Lazydog Documentation

Release 0.1.1

Clément Warneys

May 03, 2018

Contents

CHAPTER 1

Lazydog

Python module monitoring user-level file system events like Creation, Modification, Move, Copy, and Deletion of files and folders. Lazydog tries to aggregate low-level events between them in order to emit a minimum number of high-level events (actually one event per user action). Lazydog uses python Watchdog API to detect low-level events.

1.1 Getting Started

1.1.1 How to install it

The easiest way:

```
$ pip3 install lazydog
```

1.1.2 How to use it

Where the watchdog module would throw dozen of events after each user event, lazydog only throws one. For example, ask lazidog to watch any existing directory:

```
$ lazydog /the/directory/you/want/to/watch
```

And just move a file in the watched directory (here from `/watched/directory/move_test.txt` to `/watched/directory/move_test_2.txt`), and wait 2 seconds. You will get something like this in the console:

```
INFO -
INFO - LIST OF THE LAST EVENTS:
INFO - moved: '/move_test.txt' to '/move_test_2.txt' mtime[1512151173.0] size[5]
INFO -
```

Try to copy the same file, and you will get somthiing like this:

```
INFO -
INFO - LIST OF THE LAST EVENTS:
INFO - copied: '/move_test_2.txt' to '/move_test_2 - Copie.txt' mtime[1512151173.0]␣
↪size[5]
INFO -
```

Only one event per user action. You can try it with other type of action (Deletion, Creation, Modification), and also with directories.

1.1.3 How to use in in third-part apps

Below is an example on how to rapidly initialize the high-level lazydog event handler, and log every new event in the console (using logging module). The watched directory is the current one (using `os.getcwd()`).

Please note that once installed, using the `$ lazydog` command in the console does just the same.

```
import logging
import os

from lazydog.handlers import HighlevelEventHandler

# LOG
# create logger
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
# create console handler with a higher log level
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.DEBUG)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
console_handler.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(console_handler)

# INITIALIZE
# get dir in parameter else current dir
watched_dir = directory if len(directory) > 1 else os.getcwd()
# initializing a new HighlevelEventHandler
highlevel_handler = HighlevelEventHandler.get_instance(watched_dir)
# starting it (since it is a thread)
highlevel_handler.start()
# log first message
logging.info('LISTENING EVENTS IN DIR: \'%s\'' % watched_dir)

# OPERATING
try:
    while True:

        # The following loop check every 1 second if any new event.
        time.sleep(1)
        local_events = highlevel_handler.get_available_events()

        # If any, it logs it directly in the console.
        for e in local_events:
            logging.info(e)

    # Keyboard <CTRL+C> interrupts the loop
```

```
    except KeyboardInterrupt:
        highlevel_handler.stop()
```

1.1.4 Getting further

Please find full code documentation in an HTML format on ReadTheDocs.org: http://lazydog.readthedocs.io/

1.1.5 Miscellaneous...

Watchdog uses inotify by default on Linux to monitor directories for changes. It's not uncommon to encounter a system limit on the number of files you can monitor (for example 8192 directories). You can get your current inotify file watch limit by executing:

```
$ cat /proc/sys/fs/inotify/max_user_watches
8192
```

When this limit is not enough to monitor all files inside a directory, the limit must be increased for Lazydog to work properly. You can set a new limit temporary with:

```
$ sudo sysctl fs.inotify.max_user_watches=524288
$ sudo sysctl -p
```

If you like to make your limit permanent, use:

```
$ echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf
$ sudo sysctl -p
```

1.2 Get it installed, the contributor way

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes.

