.cache_lock here
get_pep.cache_clear()
```

If *info* is set to `True`, the wrapped function is instrumented with a `cache_info()` function that returns a named tuple showing *hits*, *misses*, *maxsize* and *currsize*, to help measure the effectiveness of the cache.

> **Note:** Note that this will inflict a - probably minor - performance penalty, so it has to be explicitly enabled.

```
>>> @cached(cache=LRUCache(maxsize=32), info=True)
... def get_pep(num):
...     url = 'http://www.python.org/dev/peps/pep-%04d/' % num
...     with urllib.request.urlopen(url) as s:
...         return s.read()

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

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

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 cachetools.keys import hashkey
>>> from functools import partial

>>> # shared cache for integer sequences
>>> numcache = {}

>>> # compute Fibonacci numbers
>>> @cached(numcache, key=partial(hashkey, 'fib'))
... def fib(n):
...     return n if n < 2 else fib(n - 1) + fib(n - 2)

>>> # compute Lucas numbers
>>> @cached(numcache, key=partial(hashkey, 'luc'))
... def luc(n):
...     return 2 - n if n < 2 else luc(n - 1) + luc(n - 2)

>>> fib(42)
267914296
>>> luc(42)
599074578
>>> list(sorted(numcache.items()))
[..., (('fib', 42), 267914296), ..., (('luc', 42), 599074578)]
```

@cachetools.**cachedmethod**(*cache*, *key=cachetools.keys.methodkey*, *lock=None*)

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.

The *key* function will be called as *key(self, *args, **kwargs)* to retrieve a suitable cache key. Note that the default *key* function, `cachetools.keys.methodkey()`, ignores its first argument, i.e. **self**. This has mostly historical reasons, but also ensures that **self** does not have to be hashable. You may provide a different *key* function, e.g. `cachetools.keys.hashkey()`, if you need **self** to be part of the cache key.

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

```
class CachedPEPs:

    def __init__(self, cachesize):
        self.cache = LRUCache(maxsize=cachesize)

    @cachedmethod(lambda self: self.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))
```

When using a shared cache for multiple methods, be aware that different cache keys must be created for each method even when function arguments are the same, just as with the *@cached* decorator:

```
class CachedReferences:

    def __init__(self, cachesize):
        self.cache = LRUCache(maxsize=cachesize)

    @cachedmethod(lambda self: self.cache, key=partial(hashkey, 'pep'))
    def get_pep(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()

    @cachedmethod(lambda self: self.cache, key=partial(hashkey, 'rfc'))
    def get_rfc(self, num):
        """Retrieve text of an IETF Request for Comments"""
        url = 'https://tools.ietf.org/rfc/rfc%d.txt' % num
        with urllib.request.urlopen(url) as s:
            return s.read()

docs = CachedReferences(cachesize=100)
print("PEP #1: %s" % docs.get_pep(1))
print("RFC #1: %s" % docs.get_rfc(1))
```

`cachetools.keys` — Key functions for memoizing decorators

This module provides several functions that can be used as key functions with the **`cached()`** and **`cachedmethod()`** decorators:

cachetools.keys.**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.

cachetools.keys.**methodkey**(*self*, **args*, ***kwargs*)

Return a cache key for use with cached methods.

This function is similar to **`hashkey()`**, but ignores its first positional argument, i.e. *self* when used with the **`cachedmethod()`** decorator.

cachetools.keys.**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.

cachetools.keys.**typedmethodkey**(*self*, **args*, ***kwargs*)

Return a typed cache key for use with cached methods.

This function is similar to **`typedkey()`**, but ignores its first positional argument, i.e. *self* w

the **`cachedmethod()`** decorator.

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(sorted(env.items()))
    return key
```

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

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

foo(1, 2, 3, env=dict(a='a', b='b'))
```

`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. If *maxsize* is set to **`None`**, the caching strategy is effectively disabled and the cache can grow without bound.

If the optional argument *typed* is set to **`True`**, function arguments of different types will be cached separately. For example, `f(3)` and `f(3.0)` will be treated as distinct calls with distinct results.

If a *user_function* is specified instead, it must be a callable. This allows the decorator to be applied directly to a user function, leaving the *maxsize* at its default value of 128:

