# yamicache Documentation

***Release 0.4.0***

**Timothy McFadden**

**Oct 09, 2017**

# Contents

Contents:

CHAPTER 1

# yamicache

Yet another in-memory caching package

- Free software: MIT license
- Documentation: https://yamicache.readthedocs.io.

## Features

- Memoization
- Selective caching based on decorators
- Mutli-threaded support
- Optional garbage collection thread
- Optional time-based cache expiration

## Quick Start

```
from __future__ import print_function
import time
from yamicache import Cache
c = Cache()
class MyApp(object):
    @c.cached()
    def long_op(self):
            time.sleep(30)
            return 1

app = MyApp()
t_start = time.time()
```

```
assert app.long_op() == 1  # takes 30s
assert app.long_op() == 1  # takes 0s
assert app.long_op() == 1  # takes 0s
assert 1 < (time.time() - t_start) < 31
```

CHAPTER 2

# Installation

## Stable release

To install yamicache, run this command in your terminal:

```
$ pip install yamicache
```

This is the preferred method to install yamicache, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

## From sources

The sources for yamicache can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/mtik00/yamicache
```

Or download the tarball:

```
$ curl  -OL https://github.com/mtik00/yamicache/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

# Usage

To use yamicache in a project:

```
from yaimcache import Cache

app_cache = Cache()


@app_cache.cached
def square(var):
    return var ** 2


square(2)  # Will cache the first time
square(2)  # Cache hit
square(2)  # Cache hit
square(3)  # New cached item
square(3)  # Cache hit
app_cache.clear()
square(3)  # New cached item
```

**Caution:** You probably shouldn't indefinitely store really large objects if you don't really need to.

## Object Creation

In order to enable caching, you must first create a `Cache` object:

```
from yamicache import Cache
c = Cache()
```

The caching object has the following parameters available during object creation:

- `hashing (bool)`: This controls how default cache `keys` are created. By default, they key will hashed to make things a bit more *readable*.
- `key_join (str)`: This is the character used to join the different parts that make up the default key.
- `debug (bool)`: When `True`, `Cache.counters` will be enabled and cache hits will produce output on `stdout`.
- `prefix (str)`: All cache keys will use this prefix. Since the current implementation is instance-based, this is only helpful if dumping or comparing the cache to another instance.
- `quiet (bool)`: Don't print during `debug` cache hits
- `default_timeout (int)`: If > 0, all cached items will be considered stale this many seconds after they are cached. In that case, the function will be run again, cached, and a new timeout value will be created.
- `gc_thread_wait (int)`: The number of seconds in between cache *garbage collection*. The default, `None`, will disable the garbage collection thread. This parameter is only valid if `default_timeout` is > 0 (`ValueError` is raised otherwise).

## Decorators

### *@Cache.cached(key, timeout)*

This is the main decorator you use to cache the result from the function. *Yamicache* stores the result of the function and the function's inputs. Subsequent calls to this function, with the same inputs, will not call the function's code. *Yamicache* will return the function's result.

`key`: This input parameter can be used to specify the exact key you wish to store in the cache. This can make testing easier. You would normally leave this parameter blank. This will allow `yamicache` to build a key based on the function being called and the arguments being used.

> **Warning:** You cannot duplicate a key. Attempts to instantiated a cached object with the same key will raise `ValueError`.

`timeout`: You can use this parameter to override the default timeout value used by the `yamicache.Cache` object.

### *@Cache.clear_cache()*

This decorator can be used to force `Cache.clear()` whenever the function is called. This is handy when you call a function that should change the state of the object and its cache (e.g. creating a directory after you already cached the result of `ls`).

## Context Managers

*Yamicache* includes the following context managers to override default caching behavior.

### *override_timeout(cache_obj, timeout)*

This will override the timeout value set either by the `Cache` object or the cached decorator. For example:

```
from yamicache import Cache, override_timeout
c = Cache()

@c.cached(timeout=90)
def long_op():
    return 1

with override_timeout(c, timeout=5):
    long_op()
```

### *nocache(cache_obj)*

This will disable the default caching mechanism. The cache will **not** be modified when this context manager is used. For example:

```
from yamicache import Cache, nocache
c = Cache()

@c.cached(key='test')
def long_op(value):
    return value

long_op(1)  # First time; result will be cached
long_op(1)  # cached result will be returned

with nocache(c):
    long_op(1)  # Function code will be run; value will not affect cache
```

## Garbage Collection

You may want to periodically remove items from the cache that are no longer *valid* or stale. There are a few of ways to do this:

1. Periodically call `clear()`: This removes everything from the cache.
2. Periodically call `collect()`: This removes only items that exist and are *stale**
3. Create the object with non-zero `default_timeout` and non-zero `gc_thread_wait`: This will spawn a garbage collection thread that periodically calls `collect()` for you.