1.2.1 Prerequisites

Main dependency of lazydog, is the python watchdog API. You can install it using the following command:

```
$ pip3 install watchdog
```

Please read the official documentation for any question about this project: https://pypi.org/project/watchdog/

1.2.2 Installing development environment

Just clone the repository in your local working directory (or fork it).

```
$ git clone https://github.com/warniiiz/Lazydog
```

In order to contribute, you will need pytest for testing purpose (or refer to the pytest documentation).

```
$ pip3 install pytest
```

You will also need Sphinx package for documentation purpose (or refer to the Sphinx documentation).

```
$ apt-get install python-sphinx
```

1.3 Tests

1.3.1 Module testing

The different python module are in the `/lazydog` directory. Each of them has attached test functions, that are in the `/lazydog/test` directory. You can launch tests unitary like this (for example for testing the events module):

```
$ pytest lazydog/test/test_events.py
```

Kind of results:

```
================================== test session starts␣
↪==================================
platform linux -- Python 3.4.2, pytest-3.5.0, py-1.5.3, pluggy-0.6.0
rootdir: /media/maxtor/media/Python/Lazydog, inifile:
plugins: cov-2.5.1
collected 16 items

lazydog/test/test_events.py ................                                [100%]

============================== 16 passed in 0.51 seconds␣
↪==============================
```

You can also test the whole package (assuming you are in the developpement directory):

```
$ pytest
```

1.3.2 Test coverage

Check the test coverage:

```
$ py.test --cov lazydog
```

Test coverage is > 90%. The metric is not very relevant about the test quality, but at least you will be reasssured there are some tests ;)

```
========================== test session starts ==========================
platform linux -- Python 3.4.2, pytest-3.5.0, py-1.5.3, pluggy-0.6.0
rootdir: /media/maxtor/media/Python/Lazydog, inifile:
plugins: cov-2.5.1
collected 58 items

lazydog/test/test_events.py ................                          [ 27%]
lazydog/test/test_handlers.py ......................                  [ 65%]
lazydog/test/test_queues.py ..                                        [ 68%]
lazydog/test/test_states.py ..................                        [100%]
```

```
----------- coverage: platform linux, python 3.4.2-final-0 -----------
Name                                                             Stmts   Miss  Cover
------------------------------------------------------------------------------
lazydog/__init__.py                                                  0      0   100%
lazydog/dropbox_content_hasher.py                                   66     14    79%
lazydog/events.py                                                  249      7    97%
lazydog/handlers.py                                                214     29    86%
lazydog/lazydog.py                                                  39     39     0%
lazydog/queues.py                                                   18      0   100%
lazydog/revised_watchdog/__init__.py                                 0      0   100%
lazydog/revised_watchdog/events.py                                  31      1    97%
lazydog/revised_watchdog/observers/__init__.py                       0      0   100%
lazydog/revised_watchdog/observers/inotify.py                       49      6    88%
lazydog/revised_watchdog/observers/inotify_buffer.py                12      0   100%
lazydog/revised_watchdog/observers/inotify_c.py                     72     22    69%
lazydog/states.py                                                  109      1    99%
lazydog/test/test_events.py                                        261      2    99%
lazydog/test/test_handlers.py                                      355      3    99%
lazydog/test/test_queues.py                                         31      0   100%
lazydog/test/test_states.py                                        172      0   100%
------------------------------------------------------------------------------
TOTAL                                                             1678    124    93%

====================== 58 passed in 30.15 seconds ========================
```

1.4 Documentation

1.4.1 Full code documentation

Please find full code documentation in an HTML format on ReadTheDocs.org: http://lazydog.readthedocs.io/

This documentation is automatically updated each time an update is made no GitHub.

1.4.2 Maintaining documentation up-to-date

Please document each change. If you want to check the result before publishing, you can run the following after each documentation modification:

```
$ cd docs     # first go in the /docs subdirectory.
$ make html   # recompute the sphinx documentation
```

The resulted documentation is then in the local relative folder `/docs/_build/html/index.html`.

Note that if you did not modify local file from `/docs` subdirectory, the changes will not be taken... you can use the following command to force recomputing all the changes:

```
$ touch autodoc.rst; make html
```

Last thing. If you modified the main `README.md`, and you want the changes to appear in the documentation (and not only on github), you have to convert the .md file to a .rst one. You can use the pandoc app to do thiss conversion, using the following command (after installing Pandoc, please refer to Pandoc documentation for more information):

```
pandoc --from=markdown --to=rst --output=README.rst ../README.md      # Assuming you␣
↪are in the /docs subdirectory.
```

Then don't forget to run the previous command again to recompute the whole documentation.