```
@cachetools.func.lru_cache
def count_vowels(sentence):
    sentence = sentence.casefold()
    return sum(sentence.count(vowel) for vowel in 'aeiou')
```

The wrapped function is instrumented with a **`cache_parameters()`** function that returns a new **`dict`** showing the values for *maxsize* and *typed*. This is for information purposes only. Mutating the values has no effect.

The wrapped function is also instrumented with **`cache_info()`** and **`cache_clear()`** functions to provide information about cache performance and clear the cache. Please see the **`functools.lru_cache()`** documentation for details. Also note that all the decorators in this module are thread-safe by default.

`@cachetools.func.`**`fifo_cache`**`(`*`user_function`*`)`
`@cachetools.func.`**`fifo_cache`**`(`*`maxsize=128, typed=False`*`)`

Decorator that wraps a function with a memoizing callable that saves up to *maxsize* results based on a First In First Out (FIFO) algorithm.

`@cachetools.func.`**`lfu_cache`**`(`*`user_function`*`)`

@cachetools.func.**lfu_cache**(*maxsize=128*, *typed=False*)

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

@cachetools.func.**lru_cache**(*user_function*)
@cachetools.func.**lru_cache**(*maxsize=128*, *typed=False*)

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

@cachetools.func.**mru_cache**(*user_function*)
@cachetools.func.**mru_cache**(*maxsize=128*, *typed=False*)

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

Deprecated since version 5.4.

The *mru_cache* decorator has been deprecated due to lack of use. Please choose a decorator based on some other algorithm.

@cachetools.func.**rr_cache**(*user_function*)
@cachetools.func.**rr_cache**(*maxsize=128*, *choice=random.choice*, *typed=False*)

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

@cachetools.func.**ttl_cache**(*user_function*)
@cachetools.func.**ttl_cache**(*maxsize=128*, *ttl=600*, *timer=time.monotonic*, *typed=False*)

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.

Index

C | E | F | G | H | L | M | P | R | T

C

E

F

G

H

L

M

P

R

T

Python Module Index

```
:tocdepth: 3

**********************************************************************
:mod:`cachetools` --- Extensible memoizing collections and decorators
**********************************************************************

.. module:: cachetools

This module provides various memoizing collections and decorators,
including variants of the Python Standard Library's `@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`_.

This module provides multiple cache classes based on different cache
algorithms, as well as decorators for easily memoizing function and
method calls.

.. testsetup:: *

   from cachetools import cached, cachedmethod, LRUCache, TLRUCache, TTLCache

   from unittest import mock
   urllib = mock.MagicMock()

   import time


Cache implementations
=====================

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

In general, a cache's size is the total size of its item's values.
Therefore, :class:`Cache` provides a :meth:`getsizeof` method, which
returns the size of a given `value`.  The default implementation of
:meth:`getsizeof` returns :const:`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.

Note that the values of a :class:`Cache` are mutable by default, as
are e.g. the values of a :class:`dict`.  It is the user's
responsibility to take care that cached values are not accidentally
modified.  This is especially important when using a custom
`getsizeof` function, since the size of an item's value will only be
computed when the item is inserted into the cache.

.. note::

   Please be aware that all these classes are *not* thread-safe.
   Access to a shared cache from multiple threads must be properly
   synchronized, e.g. by using one of the memoizing decorators with a
   suitable `lock` object.

.. autoclass:: Cache(maxsize, getsizeof=None)
   :members: currsize, getsizeof, maxsize

   This class discards arbitrary items using :meth:`popitem` to make
   space when necessary.  Derived classes may override :meth:`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 :meth:`__getitem__`,
```

```
   :meth:`__setitem__` and :meth:`__delitem__`.

.. autoclass:: FIFOCache(maxsize, getsizeof=None)
   :members: popitem

   This class evicts items in the order they were added to make space
   when necessary.

.. autoclass:: LFUCache(maxsize, getsizeof=None)
   :members: popitem

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

.. autoclass:: LRUCache(maxsize, getsizeof=None)
   :members: popitem

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

.. autoclass:: MRUCache(maxsize, getsizeof=None)
   :members: popitem

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

   .. deprecated:: 5.4

   `MRUCache` has been deprecated due to lack of use, to reduce
   maintenance.  Please choose another cache implementation that suits
   your needs.

.. autoclass:: RRCache(maxsize, choice=random.choice, getsizeof=None)
   :members: choice, popitem

   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
   :func:`random.choice`.  The optional argument `choice` may specify
   an alternative function that returns an arbitrary element from a
   non-empty sequence.

.. autoclass:: TTLCache(maxsize, ttl, timer=time.monotonic, getsizeof=None)
   :members: popitem, timer, ttl

   This class associates a time-to-live value with each item.  Items
   that expire because they have exceeded their time-to-live will be
   no longer accessible, and will be removed eventually.  If no
   expired items are there to remove, the least recently used items
   will be discarded first to make space when necessary.

   By default, the time-to-live is specified in seconds and
   :func:`time.monotonic` is used to retrieve the current time.

   .. testcode::

      cache = TTLCache(maxsize=10, ttl=60)

   A custom `timer` function can also be supplied, which does not have
   to return seconds, or even a numeric value.  The expression
   `timer() + ttl` at the time of insertion defines the expiration
   time of a cache item and must be comparable against later results
   of `timer()`, but `ttl` does not necessarily have to be a number,
   either.

