django-cache-memoize Documentation

Peter Bengtsson

Dec 22, 2018

Contents

CHAPTER 1

Key Features

- Memoized function calls can be invalidated.
- Works with non-trivial arguments and keyword arguments
- Insight into cache hits and cache missed with a callback.
- Ability to use as a "guard" for repeated execution when storing the function result isn't important or needed.

1.1 Installation

```
pip install django-cache-memoize
```

1.2 Usage

```
# Import the decorator
from cache_memoize import cache_memoize

# Attach decorator to cacheable function with a timeout of 100 seconds.
@cache_memoize(100)
def expensive_function(start, end):
    return random.randint(start, end)

# Just a regular Django view
def myview(request):
    # If you run this view repeatedly you'll get the same
    # output every time for 100 seconds.
    return http.HttpResponse(str(expensive_function(0, 100)))
```

The caching uses Django's default cache framework. Ultimately, it calls `django.core.cache.cache.set(cache_key, function_out, expiration)`. So if you have a function that returns something that can't be pickled and cached it won't work.

> For cases like this, Django exposes a simple, low-level cache API. You can use this API to store objects in the cache with any level of granularity you like. You can cache any Python object that can be pickled safely: strings, dictionaries, lists of model objects, and so forth. (Most common Python objects can be pickled; refer to the Python documentation for more information about pickling.)

See documentation.

1.3 Example Usage

This blog post: How to use django-cache-memoize

It demonstrates similarly to the above Usage example but with a little more detail. In particular it demonstrates the difference between *not* using `django-cache-memoize` and then adding it to your code after.

1.4 Advanced Usage

1.4.1 `args_rewrite`

Internally the decorator rewrites every argument and keyword argument to the function it wraps into a concatenated string. The first thing you might want to do is help the decorator rewrite the arguments to something more suitable as a cache key string. For example, suppose you have instances of a class whose `__str__` method doesn't return a unique value. For example:

```
class Record(models.Model):
    name = models.CharField(max_length=100)
    lastname = models.CharField(max_length=100)
    friends = models.ManyToManyField(SomeOtherModel)

    def __str__(self):
        return self.name

# Example use:
>>> record = Record.objects.create(name='Peter', lastname='Bengtsson')
>>> print(record)
Peter
>>> record2 = Record.objects.create(name='Peter', lastname='Different')
>>> print(record2)
Peter
```

This is a contrived example, but basically *you know* that the `str()` conversion of certain arguments isn't safe. Then you can pass in a callable called `args_rewrite`. It gets the same positional and keyword arguments as the function you're decorating. Here's an example implementation:

```
from cache_memoize import cache_memoize

def count_friends_args_rewrite(record):
    # The 'id' is always unique. Use that instead of the default __str__
    return record.id

@cache_memoize(100, args_rewrite=count_friends_args_rewrite)
def count_friends(record):
    # Assume this is an expensive function that can be memoize cached.
    return record.friends.all().count()
```

1.4.2 prefix

By default the prefix becomes the name of the function. Consider:

```
from cache_memoize import cache_memoize

@cache_memoize(10, prefix='randomness')
def function1():
    return random.random()

@cache_memoize(10, prefix='randomness')
def function2():  # different name, same arguments, same functionality
    return random.random()

# Example use
>>> function1()
0.39403406043780986
>>> function1()
0.39403406043780986
>>> # ^ repeated of course
>>> function2()
0.39403406043780986
>>> # ^ because the prefix was forcibly the same, the cache key is the same
```

1.4.3 hit_callable

If set, a function that gets called with the original argument and keyword arguments **if** the cache was able to find and return a cache hit. For example, suppose you want to tell your `statsd` server every time there's a cache hit.

```
from cache_memoize import cache_memoize

def _cache_hit(user, **kwargs):
    statsdthing.incr(f'cachehit:{user.id}', 1)

@cache_memoize(10, hit_callable=_cache_hit)
def calculate_tax(user, tax=0.1):
    return ...
```

1.4.4 miss_callable

Exact same functionality as `hit_callable` except the obvious difference that it gets called if it was *not* a cache hit.

1.4.5 store_result

This is useful if you have a function you want to make sure only gets called once per timeout expiration but you don't actually care that much about what the function return value was. Perhaps because you know that the function returns something that would quickly fill up your `memcached` or perhaps you know it returns something that can't be pickled. Then you can set `store_result` to `False`. This is equivalent to your function returning `True`.

```
from cache_memoize import cache_memoize

@cache_memoize(1000, store_result=False)
def send_tax_returns(user):
```

(continues on next page)

(continued from previous page)

```
    # something something time consuming
    ...
    return some_none_pickleable_thing

def myview(request):
    # View this view as much as you like the 'send_tax_returns' function
    # won't be called more than once every 1000 seconds.
    send_tax_returns(request.user)
```

1.4.6 cache_alias

The `cache_alias` argument allows you to use a cache other than the default.

```
# Given settings like:
# CACHES = {
#     'default': {...},
#     'other': {...},
# }

@cache_memoize(1000, cache_alias='other')
def myfunc(start, end):
    return random.random()
```

