
Cachual Documentation

Release 0.2.2

Alex Landau

January 17, 2017

1	Installation	3
2	Usage	5
3	How it Works	7
4	Key Generation	9
4.1	Caching Methods	10
5	Caching Different Data Types	11
5.1	Unpack	11
5.2	Pack	11
6	API Documentation	13
6.1	Caching	13
6.2	Packing and Unpacking Helpers	15
	Python Module Index	17

Cachual is a library that makes it easy to cache the return values from your Python functions with a simple decorator:

```
from cachual import RedisCache
cache = RedisCache()

@cache.cached(ttl=360)
def get_user_email(user_id):
    ...
```

Cachual currently supports Redis as the backing cache, but is very easy to extend to other caching backends.

- *Installation*
- *Usage*
- *How it Works*
- *Key Generation*
 - *Caching Methods*
- *Caching Different Data Types*
 - *Unpack*
 - *Pack*
- *API Documentation*
 - *Caching*
 - *Packing and Unpacking Helpers*

Installation

Install the library with pip:

```
$ pip install cachual
```

Usage

Initialize a cache using the desired implementation, providing the connection information in the constructor if different from the defaults:

```
from cachual import RedisCache
cache = RedisCache(host='localhost', port=1234, db=5)
```

The cache object gives you access to the `cached()` decorator, which you can apply to any function whose return value you want to cache:

```
@cache.cached()
def get_user_email(user_id):
    ...
```

If you don't specify a TTL then Cachual will use the server default, which is probably forever. In most cases you will want to specify a TTL, which is given in seconds:

```
@cache.cached(ttl=300) # 5 minutes
def get_user_email(user_id):
    ...
```

How it Works

When you decorate a function with the `cached()` decorator, before the function is executed, the cache will be checked for a prior return value for the **same function call**. This means that the cache key will be different if the function is called with different argument values.

If the value is found it is immediately returned. Otherwise, the function is executed and the return value is stored in the cache before returning it.

Any cache failures (get or put) are ignored; if the cache becomes nonfunctional and starts raising exceptions, your function will execute normally as if there was no cache.

Key Generation

Keys are generated by first appending the function name to the module where the function resides. After that one of four strings is appended:

1. If the function is called without arguments, then `()` is appended.
2. If the function is called only with positional arguments, then the unicode value of each argument is joined with `' , '` and surrounded by parentheses. For example:

```
'my.module.get_user_email(abc)'
```

3. If the function is called only with keyword arguments, they will be included in `key=value` form, where value is the unicode value of the argument, in alphabetical order of the keys. For example:

```
'my.module.get_user_emails(active=True, age=10, location=US)'
```

4. If the function is called with both, the positional argument values will be included first, and the keyword arguments will be included in alphabetical order of the keys after the positional argument values.

Finally, the entire (unicode) key is encoded as UTF-8 and hashed using MD5. This is to ensure that key format is uniform and consistent, because some backends (such as Memcached) have restrictions around cache keys, such as disallowing certain characters and size limits.

For Python 3, all strings are unicode and the default encoding is UTF-8; thus, the value for each argument will be coerced to unicode using the builtin `str` function. Python 2 is a bit more complicated; strings are bytes by default. If you pass a unicode value, that will be the value used for the cache key. If you pass a string literal, it will be converted to unicode (assuming UTF-8 encoding). Anything else will be converted to bytes (using the builtin `str` function) and then converted to unicode assuming UTF-8 encoding.

Arguments **should** be able to take on any value, but as a best practice, I highly recommend you only pass the basic types to your functions e.g. string, integer, float, etc. Even better, stick to unicode values for your strings regardless of what version of Python you're using.

Note: Because the unicode value of each argument is used to generate the cache key, you need to be careful that you are consistent in your function calls with respect to the types of your arguments. For example, if you have a function that takes a single argument like so:

```
def test(a):
    ...
```

Then these two calls will result in the same cache key:

```
test(5)
test("5")
```

Thus, you should not call your cached function with values for an argument that are different types but have the same unicode value. This is good practice anyway; mixing types for the same argument value can lead to unmaintainable code and unexpected bugs.

4.1 Caching Methods

New in version 0.2.2.

You will probably find yourself wanting to cache methods as well - both classmethods and instance methods. Calls to these methods will pass the calling class (in the former) or the object (in the latter) as the first argument.

For classmethods, you will get the correct behavior: calls to your cached method with the same arguments from the same class will get the same cache key. Calling the method with a different class will result in a different cache key (even if the arguments are the same).

Instance methods are a little trickier. By default you will get a different cache key for calls to your instance method with the same arguments *if you are calling from different instances*, because the default representation of an object in Python includes its memory location. This behavior may be undesirable in some situations, for example if you have a web service that generates a client object for an external API on every request. In this case you probably want the same cache key for any external API calls with the same arguments, even though there is a new client object each request.

