BucketCache Documentation

Release 0.12.1

Frazer McLean

Jan 17, 2018

Contents

Contents:

CHAPTER 1

Installation

The recommended installation method is using `pip`.

1.1 pip

```
$ pip install bucketcache
```

1.2 Git

```
$ git clone https://github.com/RazerM/bucketcache.git
Cloning into 'bucketcache'...
```

Check out a release tag:

```
$ cd bucketcache
$ git checkout 0.12.1
```

Test and install:

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

CHAPTER 2

Usage

2.1 Overview

In one sentence, `Bucket` is a container object with optional lifetime backed by configurable serialisation methods that can also act as a function or method decorator.

Before everything is explained in detail, here's a quick look at the functionality:

```
from bucketcache import Bucket

bucket = Bucket('cache', hours=1)

bucket[any_object] = anything_serializable_by_backend  # (Pickle is the default)
```

```
class SomeService(object):
    def __init__(self, username, password):
        self.username = username
        self.password = password

    @bucket(method=True, nocache='skip_cache')
    def expensive_method(self, a, b, c, skip_cache=False):
        print('Method called.')

    @expensive_method.callback
    def expensive_method(self, callinfo):
        print('Cache used.')

some_service = SomeService()
some_service.expensive_method(1, 2, 3)
some_service.expensive_method(1, 2, 3)
some_service.expensive_method(1, 2, 3, skip_cache=True)
```

```
Method called.
Cache used.
```

```
Method called.
```

2.2 Bucket Creation

2.2.1 Path

`Bucket` has one required parameter: *path*. This must be a `str` or `Path` directory for storing the cached files.

2.2.2 Lifetime

Lifetime refers to how long a key is readable from the bucket after setting it. By default, keys do not expire.

There are two methods of setting a lifetime:

```
from datetime import timedelta

bucket = Bucket('path', lifetime=timedelta(days=1, seconds=1, microseconds=1)))
```

```
bucket = Bucket('path', days=1, seconds=1, microseconds=1)
```

The latter is just a shortcut for the former. See `datetime.timedelta` for all supported keyword arguments.

2.2.3 Backends

Buckets can use any backend conforming to abstract class *bucketcache.backends.Backend*. There are three provided backends:

- PickleBackend
- JSONBackend
- MessagePackBackend (if python-msgpack is installed)

By default, Pickle is used. Explicitly, this is specified as follows:

```
from bucketcache import PickleBackend

bucket = Bucket('path', backend=PickleBackend)
```

Each backend has an associated *bucketcache.config.BackendConfig* subclass.

For example, protocol version 4 could be used if on Python 3.4+

```
from bucketcache import PickleConfig

bucket = Bucket('path', backend=PickleBackend, config=PickleConfig(protocol=4))
```

Typically, all of the parameters that can be used by the relevant *dump* or *load* methods can be specified in a config object.

2.2.4 KeyMakers

Buckets can use any backend conforming to abstract class *bucketcache.keymakers.KeyMaker*. By default, *DefaultKeyMaker* is used, as it can convert almost any object into a key.

As *DefaultKeyMaker* converts objects to keys in memory, this can cause problems with large key objects. *StreamingDefaultKeyMaker* can be used instead, which uses a temporary file behind the scenes to reduce memory usage.

```
from bucketcache import StreamingDefaultKeyMaker

bucket = Bucket('path', keymaker=StreamingDefaultKeyMaker())
```

2.3 Decorator

2.3.1 Functions and methods

```
@bucket
def function(a, b):
    ...

class A(object):
    @bucket(method=True)
    def method(self, a, b):
        ...

result = function(1, 2)
result = function(1, 2)  # Cached result

a = A()
result = a.method(3, 4)
result = a.method(3, 4)  # Cached result
```