   .. testcode::

      from datetime import datetime, timedelta

      cache = TTLCache(maxsize=10, ttl=timedelta(hours=12), timer=datetime.now)

   .. method:: expire(self, time=None)
```

```
      Expired items will be removed from a cache only at the next
      mutating operation, e.g. :meth:`__setitem__` or
      :meth:`__delitem__`, and therefore may still claim memory.
      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 :const:`None`, this removes all
      items that have expired by the current value returned by
      :attr:`timer`.

      :returns: An iterable of expired `(key, value)` pairs.

.. autoclass:: TLRUCache(maxsize, ttu, timer=time.monotonic, getsizeof=None)
   :members: popitem, timer, ttu

   Similar to :class:`TTLCache`, this class also associates an
   expiration time with each item.  However, for :class:`TLRUCache`
   items, expiration time is calculated by a user-provided time-to-use
   (`ttu`) function, which is passed three arguments at the time of
   insertion: the new item's key and value, as well as the current
   value of `timer()`.

   .. testcode::

      def my_ttu(_key, value, now):
          # assume value.ttu contains the item's time-to-use in seconds
          # note that the _key argument is ignored in this example
          return now + value.ttu

      cache = TLRUCache(maxsize=10, ttu=my_ttu)

   The expression `ttu(key, value, timer())` defines the expiration
   time of a cache item, and must be comparable against later results
   of `timer()`.  As with :class:`TTLCache`, a custom `timer` function
   can be supplied, which does not have to return a numeric value.

   .. testcode::

      from datetime import datetime, timedelta

      def datetime_ttu(_key, value, now):
          # assume now to be of type datetime.datetime, and
          # value.hours to contain the item's time-to-use in hours
          return now + timedelta(hours=value.hours)

      cache = TLRUCache(maxsize=10, ttu=datetime_ttu, timer=datetime.now)

   Items that expire because they have exceeded their time-to-use will
   be no longer accessible, and will be removed eventually.  If no
   expired items are there to remove, the least recently used items
   will be discarded first to make space when necessary.

   .. method:: expire(self, time=None)

      Expired items will be removed from a cache only at the next
      mutating operation, e.g. :meth:`__setitem__` or
      :meth:`__delitem__`, and therefore may still claim memory.
      Calling this method removes all items whose time-to-use would
      have expired by `time`, so garbage collection is free to reuse
      their memory.  If `time` is :const:`None`, this removes all
      items that have expired by the current value returned by
      :attr:`timer`.

      :returns: An iterable of expired `(key, value)` pairs.


Extending cache classes
=======================

Sometimes it may be desirable to notice when and what cache items are
evicted, i.e. removed from a cache to make room for new items.  Since
all cache implementations call :meth:`popitem` to evict items from the
cache, this can be achieved by overriding this method in a subclass:

.. doctest::
```

```
   :pyversion: >= 3

   >>> class MyCache(LRUCache):
   ...     def popitem(self):
   ...         key, value = super().popitem()
   ...         print('Key "%s" evicted with value "%s"' % (key, value))
   ...         return key, value

   >>> c = MyCache(maxsize=2)
   >>> c['a'] = 1
   >>> c['b'] = 2
   >>> c['c'] = 3
   Key "a" evicted with value "1"
```

With :class:`TTLCache` and :class:`TLRUCache`, items may also be removed after they expire. In this case, :meth:`popitem` will *not* be called, but :meth:`expire` will be called from the next mutating operation and will return an iterable of the expired `(key, value)` pairs. By overrding :meth:`expire`, a subclass will be able to track expired items:

```
.. doctest::
   :pyversion: >= 3

   >>> class ExpCache(TTLCache):
   ...     def expire(self, time=None):
   ...         items = super().expire(time)
   ...         print(f"Expired items: {items}")
   ...         return items

   >>> c = ExpCache(maxsize=10, ttl=1.0)
   >>> c['a'] = 1
   Expired items: []
   >>> c['b'] = 2
   Expired items: []
   >>> time.sleep(1.5)
   >>> c['c'] = 3
   Expired items: [('a', 1), ('b', 2)]
```

Similar to the standard library's :class:`collections.defaultdict`, subclasses of :class:`Cache` may implement a :meth:`__missing__` method which is called by :meth:`Cache.__getitem__` if the requested key is not found:

```
.. doctest::
   :pyversion: >= 3

   >>> class PepStore(LRUCache):
   ...     def __missing__(self, key):
   ...         """Retrieve text of a Python Enhancement Proposal"""
   ...         url = 'http://www.python.org/dev/peps/pep-%04d/' % key
   ...         with urllib.request.urlopen(url) as s:
   ...             pep = s.read()
   ...             self[key] = pep  # store text in cache
   ...             return pep

