lck.common Documentation

Release 0.4.5

Łukasz Langa, LangaCore

Sep 27, 2017

Contents

This library consists of various simple common routines and language constructs that are so useful they ten to be rewritten in every subsequent project I'm working on. Each function, decorator or module on its own is too simple to dedicate an entire PyPI package for it. Together however, this library represents a Swiss army knife for everyday needs (YMMV). Among the things you might find inside:

- robust memoization
- some less obvious collections (e.g. `orderedset`)
- a `@synchronized` decorator (with threading or lockfile backends)
- some controversial language enhancements like the Null object
- converter from ElementTree instances to dicts
- file finder (searching locations commonly used for storing app data)

The latest version can be installed via PyPI:

```
$ pip install lck.common
```

or:

```
$ easy_install lck.common
```

The source code repository and issue tracker are maintained on GitHub.

For the curious, `lck` stands for LangaCore Kit. LangaCore is a one man software development shop of mine.

Note: `lck.common` requires **Python 2.7** because all of its code is using the so-called four futures (`absolute_imports`, `division`, `print_function` and `unicode_literals`). One of the virtues in the creation of this library is to make the code beautiful. These switches give a useful transitional state between the old Python 2.x and the new Python 3.x. You should use them as well.

CHAPTER 1

Change Log

0.4.5

- fixed an uncommon bug in memoization where an exception in the memoized function could leave stale keys in the cache

0.4.4

- `lck.git` introduced with a `get_version` routine
- `decode_entities` added to `lck.xml`

0.4.3

- `lck.lang.Null` introduced, see Null Object pattern
- `lck.lang.unset` is now a `Null` instance
- `lck.xml` introduced with a `etree_to_dict` routine
- `lck.config` has been removed, use the configparser backport

0.4.2

- `lck.crypto` introduced with a couple of thin wrappers over PyCrypto
- `lck.math` introduced starting with Elo rating calculation routine.

0.4.1

- `lck.lang.unset` is now also `False` and `len(unset)` is zero

0.4.0

- migrated to the `lck` namespace from `langacore.kit`
- migrated licensing from GPL 3 to MIT
- bumped the trove from alpha status to beta, the code is in production for over a year now

Ancient history

- No proper change log was kept before 0.4.0

CHAPTER 2

This documentation

Overview

For now the library is still quite small. Functionality gets added or refined as needed.

Decorator modules

@synchronized

This decorator mimics the behaviour of the Java keyword, enabling users to treat whole functions or methods as atomic. The most simple use case involves just decorating a function:

```
from lck.concurrency import synchronized

@synchronized
def func():
  pass
```

After decoration, all calls to the function are synchronized with a reentrant threading lock so that no matter how many threads invoke the function at the same time, all calls are serialized and in effect are run one after another. The default lock is reentrant so it's okay for a synchronized function to be recursive.

In case where a whole group of functions should be serialized, the user can explicitly provide a lock object to the decorator:

```
from threading import Lock
from lck.concurrency import synchronized

LOCK=Lock()

@synchronized(lock=LOCK)
def func1():
  pass
```

```
@synchronized(lock=LOCK)
def func2():
  pass
```

Sharing a lock means that at any time at most one of the functions in the group is called, no matter how many threads are running. It's also worth noting that excplicitly providing a lock enables the user to choose another lock implementation. In the above example a simple non-reentrant lock is used, in effect the performance is higher than in the reentrant case, **but the functions sharing the same lock cannot call themselves**.

If the application is run in a multiprocess environment, locks based on threading are not the answer. In that case the decorator can be fed with a file path instead of a lock object:

```
from lck.concurrency import synchronized

@synchronized(path='/tmp/example.lock')
def func():
  pass
```

In that case upon every function call a lock file will be created on the given path to ensure serial execution across multiple processes. The implementation uses Skip Montanaro's excellent lockfile library. It is using atomic operations available on a given platform to ensure correctness. In case of POSIX systems, hard links are created. On Windows, directories are made.

@memoize

This decorator enhances performance by storing the outcome of the decorated function given a specific set of arguments. Across the application the function is called as it normally would but in fact, only the first call with a concrete set of arguments is calculated. All subsequent calls with the same arguments return the stored value calculated at first.

This is particularly a win for resource or time consuming functions that are called multiple times with the same arguments.

The most typical use case for this decorator will be simply:

```
from time import sleep
from lck.cache import memoize

@memoize
def expensive_func(arg):
  sleep(10)
  print arg

expensive_func('Hello') # 10 seconds before we see 'Hello'
expensive_func('Hello') # now 'Hello' appears instantly
expensive_func('World') # 10 seconds before we see 'World'
expensive_func('World') # now 'World' appears instantly
```

The decorator is configurable so that the user can specify how long the outcome should be cached, or how many different sets of arguments should be stored in the cache:

```
from lck.cache import memoize

@memoize(update_interval=15)
def recalculation_every_15_seconds():
  pass
```

```
@memoize(max_size=2)
def only_two_last_used_args_will_be_cached(arg):
  pass
```

Details

For more detailed view on the decorators, see the documentation below.

cache.memoization	lck.cache.memoization
concurrency.synchronization	lck.concurrency.synchronization

lck.cache.memoization

lck.cache.memoization

Implements a reusable memoization decorator. It is using a finite-size cache with pickled arguments as keys, to hold the outcome of a specific function call. When the decorated function is called again with the same arguments, the outcome is fetched from the cache instead of being recalculated again.

The cache used maintains a list of *Least Recently Used* keys so that in case of overflow only the seemingly least important ones get deleted.