2.3.2 Callback

```
>>> @bucket
... def function(a, b):
...     return a + b

>>> @function.callback
... def function(callinfo):
...     print(callinfo)

>>> function(1, 2)
3
>>> function(1, 2)
CachedCallInfo(varargs=(), callargs={'a': 1, 'b': 2}, return_value=3, expiration_
↪date=datetime.datetime(2015, 1, 1, 9, 0, 0))
3
```

2.3.3 Properties

```
class A(object):
    @property
    @bucket(method=True)
    def this(self):
        ...

    @bucket(method=True)
    @property
    def that(self):
        ...
```

To use callback with properties, define the method first.

```
class A(object):
    @bucket(method=True)
    def this(self):
        ...

    @this.callback
    def this(self, callinfo):
        ...

    this = property(this)
```

2.3.4 Skip cache

```
@bucket(nocache='refresh')
def function(self, refresh=False):
    ...

function()

# Next call to function would use cached value, unless refresh==True
function(refresh=True)
```

2.3.5 Ignored parameters

```
@bucket(ignore=['c'])
def function(a, b, c):
    ...

function(1, 2, 3)
function(1, 2, 4)  # Uses cached result even though c is different
```

2.4 Deferred Writes

To prevent writing to file immediately, a `DeferredWriteBucket` can be used. Keys are only written to file when `bucket.sync()` is called.

`bucketcache.deferred_write()` is a context manager that defers writing until completion of the block.

```
from bucketcache import deferred_write

bucket = Bucket('path')

with deferred_write(bucket) as deferred:
    deferred[some_key] = some_value
    ...
```

It's also possible to create a `DeferredWriteBucket` manually:

```
from bucketcache import DeferredWriteBucket

bucket = DeferredWriteBucket('path')

bucket[some_key] = some_value
...

bucket.sync()  # Writing happens here.
```

Note that calling `unload_key()` on a `DeferredWriteBucket` forces a sync.

2.5 Logging

You can enable logging as follows:

```
from bucketcache import logger
logger.disabled = False
```

Here, *logger* is an instance of `logbook.Logger`. By default, the level is set to `logbook.NOTSET` (i.e. everything is logged).

There is a *logger_config* object, which currently only has one option:

```
from bucketcache import logger_config

logger_config.log_full_keys = True
```

log_full_keys prevents keys from being abbreviated in the log. This may be useful for debugging:

```
from bucketcache import Bucket, logger, logger_config

logger.disabled = False
logger_config.log_full_keys = True

bucket = Bucket('cache')

class A(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    @bucket(method=True)
    def spam(self, eggs):
        return eggs
```

```
a = A('this', 'that')
a.spam(5)
```

```
DEBUG: bucketcache.log: _hash_for_key <({'a': 'this', 'b': 'that'}, ('spam',␣
↪OrderedDict([('args', ['self', 'eggs']), ('varargs', None), ('varkw', None), (
↪'defaults', None), ('kwonlyargs', []), ('kwonlydefaults', None), ('annotations', {}
↪)])), (), {'eggs': 5})>
DEBUG: bucketcache.log: Done
INFO: bucketcache.log: Attempt load from file: cache/34f8a5019b61dfa8162f09339b72fde9.
↪pickle
INFO: bucketcache.log: Function call loaded from cache: <function spam at 0x10622a848>
```

Note: At level `logbook.DEBUG`, BucketCache logs tracebacks for **handled exceptions** and they will be labeled as such. These are extremely helpful to aid development of backends.

CHAPTER 3

Module Reference

3.1 bucketcache.buckets

class bucketcache.buckets.**Bucket**(*path*, *backend=None*, *config=None*, *keymaker=None*, *lifetime=None*, ***kwargs*)

Dictionary-like object backed by a file cache.

