fcache

Release 0.4.7

Mar 12, 2017

Contents

fcache is a dictionary-like, file-based cache module for Python. This is fcache's documentation. It covers installation, a tutorial, and the API reference.

CHAPTER 1

Installation

Installing fcache is easy. fcache supports Python 2.7 and 3. fcache also requires the appdirs package.

Pip

To keep things simple, install fcache using pip:

```
$ pip install fcache
```

This will download fcache from the Python Package Index and install it in your Python's `site-packages` directory.

Tarball Release

If you'd rather install fcache manually:

1. Download the most recent release from fcache's PyPi page.
2. Unpack the tarball.
3. From inside the directory `fcache-XX`, run `python setup.py install`

This will install fcache in your Python's `site-packages` directory.

Install the Development Version

fcache's code is hosted at GitHub. To install the development version first make sure Git is installed. Then run:

```
$ git clone https://github.com/tsroten/fcache.git
$ pip install -e fcache
```

This will link the `fcache` directory into your `site-packages` directory.

Running the Tests

Running the tests is easy:

```
$ python setup.py test
```

If you want to run the tests using different versions of Python, install and run tox:

```
$ pip install tox
$ tox
```

CHAPTER 2

Getting Started

Let's create a file cache:

```
>>> from fcache.cache import FileCache
>>> mycache = FileCache('appname')
```

You can use the cache just like a `dict`:

```
>>> mycache['foo'] = 'value'
>>> mycache['foo']
'value'
>>> mycache['bar'] = 'blah'
>>> list(mycache)
['foo', 'bar']
>>> del mycache['bar']
```

By default, the cache object stores changes in a buffer that needs to be explicitly synced with the cache's files:

```
>>> mycache.sync()
>>> # closing a cache also syncs it:
... mycache.close()
```

If you want the cache to automatically write changes to disk everytime you add/modify values, open the cache and append `'s'` to the optional *flag* argument:

```
>>> mycache = FileCache('appname', flag='cs')
```

The `'c'` means that the object will open a cache if it exists, but will create a new one if no cache is found. The `'s'` means that the cache is opened in sync mode. All changes are immediately written to disk.

CHAPTER 3

Using fcache with a `Shelf`

Python's `Shelf` class provides a persistent dictionary-like object. Shelves normally use the `dbm` module as a backend. You can use easily use fcache as a shelf's backend if needed. Because shelves already serialize data, you'll need to tell fcache not to serialize the data:

```
>>> mycache = FileCache('appname', serialize=False)
>>> myshelf = Shelf(mycache)
```

That's it! You can use the `Shelf` just like you normally would.

CHAPTER 4

API

class fcache.cache.**FileCache**(*appname*, *flag='c'*, *mode=438*, *keyencoding='utf-8'*, *serialize=True*, *app_cache_dir=None*)

A persistent file cache that is dictionary-like and has a write buffer.

appname is passed to appdirs to determine a system-appropriate location for the cache files. The cache directory used is available via *cache_dir*.

By default, a write buffer is used, so writing to cache files is not done until *sync()* is explicitly called. This behavior can be changed using the optional *flag* argument.

Note: Keys and values are always stored as `bytes` objects. If data serialization is enabled, keys are returned as `str` or `unicode` objects. If data serialization is disabled, keys are returned as a `bytes` object.

Parameters

- **appname** (*str*) – The app/script the cache should be associated with.
- **flag** (*str*) – How the cache should be opened. See below for details.
- **mode** – The Unix mode for the cache files.
- **keyencoding** (*str*) – The encoding the keys use, defaults to 'utf-8'. This is used if *serialize* is `False`; the keys are treated as `bytes` objects.
- **serialize** (*bool*) – Whether or not to (de)serialize the values. If a cache is used with a `Shelf`, set this to `False`.
- **app_cache_dir** (*str*) – absolute path to root cache directory to be used in place of system-appropriate location determined by appdirs

The optional *flag* argument can be:

Value	Meaning
`'r'`	Open existing cache for reading only
`'w'`	Open existing cache for reading and writing
`'c'`	Open cache for reading and writing, creating it if it doesn't exist (default)
`'n'`	Always create a new, empty cache, open for reading and writing

If a `'s'` is appended to the *flag* argument, the cache will be opened in sync mode. Writing to the cache will happen immediately and will not be buffered.