To get the desired behavior you can set the `use_class_for_self` parameter to `True` to use the class representation instead of the instance object representation of the first parameter, which will use the same cache key for any calls as long as they are from instances of the same class:

```
class ExternalAPIClient(object):

    @cache.cached(use_class_for_self=True)
    def get_location_name_by_id(self, id):
        ...

ExternalAPIClient().get_location_name_by_id("test") # Stores in cache
ExternalAPIClient().get_location_name_by_id("test") # Cache hit
```

Caching Different Data Types

Technically the types of values that your cached functions can return depends on the underlying cache. For example, Redis's SET only stores strings; thus, if you wanted to return something other than a string from your cached function, you would have to convert the value into the desired datatype if it comes from a cache hit. To get around this limitation, you can provide two additional optional arguments to the `cached()` decorator.

5.1 Unpack

The `unpack` argument specifies a function which will be called with the value from the cache in the case of a cache hit. The result will be passed back to the caller.

In this way you can make sure you get back consistent values regardless of whether the value came from the cache (which may store every value as a string) or from the actual function call:

```
def get_int(value):
    return int(value)

@cache.cached(ttl=300, unpack=get_int)
def get_user_id(email):
    ...
```

5.2 Pack

What if you want to cache a more complex data type, like a list or a dictionary? You can provide a function to the `pack` argument, which will be applied before the function's return value is put into the cache:

```
@cache.cached(ttl=300, pack=json.dumps, unpack=json.loads)
def get_user_json(user_id):
    ...
```

While the exact way to pack or unpack a cached value depends on the underlying cache being used, Cachual provides a number of functions to unpack strings into common Python data types (since most common cache providers store all data as strings) and pack JSON values (such as dictionaries and lists that only contain basic types) so that you don't have to write these functions yourself. This allows you to rewrite the examples above:

```
from cachual import unpack_int, pack_json, unpack_json

@cache.cached(ttl=300, unpack=unpack_int)
def get_user_id(email):
```

```
...  
@cache.cached(ttl=300, pack=pack_json, unpack=unpack_json)  
def get_user_json(user_id):  
...
```

Note that for Python 3, you will want to use `unpack_json_python3` for JSON data and `unpack_bytes` for unicode string data because the value returned from the cache will be bytes (as opposed to the unicode string that is the default string type in Python 3).

For a complete list of these helper functions see *Packing and Unpacking Helpers*.

Note: Be careful with packing/unpacking. If your pack/unpack functions have unintended side effects (such as changing the encoding of the value) you may get different results when you retrieve values from the cache. Generally it is best to keep things as simple as possible - don't try to cache complex Python datatypes (such as custom objects), and keep your pack/unpack functions very simple (or use the helpers!). Make sure you understand how the underlying caching system and library deals with your data, particularly when it comes to encoding. For example, Redis will encode your value as bytes if possible, falling back to the unicode representation otherwise (which is why you need to use the pack/unpack functions as above for dictionaries).

API Documentation

6.1 Caching

class `cachual.CachualCache`

Base class for all cache implementations. Provides the `cached()` decorator which can be applied to methods whose return value you want to cache. This class should not be used directly, and is meant to be subclassed.

Subclasses should define a **get** method, which takes a single string argument, and returns the value in the cache for that key, or `None` in the case of a cache miss; and a **put** method, which takes three arguments for the cache key, the value to store, and a TLL (which may be none) and puts the value in the cache.

cached (*ttl=None, pack=None, unpack=None, use_class_for_self=False*)

Functions decorated with this will have their return values cached. It should be used as follows:

```
cache = RedisCache()

@cache.cached()
def method_to_be_cached(arg1, arg2, arg3='default'):
    ...
```

A unique cache key will be generated for each function call, so that different values for the arguments will result in different cache keys. When you decorate a function with this, the cache will be checked first; if there is a cache hit, the value in the cache will be returned. Otherwise, the function is executed and the return value is put into the cache.

Cache get/put failures are logged but ignored; if the cache goes down, the function will continue to execute as normal.

Parameters

- **ttl** (*integer*) – The time-to-live in seconds. For caches that support TTLs, the keys will expire after this time. If `None` (the default), the cache default will be used (usually no expiration).
- **pack** (*function*) – If specified, this function will be called with the decorated function's return value, and the result will be stored in the cache. This can be used to alter the value that actually gets stored in the cache in case you need to process it first (e.g. dump a JSON string that is properly escaped).
- **unpack** (*function*) – If specified, this function will be called with the value from the cache in the event of a cache hit, and the result will be returned to the caller. This can be used to alter the value that gets returned from a cache hit, in case you need to process it first (e.g. turn a JSON string into a Python dictionary).

- **use_class_for_self** (*bool*) – If True, cache keys will use the class representation of the first parameter instead of the object representation. This is useful when you want to cache instance methods, but you want the same cache key if the instance method's arguments are the same. An example would be a stateless class that is a client wrapper for an external service. The first argument would be the instance object, whose default representation contains the memory location of the object (and thus would be different for every instance, which is undesirable for a stateless class).