   >>> peps = PepStore(maxsize=4)
   >>> for n in 8, 9, 290, 308, 320, 8, 218, 320, 279, 289, 320:
   ...     pep = peps[n]
   >>> print(sorted(peps.keys()))
   [218, 279, 289, 320]
```

Note, though, that such a class does not really behave like a *cache* any more, and will lead to surprising results when used with any of the memoizing decorators described below. However, it may be useful in its own right.

Memoizing decorators

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

.. doctest::

   >>> @cached(cache={})
   ... def fib(n):
   ...   'Compute the nth number in the Fibonacci sequence'
   ...   return n if n < 2 else fib(n - 1) + fib(n - 2)

   >>> fib(42)
   267914296

.. decorator:: cached(cache, key=cachetools.keys.hashkey, lock=None, info=False)

   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
   :mod:`cachetools` module. :func:`cached` will work with any
   mutable mapping type, including plain :class:`dict` and
   :class:`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 :func:`cachetools.keys.hashkey`.

   If `lock` is not :const:`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
   :class:`threading.Lock` 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 decorator's `cache`, `key` and `lock` parameters are also
   available as :attr:`cache`, :attr:`cache_key` and
   :attr:`cache_lock` attributes of the memoizing wrapper function.
   These can be used for clearing the cache or invalidating individual
   cache items, for example.

   .. testcode::

      from threading import Lock

      # 640K should be enough for anyone...
      @cached(cache=LRUCache(maxsize=640*1024, getsizeof=len), lock=Lock())
      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()

      # make sure access to cache is synchronized
      with get_pep.cache_lock:
          get_pep.cache.clear()

      # always use the key function for accessing cache items
      with get_pep.cache_lock:
          get_pep.cache.pop(get_pep.cache_key(42), None)

   For the common use case of clearing or invalidating the cache, the
   decorator also provides a :func:`cache_clear()` function which
   takes care of locking automatically, if needed:

   .. testcode::

      # no need for get_pep.cache_lock here

```
   get_pep.cache_clear()

If `info` is set to :const:`True`, the wrapped function is
instrumented with a :func:`cache_info()` function that returns a
named tuple showing `hits`, `misses`, `maxsize` and `currsize`, to
help measure the effectiveness of the cache.

.. note::

   Note that this will inflict a - probably minor - performance
   penalty, so it has to be explicitly enabled.

.. doctest::
   :pyversion: >= 3

   >>> @cached(cache=LRUCache(maxsize=32), info=True)
   ... def get_pep(num):
   ...     url = 'http://www.python.org/dev/peps/pep-%04d/' % num
   ...     with urllib.request.urlopen(url) as s:
   ...         return s.read()

   >>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
   ...     pep = get_pep(n)

   >>> get_pep.cache_info()
   CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)

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

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:

.. doctest::
   :options: +ELLIPSIS

   >>> from cachetools.keys import hashkey
   >>> from functools import partial

   >>> # shared cache for integer sequences
   >>> numcache = {}

   >>> # compute Fibonacci numbers
   >>> @cached(numcache, key=partial(hashkey, 'fib'))
   ... def fib(n):
   ...     return n if n < 2 else fib(n - 1) + fib(n - 2)

   >>> # compute Lucas numbers
   >>> @cached(numcache, key=partial(hashkey, 'luc'))
   ... def luc(n):
   ...     return 2 - n if n < 2 else luc(n - 1) + luc(n - 2)

   >>> fib(42)
   267914296
   >>> luc(42)
   599074578
   >>> list(sorted(numcache.items()))
   [..., (('fib', 42), 267914296), ..., (('luc', 42), 599074578)]


.. decorator:: cachedmethod(cache, key=cachetools.keys.methodkey, lock=None)

   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 :func:`cached` function
   decorator is that `cache` and `lock` are not passed objects, but
   functions.  Both will be called with :const:`self` (or :const:`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 :func:`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.

The `key` function will be called as `key(self, *args, **kwargs)`
to retrieve a suitable cache key.  Note that the default `key`
function, :func:`cachetools.keys.methodkey`, ignores its first
argument, i.e. :const:`self`.  This has mostly historical reasons,
but also ensures that :const:`self` does not have to be hashable.
You may provide a different `key` function,
e.g. :func:`cachetools.keys.hashkey`, if you need :const:`self` to
be part of the cache key.

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

.. testcode::

   class CachedPEPs:

       def __init__(self, cachesize):
           self.cache = LRUCache(maxsize=cachesize)

       @cachedmethod(lambda self: self.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))

.. testoutput::
   :hide:
   :options: +ELLIPSIS

   PEP #1: ...