---

**Important:** Calling `collect()`, or using the garbage collection thread, is only valid when using a timeout value > 0

---

CHAPTER 4

# Contributing

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

You can contribute in many ways:

## Types of Contributions

### Report Bugs

Report bugs at https://github.com/mtik00/yamicache/issues.

If you are reporting a bug, please include:

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

### Fix Bugs

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

### Implement Features

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

### Write Documentation

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

### Submit Feedback

The best way to send feedback is to file an issue at https://github.com/mtik00/yamicache/issues.

If you are proposing a feature:

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

## Get Started!

Ready to contribute? Here's how to set up *yamicache* for local development.

1. Fork the *yamicache* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/yamicache.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv yamicache
$ cd yamicache/
$ pip install -e .
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests:

```
$ flake8 yamicache tests
$ py.test
```

   To get flake8, just pip install it into your virtualenv.

6. Commit your changes and push your branch to GitHub:

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

7. Submit a pull request through the GitHub website.

## Pull Request Guidelines

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

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4, 3.5, and 3.6, and for PyPy. Check https://travis-ci.org/mtik00/yamicache/pull_requests and make sure that the tests pass for all supported Python versions.

## Tips

To run a subset of tests:

```
$ py.test tests/test_class.py
```

CHAPTER 5

# History

## 0.4.0 (2017-10-09)

- Added `serialize()` and `deserialize()`

## 0.3.0 (2017-09-05)

- Added `@clear_cache()` decorator
- Added imports to allow for `from yamicache import Cache`

## 0.2.0 (2017-09-03)

- Added cache key collision checking

## 0.1.1 (2017-09-01)

- Fix #1: `Cache.cached()` ignores `timeout` parameter

## 0.1.0 (2017-08-28)

- First release on PyPI.

CHAPTER 6

# yamicache

## yamicache package

### Submodules

### yamicache.yamicache module

yamicache : Yet another in-memory cache module ('yami' sounds better to me than 'yaim')

This module provides a simple in-memory interface for caching results from function calls.

**class** `yamicache.yamicache.`**`Cache`**(*hashing=True*, *key_join='|'*, *debug=False*, *prefix=None*, *quiet=False*, *default_timeout=0*, *gc_thread_wait=None*)

Bases: `_abcoll.MutableMapping`

A class for caching and retreiving returns from function calls.

**Parameters**

- **`hashing`** (*`bool`*) – Whether or not to hash the function inputs when calculating the key. This helps keep the keys *readable*, especially for functions with many inputs.
- **`key_join`** (*`str`*) – The character used to join the different parts that make up the hash key.
- **`debug`** (*`bool`*) – When `True`, `Cache.counters` will be enabled and cache hits will produce output on `stdout`.
- **`prefix`** (*`str`*) – All cache keys will use this prefix. Since the current implementation is instance-based, this is only helpful if dumping or comparing the cache to another instance.
- **`quiet`** (*`bool`*) – Don't print during `debug` cache hits
- **`default_timeout`** (*`int`*) – If > 0, all cached items will be considered stale this many seconds after they are cached. In that case, the function will be run again, cached, and a new timeout value will be created.

- **gc_thread_wait** (*int*) – The number of seconds in between cache *garbage collection*. The default, `None`, will disable the garbage collection thread. This parameter is only valid if `default_timeout` is > 0 (`ValueError` is raised otherwise).

**cached**(*key=None*, *timeout=None*)
A decorator used to memoize the return of a function call.

**clear**()
Clear the cache

**clear_cache**()
A decorator used to clear the cache everytime the function is called.

For example, let's say you have a "discovery" function that stores data read by other functions, and those function use caching. You want to use `@c.clear_cache()` for your main function so you don't have to worry about cache being stale.

**collect**(*since=None*)
Clear any item from the cache that has timed out.

**deserialize**(*filename*)
Read the serialized cache data from a file.

**dump**()
Dump the entire cache as a JSON string

**items**()
Return all items in the cache as a list of `tuple(key, value)`

**keys**()
Return a list of keys in the cache

**pop**(*key*)
Remove the cached value specified by `key`

**popitem**()
Remove a random item from the cache (only useful during testing)

**serialize**(*filename*)
Serialize the cache to a filename. This process uses `pickle`; Do not use this function if you are caching something that is not picklable!

**values**()
Return a list of cached values

yamicache.yamicache.**nocache**(**args*, ***kwds*)
Use this context manager to temporarily disable all caching for an object.

Example:

```
>>> from yamicache import Cache, nocache
>>> c = Cache()
>>> @c.cached
... def test():
...     return 4
...
>>> with nocache(c):
...     test()
...
4
>>> print c.data_store
```

```
{}
>>>
```

yamicache.yamicache.**override_timeout**(**args*, ***kwds*)

## Module contents

Top-level package for yamicache.

CHAPTER 7

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## y

# Index