Parameters

- **backend** (*Backend*) – Backend class. Default: *PickleBackend*
- **path** – Directory for storing cached objects. Will be created if missing.
- **config** (Config) – *Config* instance for backend.
- **keymaker** (*KeyMaker*) – *KeyMaker* instance for object -> key serialization.
- **lifetime** (timedelta) – Key lifetime.
- **kwargs** – Keyword arguments to pass to datetime.timedelta as shortcut for lifetime.

setitem(*key*, *value*)

Provide setitem method as alternative to `bucket[key] = value`

getitem(*key*)

Provide getitem method as alternative to `bucket[key]`.

prune_directory()

Delete any objects that can be loaded and are expired according to the current lifetime setting.

A file will be deleted if the following conditions are met:

- The file extension matches *bucketcache.backends.Backend.file_extension()*
- The object can be loaded by the configured backend.
- The object's expiration date has passed.

Returns File size and number of files deleted.

Return type PrunedFilesInfo

Note: For any buckets that share directories, prune_directory will affect files saved with both, if they use the same backend class.

This is not destructive, because only files that have expired according to the lifetime of the original bucket are deleted.

---

unload_key(*key*)
Remove key from memory, leaving file in place.

class bucketcache.buckets.**DeferredWriteBucket**(*path*, *backend=None*, *config=None*, *keymaker=None*, *lifetime=None*, ***kwargs*)
Alternative implementation of *Bucket* that defers writing to file until *sync()* is called.

unload_key(*key*)
Remove key from memory, leaving file in place.

This forces *sync()*.

sync()
Commit deferred writes to file.

bucketcache.buckets.**deferred_write**(*bucket*)
Context manager for deferring writes of a *Bucket* within a block.

Parameters **bucket** (*Bucket*) – Bucket to defer writes for within context.

Returns Bucket to use within context.

Return type *DeferredWriteBucket*

When the context is closed, the stored objects are written to file. The in-memory cache of objects is used to update that of the original bucket.

```
bucket = Bucket(path)

with deferred_write(bucket) as deferred:
    deferred[key] = value
    ...
```

3.2 bucketcache.backends

class bucketcache.backends.**Backend**(*value*, *expiration_date=None*, *config=None*)
Bases: object

Abstract base class for backends.

Classes must implement abstract attributes:

- *binary_format*
- *default_config*
- *file_extension*

and abstract methods:

- *dump()*
- *from_file()*

binary_format
Return *True* or *False* to indicate if files should be opened in binary mode before passing to *from_file()* and *dump()*.

default_config
Associated `bucketcache.config.Config` class. Instantiated without arguments to create default configuration.

file_extension
File extension (without full stop) for files created by the backend.

classmethod check_concrete(*skip_methods=False*)
Verify that we're a concrete class.

Check that abstract methods have been overridden, and verify that abstract class attributes exist.

Although ABCMeta checks the abstract methods on instantiation, we can also do this here to ensure the given Backend is valid when the Bucket is created.

classmethod valid_config(*config*)
Verify passed config, or return *default_config*.

has_expired()
Determine if value held by this instance has expired.

Returns False if object does not expire.

classmethod from_file(*fp*, *config=None*)
Class method for instantiating from file.

Parameters

- **fp** (file-like object) – File containing stored data.
- **config** (`Config`) – Configuration passed from *Bucket*

Note: Implementations should ensure *config* is valid as follows:

```
config = cls.valid_config(config)
```

Warning: If the appropriate data cannot be ready from *fp*, this method should raise *BackendLoadError* to inform *Bucket* that this cached file cannot be used.

dump(*fp*)
Called to save state to file.

Parameters **fp** (file-like object) – File to contain stored data.

Attributes *value* and *expiration_date* should be saved, so that *from_file()* can use them.

class bucketcache.backends.**PickleBackend**(*value*, *expiration_date=None*, *config=None*)
Bases: *bucketcache.backends.Backend*

Backend that serializes objects using Pickle.

default_config
alias of `PickleConfig`

class bucketcache.backends.**JSONBackend**(*value*, *expiration_date=None*, *config=None*)
Bases: *bucketcache.backends.Backend*

Backend that stores objects using JSON.

default_config
alias of JSONConfig

class bucketcache.backends.**MessagePackBackend**(**args*, ***kwargs*)
Bases: *bucketcache.backends.Backend*

Backend that stores objects using MessagePack.

default_config
alias of MessagePackConfig

3.3 bucketcache.config

class bucketcache.config.**BackendConfig**
Bases: object

Base class for *Backend* configuration classes.

class bucketcache.config.**PickleConfig**(*protocol=2*, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*)
Bases: *bucketcache.config.BackendConfig*

Configuration class for *PickleBackend*

Parameters reflect those given to pickle.dump() and pickle.load().