1.4.7 Cache invalidation

When you want to "undo" some caching done, you simple call the function again with the same arguments except you add `.invalidate` to the function.

```
from cache_memoize import cache_memoize

@cache_memoize(10)
def expensive_function(start, end):
    return random.randint(start, end)

>>> expensive_function(1, 100)
65
>>> expensive_function(1, 100)
65
>>> expensive_function(100, 200)
121
>>> exensive_function.invalidate(1, 200)
>>> expensive_function(1, 100)
89
>>> expensive_function(100, 200)
121
```

An "alias" of doing the same thing is to pass a keyword argument called `_refresh=True`. Like this:

```
# Continuing from the code block above
>>> expensive_function(100, 200)
121
>>> expensive_function(100, 200, _refresh=True)
177
```

(continues on next page)

(continued from previous page)

```
>>> expensive_function(100, 200)
177
```

There is no way to clear more than one cache key. In the above example, you had to know the "original arguments" when you wanted to invalidate the cache. There is no method "search" for all cache keys that match a certain pattern.

1.5 Compatibility

- Python 2.7, 3.4, 3.5, 3.6
- Django 1.8, 1.9, 1.10, 1.11, 2.0, 2.1

Check out the tox.ini file for more up-to-date compatibility by test coverage.

1.6 Prior Art

1.6.1 History

Mozilla Symbol Server is written in Django. It's a web service that sits between C++ debuggers and AWS S3. It shuffles symbol files in and out of AWS S3. Symbol files are for C++ (and other compiled languages) what sourcemaps are for JavaScript.

This service gets a LOT of traffic. The download traffic (proxying requests for symbols in S3) gets about ~40 requests per second. Due to the nature of the application most of these GETs result in a 404 Not Found but instead of asking AWS S3 for every single file, these lookups are cached in a highly configured Redis configuration. This Redis cache is also connected to the part of the code that uploads new files.

New uploads are arriving as zip file bundles of files, from Mozilla's build systems, at a rate of about 600MB every minute, each containing on average about 100 files each. When a new upload comes in we need to quickly be able find out if it exists in S3 and this gets cached since often the same files are repeated in different uploads. But when a file does get uploaded into S3 we need to quickly and confidently invalidate any local caches. That way you get to keep a really aggressive cache without any stale periods.

This is the use case `django-cache-memoize` was built for and tested in. It was originally written for Python 3.6 in Django 1.11 but when extracted, made compatible with Python 2.7 and as far back as Django 1.8.

`django-cache-memoize` is also used in SongSear.ch to cache short queries in the autocomplete search input. All autocomplete is done by Elasticsearch, which is amazingly fast, but not as fast as `memcached`.

1.6.2 "Competition"

There is already django-memoize by Thomas Vavrys. It too is available as a memoization decorator you use in Django. And it uses the default cache framework as a storage. It used `inspect` on the decorated function to build a cache key.

In benchmarks running both `django-memoize` and `django-cache-memoize` I found `django-cache-memoize` to be **~4 times faster** on average.

Another key difference is that `django-cache-memoize` uses `str()` and `django-memoize` uses `repr()` which in certain cases of mutable objects (e.g. class instances) as arguments the caching will not work. For example, this does *not* work in `django-memoize`:

```
from memoize import memoize

@memoize(60)
def count_user_groups(user):
    return user.groups.all().count()

def myview(request):
    # this will never be memoized
    print(count_user_groups(request.user))
```

However, this works...

```
from cache_memoize import cache_memoize

@cache_memoize(60)
def count_user_groups(user):
    return user.groups.all().count()

def myview(request):
    # this *will* work as expected
    print(count_user_groups(request.user))
```

1.7 Development

The most basic thing is to clone the repo and run:

```
pip install -e ".[dev]"
tox
```

1.7.1 Code style is all black

All code has to be formatted with Black and the best tool for checking this is therapist since it can help you run all, help you fix things, and help you make sure linting is passing before you git commit. This project also uses `flake8` to check other things Black can't check.

To check linting with `tox` use:

```
tox -e lint-py36
```

To install the `therapist` pre-commit hook simply run:

```
therapist install
```

When you run `therapist run` it will only check the files you've touched. To run it for all files use:

```
therapist run --use-tracked-files
```

And to fix all/any issues run:

```
therapist run --use-tracked-files --fix
```

CHAPTER 2

Contents

2.1 Authors

- Peter Bengtsson (@peterbe)
- Ben Spaulding (@benspaulding)

2.2 Changelog

2.2.1 0.1.5

- Fix when using `_refresh=False` and the `.invalidate()`. pull#19

2.2.2 0.1.4

- Ability to have the memoized function return `None` as an actual result. pull#9

2.2.3 0.1.3

- Ability to pass in your own custom cache-key callable function. Thanks @jaumebecks pull#10

2.2.4 0.1.2

- Ability to specify a different-than-default cache alias Thanks @benspaulding pull#6

2.2.5 0.1.1

- Package sit-ups. Main file not a package so it wasn't distributed.

0.1.0

- Basic version released.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search