When using a shared cache for multiple methods, be aware that
different cache keys must be created for each method even when
function arguments are the same, just as with the `@cached`
decorator:

.. testcode::

   class CachedReferences:

       def __init__(self, cachesize):
           self.cache = LRUCache(maxsize=cachesize)

       @cachedmethod(lambda self: self.cache, key=partial(hashkey, 'pep'))
       def get_pep(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()

       @cachedmethod(lambda self: self.cache, key=partial(hashkey, 'rfc'))
       def get_rfc(self, num):
           """Retrieve text of an IETF Request for Comments"""
           url = 'https://tools.ietf.org/rfc/rfc%d.txt' % num
           with urllib.request.urlopen(url) as s:
               return s.read()

   docs = CachedReferences(cachesize=100)
   print("PEP #1: %s" % docs.get_pep(1))
   print("RFC #1: %s" % docs.get_rfc(1))

.. testoutput::
```

```
      :hide:
      :options: +ELLIPSIS

      PEP #1: ...
      RFC #1: ...


****************************************************************
:mod:`cachetools.keys` --- Key functions for memoizing decorators
****************************************************************

.. module:: cachetools.keys

This module provides several functions that can be used as key
functions with the :func:`cached` and :func:`cachedmethod` decorators:

.. autofunction:: hashkey

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

.. autofunction:: methodkey

   This function is similar to :func:`hashkey`, but ignores its
   first positional argument, i.e. `self` when used with the
   :func:`cachedmethod` decorator.

.. autofunction:: typedkey

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

.. autofunction:: typedmethodkey

   This function is similar to :func:`typedkey`, but ignores its
   first positional argument, i.e. `self` when used with the
   :func:`cachedmethod` decorator.

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 :class:`TypeError`, since :class:`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(sorted(env.items()))
      return key

The :func:`envkey` function can then be used in decorator declarations
like this::

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

  foo(1, 2, 3, env=dict(a='a', b='b'))


************************************************************************
:mod:`cachetools.func` --- :func:`functools.lru_cache` compatible decorators
************************************************************************

.. module:: cachetools.func
```

To ease migration from (or to) Python 3's :func:`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. If `maxsize` is set to :const:`None`,
the caching strategy is effectively disabled and the cache can grow
without bound.

If the optional argument `typed` is set to :const:`True`, function
arguments of different types will be cached separately. For example,
``f(3)`` and ``f(3.0)`` will be treated as distinct calls with
distinct results.

If a `user_function` is specified instead, it must be a callable.
This allows the decorator to be applied directly to a user function,
leaving the `maxsize` at its default value of 128::

  @cachetools.func.lru_cache
  def count_vowels(sentence):
      sentence = sentence.casefold()
      return sum(sentence.count(vowel) for vowel in 'aeiou')

The wrapped function is instrumented with a :func:`cache_parameters`
function that returns a new :class:`dict` showing the values for
`maxsize` and `typed`. This is for information purposes only.
Mutating the values has no effect.

The wrapped function is also instrumented with :func:`cache_info` and
:func:`cache_clear` functions to provide information about cache
performance and clear the cache. Please see the
:func:`functools.lru_cache` documentation for details. Also note that
all the decorators in this module are thread-safe by default.

.. decorator:: fifo_cache(user_function)
               fifo_cache(maxsize=128, typed=False)

   Decorator that wraps a function with a memoizing callable that
   saves up to `maxsize` results based on a First In First Out
   (FIFO) algorithm.

.. decorator:: lfu_cache(user_function)
               lfu_cache(maxsize=128, typed=False)

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

.. decorator:: lru_cache(user_function)
               lru_cache(maxsize=128, typed=False)

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

.. decorator:: mru_cache(user_function)
               mru_cache(maxsize=128, typed=False)

   Decorator that wraps a function with a memoizing callable that
   saves up to `maxsize` results based on a Most Recently Used (MRU)
   algorithm.

   .. deprecated:: 5.4

   The `mru_cache` decorator has been deprecated due to lack of use.
   Please choose a decorator based on some other algorithm.

.. decorator:: rr_cache(user_function)
               rr_cache(maxsize=128, choice=random.choice, typed=False)

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

```
.. decorator:: ttl_cache(user_function)
               ttl_cache(maxsize=128, ttl=600, timer=time.monotonic, typed=False)

   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.