Note: Instead of importing the whole structure, a recommended shortcut is available. Use `from lck.cache import memoize`.

Functions

memoize(*func=None*, *update_interval=300*, *max_size=256*, *skip_first=False*, *fast_updates=True*)

Memoization decorator.

Parameters

- **update_interval** – time in seconds after which the actual function will be called again
- **max_size** – maximum buffer count for distinct memoize hashes for the function. Can be set to 0 or `None`. Be aware of the possibly inordinate memory usage in that case
- **skip_first** – `False` by default; if `True`, the first argument to the actual function won't be added to the memoize hash
- **fast_updates** – if `True` (the default), an optimized LRU algorithm is used where all function invocations except every Nth (where N == sys.maxint) are much faster but cache overflow is costly. In general, having `fast_updates` set to `True` gives a 15% performance boost when there are no cache misses (the possible number of used argument combinations for the decorated function is smaller than the value of `max_size`). If cache misses exceed 50%, you might want to increase `max_size`. If that's not feasible, memoization with `fast_updates` set to `False` will perform faster.

lck.concurrency.synchronization

lck.concurrency.synchronization

Implements a reusable Java-like synchronization decorator. It is using threading locks or filesystem-based locks to synchronize subsequent calls of the specified functions. The former kind of lock is reentrant, the latter is not.

For filesystem-based locks the module is using Skip Montanaro's lockfile library, compatible with Windows and POSIX environments.

Note: Instead of importing the whole structure, a recommended shortcut is available. Use `from lck.concurrency import synchronized`.

Functions

synchronized(*func=None*, *lock=None*, *path=None*)
Synchronization decorator.

Parameters

- **lock** – the user can specify a concrete lock object to be used with this specific synchronization decorator. This is useful when a group of functions should be synchronized together.
- **path** – instead of using threading-based locking, file-based locks may be used instead. Beware, these are radically less performant than threading locks.

lck.crypto

lck.crypto

High-level cryptographic routines.

Factory functions

These are convenience routines that create `Cipher` instances with the correct algorithm implementation plugged in.

aes([*key*, *path*, *create*]) → Cipher instance
Factory creating a cipher using the AES algorithm. Arguments have the same meaning as in the raw Cipher class.

blowfish([*key*, *path*, *create*]) → Cipher instance
Factory creating a cipher using the Blowfish algorithm. Arguments have the same meaning as in the raw Cipher class.

cast([*key*, *path*, *create*]) → Cipher instance
Factory creating a cipher using the CAST algorithm. Arguments have the same meaning as in the raw Cipher class.

des([*key*, *path*, *create*]) → Cipher instance
Factory creating a cipher using the DES algorithm. Arguments have the same meaning as in the raw Cipher class.

des3([*key*, *path*, *create*]) → Cipher instance
Factory creating a cipher using the DES3 algorithm. Arguments have the same meaning as in the raw Cipher class.

Classes

lck.files

lck.files

Filesystem based utilities.

Functions

finder(*explicit_path*, *envvar=None*, *multiple_allowed=False*)

Finds a specific file using explicitly given path (or given by an environment variable). The algorithm is as follows: for every given path from the args (explicitly given, environment variable, fallback) check whether the file exists. If it doesn't and the path is not absolute, search the working directory, its parent directory and all child directories, the current user's home directory and /etc.

Parameters

- **explicit_path** – path explicitly given by the user, can be a single entry or a sequence
- **envvar** – name of the environment variable where to look for the path
- **fallback** – name of the file to check if everything else fails
- **multiple_allowed** – False by default. If True, the returned type is a tuple with potentially many entries.

Returns the real absolute path to the file. Raises IOError if no found.

..note:: Works only on POSIX systems.

lck.git

lck.git

Helpers for git repositories.

Functions

get_version(*module*) → u'git-shortSHA1 (date & time of last commit)'

Returns a short, nicely formatted tag that can be used for versioning purposes on websites or command-line tools. The version given is based on the last commit on the repository the specified *module* object is a part of.

lck.lang

lck.lang

Holds various constructs which extend or alter behaviour of the core language.

Null

unset

class NullDict

class `NullList`

`nullify`(*obj*)

lck.math

lck.math

Various math related utilities.

lck.math.elo_rating

`rate`(*[winner_rank, loser_rank, penalize_loser]) -> (new_winner_rank, new_loser_rank*)
Computes the new ratings after a game. *winner_rank* and *loser_rank* must be integers, default is 1000. If *penalize_loser* is `True` (the default), points added to the winner are subtracted from the loser.

lck.xml

lck.xml

Various XML-related utilities.

`decode_entities`(*string*[, *encoding*]) → string_with_decoded_entities
Decodes XML entities from the given string. Supports both Unicode and bytestring arguments.

Note: when using a bytestring *string* argument, a bytestring will be returned. In that case however, *encoding* has to be specified, otherwise an UnicodeDecodeError will be raised. This is because we have to support the &#xxxx; entity which enables people to use any Unicode codepoint.

`etree_to_dict`(*element, [namespace]) -> ("tag_name", dict_with_children*)
element must be a valid ElementTree element. *namespace* is optional, must be given in Clark notation, e.g. "{ns_uri}".

License

TODO

Things that would be great to have but I haven't gotten to do them yet.

Code

- Migrating to `configparser` and throwing out FunkyConfigParser would do much good
- There are not enough unit tests
- No examples in the code

Docs

- Bits documented only by means of API, no proper introduction:
  - forms
  - models
- Bits undocumented:
  - orderedset
  - score
  - tags
- There is no clear roadmap of where this project is heading
- No FAQ, Tutorial

Community

- There is no community
- Some publicity would be helpful

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index