Note: On Python 2, the following parameters are ignored:

- *fix_imports*
- *encoding*
- *errors*

---

class bucketcache.config.**JSONConfig**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *dump_cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, *load_cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*)
Bases: *bucketcache.config.BackendConfig*

Configuration class for *JSONBackend*

Parameters reflect those given to json.dump() and json.load().

Note: json.dump() and json.load() both have parameters called *cls*, so they are named *dump_cls* and *load_cls*.

---

class bucketcache.config.**MessagePackConfig**(*default=None*, *pack_encoding='utf-8'*, *unicode_errors='strict'*, *use_single_float=False*, *autoreset=1*, *use_bin_type=1*, *object_hook=None*, *list_hook=None*, *use_list=1*, *unpack_encoding='utf-8'*, *object_pairs_hook=None*)

Bases: *bucketcache.config.BackendConfig*

Configuration class for *MessagePackBackend*

Parameters reflect those given to `msgpack.pack()` and `msgpack.unpack()`.

Note: `msgpack.pack()` and `msgpack.unpack()` both have parameters called *encoding*, so they are named *pack_encoding* and *unpack_encoding*.

3.4 bucketcache.exceptions

These exceptions do not bubble up from user-facing methods. They are raised by private methods only.

exception bucketcache.exceptions.**KeyInvalidError**

Bases: `Exception`

Raised internally when a key cannot be loaded.

exception bucketcache.exceptions.**KeyFileNotFoundError**

Bases: *bucketcache.exceptions.KeyInvalidError*

Raised when file for key doesn't exist.

exception bucketcache.exceptions.**KeyExpirationError**

Bases: *bucketcache.exceptions.KeyInvalidError*

Raised when key has expired.

exception bucketcache.exceptions.**BackendLoadError**

Bases: `Exception`

Raised when *bucketcache.backends.Backend.from_file()* cannot load an object.

3.5 bucketcache.keymakers

class bucketcache.keymakers.**KeyMaker**

Bases: `object`

KeyMaker abstract base class.

make_key(*obj*)

Make key from passed object.

Parameters **obj** – Any Python object.

Yields bytes of key to represent object.

class bucketcache.keymakers.**StreamingDefaultKeyMaker**(*sort_keys=True*)

Bases: *bucketcache.keymakers.DefaultKeyMaker*

Subclass of DefaultKeyMaker that uses a temporary file to save memory.

class bucketcache.keymakers.**_AnyObjectJSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.encoder.JSONEncoder`

Serialize objects that can't normally be serialized by json.

Attempts to get state will be done in this order:

- `o.__getstate__()`
- Parameters from `o.__slots__`
- `o.__dict__`
- `repr(o)`

class bucketcache.keymakers.**DefaultKeyMaker**(*sort_keys=True*)

Bases: *bucketcache.keymakers.KeyMaker*

Default KeyMaker that is consistent across Python versions.

Uses *_AnyObjectJSONEncoder* to convert any object into a string representation.

Parameters **sort_keys** (*bool*) – Sort dictionary keys for consistency across Python versions with different hash algorithms.

class bucketcache.keymakers.**StreamingDefaultKeyMaker**(*sort_keys=True*)

Bases: *bucketcache.keymakers.DefaultKeyMaker*

Subclass of DefaultKeyMaker that uses a temporary file to save memory.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

Symbols

B

C

D

F

G

H

J

K

M

P

S

U

V