1.5 Contributing

For **lazydog** to be a truly great project, third-party code contributions are important. If you want to enhance lazydog, spot bugs or fix them, or just ask for new enhancements, you are so much welcome! Below is a list of things that might help you in contributing to lazydog.

1.5.1 Check the current issues

The list of the current bugs, issues, new enhancement proposals, etc. are all grouped on GitHub Issues' tab:

- Issue tracker

For more information about GitHub, please check the followings:

- General GitHub documentation
- GitHub pull request documentation

1.5.2 Getting Started

To get involved in code enhancement:

- Make sure you have a GitHub account
- Get the latest version, by either way cloning of forking this repository (depending on what you want to do)
- Install the requirements via pip: `pip install -r requirements.txt`
- Submit an issue directly on GitHub:
- For bugs, clearly describe the issue including steps to reproduce
- For enhancement proposals, be sure to indicate if you're willing to work on implementing the enhancement

If you do not have GitHub account and you just want to notify for a new bug, please report me by e-mail.

1.5.3 Making Changes

- `lazydog` does not use any git Workflow until now. This will remains until the volume of changes and contribution needs a clearer workflow.
- Make commits of logical units.
- Check for unnecessary whitespace with `git diff --check` before committing.
- Make sure you have added the necessary tests for your changes.
- Run `python setup.py test` to make sure your tests pass
- Run `coverage run --source=lazydog setup.py test` if you have the `coverage` package installed to generate coverage data
- Check your coverage by running `coverage report`
- Please correctly document the code you wrote, and ensure it is readable once HTML generated
- Update main documentation files (README.md, etc.) when necessary.

1.5.4 Submitting Changes

- Push your changes to the feature branch in your fork of the repository.
- Submit a pull request to the main repository

1.6 Versioning and release notes

We use SemVer for versioning. Please read RELEASE-NOTES.md for details about each releases.

1.7 Authors and contributors

- **Clément Warneys** - *Initial work* - warniiiz

1.8 License

This project is licensed under the Apache License Version 2.0. Please see the LICENSE.md file for details.

1.9 Special thanks

Thanks to Jeff Knupp for this general guidelines for open sourcing a python project (which helped me a lot since it is my first open source project I deliver):

CHAPTER 2

Lazydog code documentation

2.1 Lazydog

package lazydog

synopsis File system user-level events monitoring.

author Clément Warneys <clement.warneys@gmail.com>

This is the main package of the **lazydog** library. It relies on another sub-package **revised_watchdog** which is a modified version of **watchdog** package.

As a summary, **watchdog** is a "*python API and shell utilities to monitor file system events*". As such, **watchdog** is monitoring and emitting every tiny local event on the file system, which often means 5 or more watchdog events per user event. For example, when a user is creating a new file, you will get 1 creation event, multiple modification events (some of them for the content modification, others for metadata mosification), and 1 or more modification event for the directory of the file.

The goal of **lazydog** is to emit only 1 event per user event. This kind of event will sometimes be call **high-level event**, compared to **low-level event** which are emitted by the **watchdog** API. To do so, **lazydog** is waiting a little amount of time in order to correlate different **watchdog** events between them, and to aggregate them when related. This mechanism results in some delays between the user action and the event emission. The total delay depends on the **watchdog** observer class. For example, if you use an `InotifyObserver` observer, you only need a 2-seconds delay. But if you use a more basic observer as the `PollingObserver` observer (which is more compatible between different system), then you need a greater delay such as 10-seconds.

The **lazydog** package contains the following modules:

- *lazydog* is a sample module that show how to use the package, by logging the high-level events in the console. The main function of this module is called when calling `$ lazidog` in the console.
- *handlers* is the main module of the library with the aggregation algorithms.
- *events* defines the high-level lazydog events, based on the low-level watchdog ones, which are now aggregable and also convertible to copy or move events.
- *queues* bufferizes lazydog events pending for a possible aggregation with other simultaneous events.