.. _@lru_cache: http://docs.python.org/3/library/functools.html#functools.lru_cache
.. _cache algorithm: http://en.wikipedia.org/wiki/Cache_algorithms
.. _context manager: http://docs.python.org/dev/glossary.html#term-context-manager
.. _mapping: http://docs.python.org/dev/glossary.html#term-mapping
.. _mutable: http://docs.python.org/dev/glossary.html#term-mutable
```

`cachetools` — Extensible memoizing collections and decorators

This module provides various memoizing collections and decorators, including variants of the Python Standard Library's @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.

This module provides multiple cache classes based on different cache algorithms, as well as decorators for easily memoizing function and method calls.

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, `Cache.__setitem__()` calls `self.popitem()` repeatedly until there is enough room for the item to be added.

In general, a cache's size is the total size of its item's values. Therefore, `Cache` provides 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.

Note that the values of a `Cache` are mutable by default, as are e.g. the values of a `dict`. It is the user's responsibility to take care that cached values are not accidentally modified. This is especially important when using a custom *getsizeof* function, since the size of an item's value will only be computed when the item is inserted into the cache.

> **Note:** Please be aware that all these classes are *not* thread-safe. Access to a shared cache from multiple threads must be properly synchronized, e.g. by using one of the memoizing decorators with a suitable *lock* object.

class cachetools.**Cache**(*maxsize*, *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__()`.

property **currsize**

The current size of the cache.

static **getsizeof**(*value*)

Return the size of a cache element's value.

property **maxsize**

The maximum size of the cache.

class cachetools.**FIFOCache**(*maxsize*, *getsizeof=None*)

First In First Out (FIFO) cache implementation.

This class evicts items in the order they were added to make space when necessary.

popitem()

Remove and return the *(key, value)* pair first inserted.

class cachetools.**LFUCache**(*maxsize*, *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*, *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.**MRUCache**(*maxsize*, *getsizeof=None*)

Most Recently Used (MRU) cache implementation.

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

Deprecated since version 5.4.

MRUCache has been deprecated due to lack of use, to reduce maintenance. Please choose another cache implementation that suits your needs.

popitem()

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

class cachetools.**RRCache**(*maxsize*, *choice=random.choice*, *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.

property **choice**

The *choice* function used by the cache.

popitem()

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

class cachetools.**TTLCache**(*maxsize*, *ttl*, *timer=time.monotonic*, *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 no longer accessible, and will be removed eventually. If no expired items are there to remove, the least recently used items will be discarded first to make space when necessary.

By default, the time-to-live is specified in seconds and **`time.monotonic()`** is used to retrieve the current time.

```
cache = TTLCache(maxsize=10, ttl=60)
```

A custom *timer* function can also be supplied, which does not have to return seconds, or even a numeric value. The expression *timer() + ttl* at the time of insertion defines the expiration time of a cache item and must be comparable against later results of *timer()*, but *ttl* does not necessarily have to be a number, either.

```
from datetime import datetime, timedelta

cache = TTLCache(maxsize=10, ttl=timedelta(hours=12), timer=datetime.now)
```

expire(*self*, *time=None*)

Expired items will be removed from a cache only at the next mutating operation, e.g. **`__setitem__()`** or **`__delitem__()`**, and therefore may still claim memory. 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`.

Returns: An iterable of expired *(key, value)* pairs.

popitem()

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

property **timer**

The timer function used by the cache.

property **ttl**

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

class cachetools.**TLRUCache**(*maxsize*, *ttu*, *timer=time.monotonic*, *getsizeof=None*)

Time aware Least Recently Used (TLRU) cache implementation.

Similar to `TTLCache`, this class also associates an expiration time with each item. However, for `TLRUCache` items, expiration time is calculated by a user-provided time-to-use (*ttu*) function, which is passed three arguments at the time of insertion: the new item's key and value, as well as the current value of *timer()*.

```
def my_ttu(_key, value, now):
    # assume value.ttu contains the item's time-to-use in seconds
    # note that the _key argument is ignored in this example
    return now + value.ttu
```

```
cache = TLRUCache(maxsize=10, ttu=my_ttu)
```

The expression *ttu(key, value, timer())* defines the expiration time of a cache item, and must be comparable against later results of *timer()*. As with `TTLCache`, a custom *timer* function can be supplied, which does not have to return a numeric value.

```
from datetime import datetime, timedelta

def datetime_ttu(_key, value, now):
    # assume now to be of type datetime.datetime, and
    # value.hours to contain the item's time-to-use in hours
    return now + timedelta(hours=value.hours)

cache = TLRUCache(maxsize=10, ttu=datetime_ttu, timer=datetime.now)
```

Items that expire because they have exceeded their time-to-use will be no longer accessible, and will be removed eventually. If no expired items are there to remove, the least recently used items will be discarded first to make space when necessary.

expire(*self, time=None*)

Expired items will be removed from a cache only at the next mutating operation, e.g. **`__setitem__()`** or **`__delitem__()`**, and therefore may still claim memory. Calling this method removes all items whose time-to-use 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`.

Returns: An iterable of expired *(key, value)* pairs.

popitem()

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

property **timer**

The timer function used by the cache.

property **ttu**

The local time-to-use function used by the cache.

Extending cache classes

Sometimes it may be desirable to notice when and what cache items are evicted, i.e. removed from a cache to make room for new items. Since all cache implementations call **`popitem()`** to evict items from the cache, this can be achieved by overriding this method in a subclass:

