Timed Dictionary Documentation

Release 0.1.0

Zachary Ernst

Oct 11, 2018

CONTENTS:

CHAPTER

ONE

TIMED DICTIONARY MODULE

This is the `TimedDict` class and its various methods and helpers.

`TimedDict` instances are dictionaries (`MutableMapping` objects, to be more precise) whose keys expire after a predetermined time has elapsed. When they expire, a callback function may optionally be executed. The callback function is passed the key and value that have just expired.

Using the class is pretty simple.

```
import time
from timed_dict.timed_dict import TimedDict

d = TimedDict(timeout=10)
d['foo'] = 'bar'

print('Retrieving the key right away...')
print(d['foo'])
print('Waiting 11 seconds...')
time.sleep(11)
print('The key is expired, so we get an `Empty()` object.')
print(d['foo'])
```

Running this code produces the following output:

```
Retrieving the key right away...
bar
Waiting 11 seconds...
The key is expired, so we get an `Empty()` object.
<timed_dict.timed_dict.Empty object at 0x7ff82003a860>
```

That's the simplest use-case. Often, you'll want something to happen when a key expires. For this, you specify a callback function, like so:

```
import time
from timed_dict.timed_dict import TimedDict

def my_callback(key, value):  # Key and value are required arguments
    print('The key {key} has expired. Its value was {value}.'.format(
        key=str(key), value=str(value)))


d = TimedDict(timeout=10, callback=my_callback)  # Specify callback here
d['foo'] = 'bar'

print('Retrieving the key right away..')
```

(continues on next page)

(continued from previous page)

```
print(d['foo'])
print('Waiting 11 seconds...')
time.sleep(11)
print('The key is expired, so we get an `Empty()` object.')
print(d['foo'])
```

Running the new code, with the callback function, produces this:

```
Retrieving the key right away..
bar
Waiting 11 seconds...
The key foo has expired. Its value was bar.
The key is expired, so we get an `Empty()` object.
<timed_dict.timed_dict.Empty object at 0x7f0a56cfccf8>
```

As you can see, the behavior is identical, except that we now see the output from the callback function (on the fourth line).

Other arguments may be passed to the `TimedDict` constructor. They are documented in its `__init__` function below.

class timed_dict.timed_dict.**Empty**

Bases: `object`

Just to provide a unique default class when asking for a key that's been deleted.

__weakref__

list of weak references to the object (if defined)

class timed_dict.timed_dict.**TimedDict**(*timeout=None*, *checks_per_second=0.5*, *sample_probability=0.25*, *callback=None*, *expired_keys_ratio=0.25*, *sweep_flag=True*, *callback_args=None*, *callback_kwargs=None*)

Bases: `collections.abc.MutableMapping`

A dictionary whose keys time out. After a pre-determined number of seconds, the key and value will be deleted from the dictionary. Optionally, a callback function is executed, which is passed the key and the associated value.

When it is instantiated, this class creates a thread which runs all the time, looking for expired keys. Each `TimedDict` object gets its own thread.

The algorithm is the same one that Redis uses. It is semi-lazy and probabilistic. After sleeping for a set interval, it iterates through a random sample of the keys (which is determined by the `sample_probability` kwarg in the class constructor). It expires any keys it finds during the sweep which have existed for more then `timeout` seconds. If at least `expired_keys_ratio` of the sampled keys have to be expired, then the process is repeated again immediately. If not, then it sleeps for the interval again before restarting.

Additionally, a check is made to any specific key that's accessed. If the key should be expired, then it does so and returns an `Empty` object.

__delitem__(*key*)

Deletes the key and value from both the `base_dict` and the `timed_dict`.

__getitem__(*key*)

Gets the item. Before it does so, checks whether the key has expired. If so, it expires the key **first**, before returning a value.

If the key does not exist (or gets expired during this call) the method returns an instance of the *Empty* class. We use the *Empty* class (defined above) because we want to avoid raising exceptions, but we also want to

allow that the legitimate value of a key might be *None* (or any other default value we like).

__init__(*timeout=None*, *checks_per_second=0.5*, *sample_probability=0.25*, *callback=None*, *expired_keys_ratio=0.25*, *sweep_flag=True*, *callback_args=None*, *callback_kwargs=None*)
Init function for `TimedDict`. This automatically creates and starts a thread which intermittantly sweeps through the keys, looking for things to expire.