- *states* keeps track of the current state of the watched local directory. The idea is to save computational time, avoiding recomputing file hashes or getting size and time of each watched files (depending on the requested method), thus facilitating identification of copy events.
- *dropbox_content_hasher* is the default hash function to get a hash of a file. Based on the hash function of the Dropbox API.

2.1.1 lazydog.lazydog

module lazydog.lazidog

synopsis An sample module that show how to use the package, by logging the high-level lazydog events in the console. The main function of this module is executed by calling `$ lazidog` in the system console.

author Clément Warneys <clement.warneys@gmail.com>

Please read the source code for more information. Below is an example on how to initialize the high-level lazydog event handler, and log every new event in the console (using logging module). The watched directory is the current one (using `os.getcwd()`).

```
import logging
import os

from lazydog.handlers import HighlevelEventHandler

# LOG
# create logger
logger = logging.getLogger()
logger.setLevel(logging.INFO)
# create console handler with a higher log level
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.INFO)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
console_handler.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(console_handler)

# INITIALIZE
# get dir in parameter else current dir
watched_dir = directory if len(directory) > 1 else os.getcwd()
# initializing a new HighlevelEventHandler
highlevel_handler = HighlevelEventHandler.get_instance(watched_dir)
# starting it (since it is a thread)
highlevel_handler.start()
# log first message
logging.info('LISTENING EVENTS IN DIR: '%s'' % watched_dir)

# OPERATING
try:
    while True:

        # The following loop check every 1 second if any new event.
        time.sleep(1)
        local_events = highlevel_handler.get_available_events()

        # If any, it logs it directly in the console.
```

```
38          for e in local_events:
39              logging.info(e)
40
41      # Keyboard <CTRL+C> interrupts the loop
42      except KeyboardInterrupt:
43          highlevel_handler.stop()
```

2.1.2 lazydog.handlers

module lazydog.handlers

synopsis Main module of lazydog including the aggregation logics.

author Clément Warneys <clement.warneys@gmail.com>

class lazydog.handlers.**HighlevelEventHandler**(*lowlevel_event_queue: lazydog.queues.DatedlocaleventQueue*, *local_states: lazydog.states.LocalState*)

Post-treats the low level events to suggest only high-level ones.

To do so, a high-level handler needs a *DatedlocaleventQueue* (an event queue containing the last lazydog events and inherited from `FileSystemEventHandler` so it is compatible with watchdog observers), that is already populated by a low-level watchdog observer, for example `InotifyObserver`, that retrieves the low-level file-system events.

The simplest way to instanciate *HighlevelEventHandler* is to use the *get_instance()* method. In this case, you only need to specify the directory to watch `watched_dir`. Two other optional parameters `hashing_function` and `custom_intializing_values` respectively allow to use custom hashing function (which will be use to compute the hashs of each file, in order to correlate copy events) and to accelerate the initialization phase (by providing already computed hash values of the current files in the watched directory, thus avoiding to compute them at the start). Please see the related methods documentation for more information.

This class inherits from `threading.Thread` so it works autonomously. It can be started `start()` method (from *Thread* module, and a stopping order can be send with *stop()* inner method.

Parameters

- **lowlevel_event_queue** (*DatedlocaleventQueue*) – An event queue containing the last lazydog events. Note that the provided queue shall be already associated with a low-level watchdog observer that retrieves the low-level file-system events (in order to fill the queue).
- **local_states** (*LocalState*) – The reference state of the local files in the watched directory. This state will be dynamically updated by the handler, depending on the low-level events. This state contains also the path of the watched directory.

Returns A non-running high-level lazydog events handler.

Return type *HighlevelEventHandler*

POSTTREATMENT_TIME_LIMIT = datetime.timedelta(0, 2)

If neither new low-level events nor high-level post-treatments appends during this 2-seconds delay, the current events in the queue are ready to be emitted the listener, when using *get_available_events()* method.