```
>>> class MyCache(LRUCache):
...     def popitem(self):
...         key, value = super().popitem()
...         print('Key "%s" evicted with value "%s"' % (key, value))
...         return key, value

>>> c = MyCache(maxsize=2)
>>> c['a'] = 1
>>> c['b'] = 2
>>> c['c'] = 3
Key "a" evicted with value "1"
```


With `TTLCache` and `TLRUCache`, items may also be removed after they expire. In this case, **`popitem()`** will *not* be called, but **`expire()`** will be called from the next mutating operation and will return an iterable of the

expired *(key, value)* pairs. By overrding **`expire()`**, a subclass will be able to track expired items:

```
>>> class ExpCache(TTLCache):
...     def expire(self, time=None):
...         items = super().expire(time)
...         print(f"Expired items: {items}")
...         return items

>>> c = ExpCache(maxsize=10, ttl=1.0)
>>> c['a'] = 1
Expired items: []
>>> c['b'] = 2
Expired items: []
>>> time.sleep(1.5)
>>> c['c'] = 3
Expired items: [('a', 1), ('b', 2)]
```

Similar to the standard library's **`collections.defaultdict`**, subclasses of `Cache` may implement a **`__missing__()`** method which is called by **`Cache.__getitem__()`** if the requested key is not found:

```
>>> class PepStore(LRUCache):
...     def __missing__(self, key):
...         """Retrieve text of a Python Enhancement Proposal"""
...         url = 'http://www.python.org/dev/peps/pep-%04d/' % key
...         with urllib.request.urlopen(url) as s:
...             pep = s.read()
...             self[key] = pep  # store text in cache
...             return pep

>>> peps = PepStore(maxsize=4)
>>> for n in 8, 9, 290, 308, 320, 8, 218, 320, 279, 289, 320:
...     pep = peps[n]
>>> print(sorted(peps.keys()))
[218, 279, 289, 320]
```

Note, though, that such a class does not really behave like a *cache* any more, and will lead to surprising results when used with any of the memoizing decorators described below. However, it may be useful in its own right.

Memoizing 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:

```
>>> @cached(cache={})
... def fib(n):
...     'Compute the nth number in the Fibonacci sequence'
...     return n if n < 2 else fib(n - 1) + fib(n - 2)

>>> fib(42)
267914296
```

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

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 plai

`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 `cachetools.keys.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.Lock` 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 decorator's *cache*, *key* and *lock* parameters are also available as `cache`, `cache_key` and `cache_lock` attributes of the memoizing wrapper function. These can be used for clearing the cache or invalidating individual cache items, for example.

```
from threading import Lock

# 640K should be enough for anyone...
@cached(cache=LRUCache(maxsize=640*1024, getsizeof=len), lock=Lock())
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()

# make sure access to cache is synchronized
with get_pep.cache_lock:
    get_pep.cache.clear()

# always use the key function for accessing cache items
with get_pep.cache_lock:
    get_pep.cache.pop(get_pep.cache_key(42), None)
```

For the common use case of clearing or invalidating the cache, the decorator also provides a `cache_clear()` function which takes care of locking automatically, if needed:

```
# no need for get_pep.cache_lock here
get_pep.cache_clear()
```

If *info* is set to `True`, the wrapped function is instrumented with a `cache_info()` function that returns a named tuple showing *hits*, *misses*, *maxsize* and *currsize*, to help measure the effectiveness of the cache.

> **Note:** Note that this will inflict a - probably minor - performance penalty, so it has to be explicitly enabled.

```
>>> @cached(cache=LRUCache(maxsize=32), info=True)
... def get_pep(num):
...     url = 'http://www.python.org/dev/peps/pep-%04d/' % num
...     with urllib.request.urlopen(url) as s:
...         return s.read()

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

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

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 cachetools.keys import hashkey
>>> from functools import partial

>>> # shared cache for integer sequences
>>> numcache = {}

>>> # compute Fibonacci numbers
>>> @cached(numcache, key=partial(hashkey, 'fib'))
... def fib(n):
...     return n if n < 2 else fib(n - 1) + fib(n - 2)

>>> # compute Lucas numbers
>>> @cached(numcache, key=partial(hashkey, 'luc'))
... def luc(n):
...     return 2 - n if n < 2 else luc(n - 1) + luc(n - 2)