Parameters

- **timeout** (*float*) – Number of seconds after which keys will expire.
- **checks_per_second** (*float*) – Approximstely how many timed per second to scan for expired keys.
- **sample_probability** (*float*) – Keys are scanned randomly. This is the proportion of keys that will be checked on any given sweep.
- **callback** (*function*) – Function to be executed whenever a key expires. The function will be passed the arguments `key` and `value` as (non-keyword) arguments.
- **expired_keys_ratio** (*float*) – If the proportion of expired keys that are discovered on a given sweep exceeds this number, then another sweep is executed immediately. This on the theory that if there are many keys to be expired within a random sample, then there are probably a lot more.
- **sweep_flag** (*bool*) – If `True`, the thread runs which starts sweeping through the keys. It is not normally necessary for the user to fuss with this.
- **callback_args** (*tuple*) – Positional arguments to be passed to the callback function.
- **callback_kwargs** (*dict*) – Dictionary of keyword arguments to be passed to the callback function.

__len__()
Returns the number of items currently in the `TimedDict`.

__repr__()
String representation of the `TimedDict`. It returns the keys, values, and time of expiration for each.

__setitem__(*key*, *value*)
Replaces the `__setitem__` from the parent class. Sets both the `base_dict` (which holds the values) and the `timed_dict` (which holds the expiration time.

__weakref__
list of weak references to the object (if defined)

expire_key(*key*)
Expire the key, delete the value, and call the callback function if one is specified.

Parameters **key** – The `TimedDict` key

keys()
Replaces the `keys` method. There's probably a better way to accomplish this.

set_expiration(*key*, *ignore_missing=False*, *additional_seconds=None*, *seconds=None*)
Alters the expiration time for a key. If the key is not present, then raise an Exception unless *ignore_missing* is set to *True*.

Parameters

- **key** – The key whose expiration we are changing.
- **ignore_missing** (*bool*) – If set, then return silently if the key does not exist. Default is *False*.

- **additional_seonds** (*int*) – Add this many seconds to the current expiration time.
- **seconds** (*int*) – Expire the key this many seconds from now.

stop_sweep()
Stops the thread that periodically tests the keys for expiration.

sweep()
This methods runs in a separate thread. So long as *self.sweep_flag* is set, it expires keys according to the process explained in the docstring for the *TimedDict* class. The thread is halted by calling *self.stop_sweep()*, which sets the *self.sweep_flag* to *False*.

values()
Replaces the `values` method. There's probably a better way to accomplish this.

timed_dict.timed_dict.**cleanup_sweep_threads**()
Not used. Keeping this function in case we decide not to use daemonized threads and it becomes necessary to clean up the running threads upon exit.

timed_dict.timed_dict.**my_callback**(*key*, *value*)
Simple test of callback function.

CHAPTER

TWO

OVERVIEW

2.1 What it is

"Timed Dictionary" provides a class – `TimedDict` – which is a dictionary whose keys time out. After a predetermined number of seconds, the key and value will be deleted from the dictionary. Optionally, a callback function may be specified, which is called whenever a key is expired.

When it is instantiated, this class creates a thread which runs all the time, looking for expired keys. Each `TimedDict` object gets its own thread.

The algorithm is the same one that Redis uses. It is semi-lazy and probabilistic. After sleeping for a set interval, it iterates through a random sample of the keys (which is determined by the `sample_probability` kwarg in the class constructor). It expires any keys it finds during the sweep which have existed for more then `timeout` seconds. If at least `expired_keys_ratio` of the sampled keys have to be expired, then the process is repeated again immediately. If not, then it sleeps for the interval again before restarting.

Additionally, a check is made to any specific key that's accessed. If the key should be expired, then it does so and returns an `Empty` object.

2.2 What it is not

This module is **not** a replacement for Redis, MEMCACHE, or any other such things. It is intended to be simple, lightweight, free of any infrastructure requirements, stand-alone, pure Python, and above all, functional. If you need failover, distributed caching, guarantees, a magical solution to the CAP theorem, or you're processing huge volumes of data, you should not use this.

2.3 Why it is

As a data engineer, I'm constantly coming across use-cases that look like they require heavyweight tools, but which aren't demanding enough to justify the investment. Redis, although it isn't exactly "heavyweight", requires a server to run, with a dedicated IP address, and so on for virtually any production use-case. So engineers who would like to have a key-value store with expiration and callbacks either roll their own one-off kludgey solution, or they stand up a Redis instance somewhere and have one more thing to worry about.

This module provides that core functionality, implemented as a Python dictionary. But it does not require anything other than the standard Python library to run. It is dead-simple, reliable, and tested.

2.4 Limitations

If you need strong guarantees or tremendous precision, you won't want to use this module. The expiration algorithm is probabilistic, so it's likely that keys could expire in a somewhat different order than they were added, for example. Keys could also hang around longer than their expiration time.

Generally, these slight imprecisions are not a big deal. Delays in expiring keys are usually not more than half a second or so. We're trading off a little precision for better performance and simplicity (which translates into reliability).

Then there's the GIL. This module uses threading instead of multiprocessing, which will send some people into fits of consternation and bitter indignation. But if your application is so demanding that the GIL really poses a problem, then you're probably better off using a more sophisticated tool, anyway. The GIL is fine. Really.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

C

E

K

M

S

T

V