If an application needs to use more than one cache, then it should use subcaches. To create a subcache, append a series of one or more names separated by periods to the application name when creating a *FileCache* object (e.g. `'appname.subcache'` or `'appname.subcache.subcache'`). Subcaches are a way for an application to use more than one cache without polluting a user's cache directory. All caches – main caches or subcaches – are totally independent. The only aspect in which they are linked is that all of an application's caches exist in the same system directory. Because each cache is independent of every other cache, calling *delete()* on an application's main cache will not delete data in its subcaches.

cache_dir
: The absolute path to the directory where the cache files are stored. The *appname* passed to *FileCache* is used to determine a system-appropriate place to store the cache files.

close()
: Sync the write buffer, then close the cache.

  If a closed *FileCache* object's methods are called, a `ValueError` will be raised.

create()
: Create the write buffer and cache directory.

delete()
: Delete the write buffer and cache directory.

sync()
: Sync the write buffer with the cache files and clear the buffer.

  If the *FileCache* object was opened with the optional `'s'` *flag* argument, then calling *sync()* will do nothing.

In addition to the four methods listed above, *FileCache* objects also support the following standard `dict` operations and methods:

len(f)
: Return the number of items in the *FileCache* *f*.

f[key]
: Return the item of *f* with key *key*. Raises a `KeyError` if *key* is not in the cache.

f[key] = value
: Set `f[key]` to *value*.

del f[key]
: Remove `f[key]` from *f*. Raises a `KeyError` if *key* is not in the cache.

key in f
: Return `True` if *f* has a key *key*, else `False`.

key not in f
: Equivalent to `not key in f`.

iter(f)
: Return an iterator over the keys of the cache. This is a shortcut for `iter(f.keys())` (`iterkeys()` in Python 2).

clear()
Remove all items from the write buffer and cache.

The write buffer object and cache directory are not deleted.

get(*key*[, *default*])
Return the value for *key* if *key* is in the cache, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

items()
Return a new view of the cache's items (`(key, value)` pairs). See the documentation of view objects.

In Python 2, this method returns a copy of the cache's list of `(key, value)` pairs.

keys()
Return a new view of the cache's keys. See the documentation of view objects.

In Python 2, this method returns a copy of the cache's list of keys.

pop(*key*[, *default*])
If *key* is in the cache, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the cache, a `KeyError` is raised.

popitem()
Remove and return an arbitrary `(key, value)` pair from the cache.

setdefault(*key*[, *default*])
If *key* is in the cache, return its value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to `None`.

update([*other*])
Update the cache with the key/value pairs from *other*, overwriting existing keys. Return `None`.

values()
Return a new view of the cache's values. See the documentation of view objects.

In Python 2, this method returns a copy of the cache's list of values.

CHAPTER 5

Change Log

v.0.4.7 (2017-03-11)

- Minor code changes/updates.

v.0.4.6 (2017-01-30)

- Allow app_cache_dir to be specified by user

v.0.4.5 (2015-10-21)

- Uses shutil.move() instead of os.rename(). Fixes #22. Thanks Philip!
- Adds pypi and travis-ci badges to README.
- Adds flake8 to travis-ci and tox.
- Adds Python 3.5 tests.
- Includes tests in release package.

v.0.4.4 (2014-03-19)

- Adds support for subcaches. Resolves #20.

v.0.4.3 (2014-03-13)

- Creates AUTHORS.txt file.

- Adds test for FileCache.__iter__() and FileCache.__contains__().
- Fixes FileCache._all_keys assuming _buffer attribute (#19). Thanks soult!

v.0.4.2 (2014-03-01)

- Adds unicode key support. Fixes #18.
- Adds docs test environment to tox. Fixes #17.
- Fixes code example typo. Fixes #16.
- Fixes typo in docstrings about serialization. Fixes #15.
- Adds not about appdirs requirement. Fixes #14.

v.0.4.1 (2014-01-03)

- Adds appdirs support (issue #13)

v.0.4 (2014-01-02)

- backwards-incompatible rewrite; fcache now emulates a `dict`.

v.0.3.1 (2013-04-19)

- bug fix: close temp file after creation (issue #1)

v.0.3 (2013-01-03)

- now supports Python 2.6, 2.7, and 3.
- added `set_default()` method.
- `invalidate()` can now be called with no arguments, in which case it forces all data to expire.
- added `keys()` method.
- added `values()` method.
- added `items()` method.

v.0.2.1 (2012-12-31)

- removed code-blocks from README so that PyPI would render the readme correctly.

v0.2 (2012-12-31)

- added `invalidate()` method.
- added documentation.
- added *override* switch to the `get()` method.

v0.1 (2012-12-30)

- Initial release.

Python Module Index

f

Index