>>> fib(42)
267914296
>>> luc(42)
599074578
>>> list(sorted(numcache.items()))
[..., (('fib', 42), 267914296), ..., (('luc', 42), 599074578)]
```

@cachetools.**cachedmethod**(*cache*, *key=cachetools.keys.methodkey*, *lock=None*)

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.

The *key* function will be called as *key(self, *args, **kwargs)* to retrieve a suitable cache key. Note that the default *key* function, `cachetools.keys.methodkey()`, ignores its first argument, i.e. **self**. This has mostly historical reasons, but also ensures that **self** does not have to be hashable. You may provide a different *key* function, e.g. `cachetools.keys.hashkey()`, if you need **self** to be part of the cache key.

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

```
class CachedPEPs:

    def __init__(self, cachesize):
        self.cache = LRUCache(maxsize=cachesize)

    @cachedmethod(lambda self: self.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))
```

When using a shared cache for multiple methods, be aware that different cache keys must be created for each method even when function arguments are the same, just as with the *@cached* decorator:

```
class CachedReferences:

    def __init__(self, cachesize):
        self.cache = LRUCache(maxsize=cachesize)

    @cachedmethod(lambda self: self.cache, key=partial(hashkey, 'pep'))
    def get_pep(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()

    @cachedmethod(lambda self: self.cache, key=partial(hashkey, 'rfc'))
    def get_rfc(self, num):
        """Retrieve text of an IETF Request for Comments"""
        url = 'https://tools.ietf.org/rfc/rfc%d.txt' % num
        with urllib.request.urlopen(url) as s:
            return s.read()

docs = CachedReferences(cachesize=100)
print("PEP #1: %s" % docs.get_pep(1))
print("RFC #1: %s" % docs.get_rfc(1))
```

`cachetools.keys` — Key functions for memoizing decorators

This module provides several functions that can be used as key functions with the **`cached()`** and **`cachedmethod()`** decorators:

cachetools.keys.**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.

cachetools.keys.**methodkey**(*self*, **args*, ***kwargs*)

Return a cache key for use with cached methods.

This function is similar to **`hashkey()`**, but ignores its first positional argument, i.e. *self* when used with the **`cachedmethod()`** decorator.

cachetools.keys.**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.

cachetools.keys.**typedmethodkey**(*self*, **args*, ***kwargs*)

Return a typed cache key for use with cached methods.

This function is similar to **`typedkey()`**, but ignores its first positional argument, i.e. *self* w

the **`cachedmethod()`** decorator.

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(sorted(env.items()))
    return key
```

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

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

foo(1, 2, 3, env=dict(a='a', b='b'))
```

`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. If *maxsize* is set to **`None`**, the caching strategy is effectively disabled and the cache can grow without bound.

If the optional argument *typed* is set to **`True`**, function arguments of different types will be cached separately. For example, `f(3)` and `f(3.0)` will be treated as distinct calls with distinct results.

If a *user_function* is specified instead, it must be a callable. This allows the decorator to be applied directly to a user function, leaving the *maxsize* at its default value of 128:

```
@cachetools.func.lru_cache
def count_vowels(sentence):
    sentence = sentence.casefold()
    return sum(sentence.count(vowel) for vowel in 'aeiou')
```

The wrapped function is instrumented with a **`cache_parameters()`** function that returns a new **`dict`** showing the values for *maxsize* and *typed*. This is for information purposes only. Mutating the values has no effect.

The wrapped function is also instrumented with **`cache_info()`** and **`cache_clear()`** functions to provide information about cache performance and clear the cache. Please see the **`functools.lru_cache()`** documentation for details. Also note that all the decorators in this module are thread-safe by default.

`@cachetools.func.`**`fifo_cache`**`(`*`user_function`*`)`
`@cachetools.func.`**`fifo_cache`**`(`*`maxsize=128, typed=False`*`)`

Decorator that wraps a function with a memoizing callable that saves up to *maxsize* results based on a First In First Out (FIFO) algorithm.

`@cachetools.func.`**`lfu_cache`**`(`*`user_function`*`)`

@cachetools.func.**lfu_cache**(*maxsize=128*, *typed=False*)

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

@cachetools.func.**lru_cache**(*user_function*)
@cachetools.func.**lru_cache**(*maxsize=128*, *typed=False*)

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

@cachetools.func.**mru_cache**(*user_function*)
@cachetools.func.**mru_cache**(*maxsize=128*, *typed=False*)

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

Deprecated since version 5.4.

The *mru_cache* decorator has been deprecated due to lack of use. Please choose a decorator based on some other algorithm.

@cachetools.func.**rr_cache**(*user_function*)
@cachetools.func.**rr_cache**(*maxsize=128*, *choice=random.choice*, *typed=False*)

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

@cachetools.func.**ttl_cache**(*user_function*)
@cachetools.func.**ttl_cache**(*maxsize=128*, *ttl=600*, *timer=time.monotonic*, *typed=False*)

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.