Changed in version 0.2.2: Added `use_class_for_self` parameter.

class `cachual.RedisCache` (*host='localhost', port=6379, db=0, **kwargs*)

A cache using [Redis](#) as the backing cache. All values will be stored as strings, meaning if you try to store non-string values in the cache, their unicode equivalent will be stored. If you want to alter this behavior e.g. to store Python dictionaries, use the `pack/unpack` arguments when you specify your `@cached` decorator.

Parameters

- **host** (*string*) – The Redis host to use for the cache.
- **port** (*integer*) – The port to use for the Redis server.
- **db** (*integer*) – The Redis database to use on the server for the cache.
- **kwargs** (*dict*) – Any additional args to pass to the [CachualCache](#) constructor.

get (*key*)

Get a value from the cache using the given key.

Parameters **key** (*string*) – The cache key to get the value for.

Returns The value for the cache key, or None in the case of cache miss.

put (*key, value, ttl=None*)

Put a value into the cache at the given key. If the value is not a string, its unicode value will be used. This behavior is defined by the underlying Redis library, and could be subject to change in future versions; thus it is safest to only store strings in Redis (although basic types should serialize into a string cleanly, you will need to unpack them when they are retrieved from the cache).

Parameters

- **key** (*string*) – The cache key to use for the value.
- **value** – The value to store in the cache.
- **ttl** (*integer*) – The time-to-live for key in seconds, after which it will expire.

class `cachual.MemcachedCache` (*host='localhost', port=11211, **kwargs*)

A cache using [Memcached](#) as the backing cache. The same caveats apply to keys and values as for Redis - you should only try to store strings (using the packing/unpacking functions). See the documentation on Keys and Values here: [pymemcache.client.base.Client](#).

Parameters

- **host** (*string*) – The Memcached host to use for the cache.
- **port** (*integer*) – The port to use for the Memcached server.
- **kwargs** (*dict*) – Any additional args to pass to the [CachualCache](#) constructor.

get (*key*)

Get a value from the cache using the given key.

Parameters **key** (*string*) – The cache key to get the value for.

Returns The value for the cache key, or None in the case of cache miss.

put (*key*, *value*, *ttl=None*)

Put a value into the cache at the given key. For constraints on keys and values, see `pymemcache.client.base.Client`.

Parameters

- **key** (*string*) – The cache key to use for the value.
- **value** – The value to store in the cache.
- **ttl** (*integer*) – The time-to-live for key in seconds, after which it will expire.

6.2 Packing and Unpacking Helpers

These functions are helpers for packing/unpacking common Python data types for a cache which stores everything as a string.

`cachual.pack_json` (*value*)

Pack the given JSON structure for storage in the cache by dumping it as a JSON string.

Parameters **value** – The JSON structure (e.g. dict, list) to pack.

Return type *string*

Returns The JSON structure as a JSON string.

`cachual.unpack_json` (*value*)

Unpack the given string by loading it as JSON.

Parameters **value** (*string*) – The string to unpack.

Returns The string as JSON.

`cachual.unpack_json_python3` (*value*)

Unpack the given Python 3 bytes by loading them as JSON.

Parameters **value** (*bytes*) – The bytes to unpack.

Returns The bytes as JSON.

`cachual.unpack_bytes` (*value*)

Unpack the given Python 3 bytes by loading them as a Python 3 unicode string, assuming UTF-8 encoding.

Parameters **value** (*bytes*) – The bytes to unpack.

Returns The bytes as a Python 3 unicode string, assuming UTF-8 encoding.

`cachual.unpack_int` (*value*)

Unpack the given string into an integer.

Parameters **value** (*string*) – The string to unpack.

Return type *integer*

Returns The string as an integer.

`cachual.unpack_long` (*value*)

Unpack the given string into a long.

Parameters **value** (*string*) – The string to unpack.

Return type *long*

Returns The string as a long.

`cachual.unpack_float` (*value*)

Unpack the given string into a float.

Parameters `value` (*string*) – The string to unpack.

Return type `float`

Returns The string as a float.

`cachual.unpack_bool` (*value*)

Unpack the given string into a boolean. ‘True’ will become True, and ‘False’ will become False (as per the unicode values of booleans); anything else will result in a ValueError.

Parameters `value` (*string*) – The string to unpack.

Return type `bool`

Returns The string as a boolean.

C

cachual, [13](#)

C

`cached()` (`cachual.CachualCache` method), 13
`cachual` (module), 13
`CachualCache` (class in `cachual`), 13

G

`get()` (`cachual.MemcachedCache` method), 14
`get()` (`cachual.RedisCache` method), 14

M

`MemcachedCache` (class in `cachual`), 14

P

`pack_json()` (in module `cachual`), 15
`put()` (`cachual.MemcachedCache` method), 14
`put()` (`cachual.RedisCache` method), 14

R

`RedisCache` (class in `cachual`), 14

U

`unpack_bool()` (in module `cachual`), 16
`unpack_bytes()` (in module `cachual`), 15
`unpack_float()` (in module `cachual`), 15
`unpack_int()` (in module `cachual`), 15
`unpack_json()` (in module `cachual`), 15
`unpack_json_python3()` (in module `cachual`), 15
`unpack_long()` (in module `cachual`), 15