CREATE_EVENT_TIME_LIMIT_FOR_EMPTY_FILES = datetime.timedelta(0, 2)

Deprecated. At the beginning of the project, empty file creation was more delayed before the handler emits them. Because empty file are often created and then rapidly renamed and modified... The idea was

to limit the number of high-level events that were being sent. But this specific behaviour could generate unwanted problems for the third-application using this library.

classmethod get_instance(*watched_dir: str*, *hashing_function=None*, *custom_intializing_values=None*)

This method provides you with the simplest way to instanciate *HighlevelEventHandler*. You only need to specify the directory to watch `watched_dir`. Two other optional parameters `hashing_function` and `custom_intializing_values` respectively allow to use custom hashing function (which will be use to compute the hashs of each file, in order to correlate copy events) and to accelerate the initialization phase (by providing already computed hash values of the current files in the watched directory, thus avoiding to compute them at the start).

Parameters

- **watched_dir** (*str*) – The path you want to watch.
- **hashing_function** (*function*) – Custom hashing function that will be use to compute the hashs of each file, in order the handler is able to correlate copy events. The function shall be defined with the same parameters and retrun format than *default_hash_function()*.
- **custom_intializing_values** (*LocalState*) – Providing custom intializing values accelerate the initialization phase by providing already computed hash values of all the files currently in the watched directory. The provided dictionary shall cover all the local files and directory because hash values will not be computed for missing files. If some reference were missing in the provided dictionary, they can be completed later using *save_locals()* method. For more information about the structure of this parameter, please see the documentation of *LocalState*.

Returns An already running high-level lazydog events handler.

Return type *HighlevelEventHandler*

stop()

Set the `threading.Event` so that the handler thread knows that it has to stop running. Call this method when you want to preperly stop the handler. The handler will then stop a few seconds afterwards.

posttreat_lowlevel_event(*local_event: lazydog.events.LazydogEvent*)

Executes the main logics of the High-level Handler. These are all the aggregation rules, depending on the order of arrival of the low-level event, how to identify the relation between them and when to decide to aggregate them, or to transform them into a high-level *Copied* or a *Moved* event.

Please read directly the commented code for more information about these rules. Here is a summary of the execution:

- Aggregation rules:
  - Using an `InotifyObserver`, *Deleted* events arrive backward, which means that if you delete a directory with some files inside, you will get first a *Deleted* event for the inside files then another one for their parent directory. So if we find a *Deleted* event for a directory, we remove every children *Deleted* events previously queued. Note that if a *Deleted* event arrives after a *Modified* event or anything else for the same file or folder, then we just remove (or adapt) the previous related events.
  - Using an `InotifyObserver`, *Moved* events are the most simple to post-treat: if you move a folder with sub-files, you only get one low-level event. So nothing to aggregate here... The only thing is when a *Moved* event is rapidly succeding a *Created* event (or anything else), then you have to adapt the original event in the queue.
  - *Modified* events are easy to aggregate to other ones. They are often meaningless, since a low-level *Whatever* event often comes with one or more *Modified* events, so we often just ignore these

Modified events... Note that when you copy or create a large file, you will get multiple low-level *Modified* events per seconds that you will have to ignore (since you want to do a high-level lazy observer).

- If the new event is not related to any other already-listed events, thent it is added to the queue as a new high-level event.
- Transformation of *Created* events into *Copied* ones, if one or more potentiel sources have been found for the *Created* event. The identification of the sources is based on the *file_size*, the *file_mtime* and the *file_hash* attributes. The first step concerns only the files. Then at the end, if any event has been transformed into a *Copied* one, the `_posttreat_copied_folder()` helper method is called.

save_locals (*file_path*, *file_references*)
Directly modifies the local state dictionary associated to the handler, by providing new reference for a file or folder. This method should be use in combination of the optional parameter `custom_intializing_values` when calling *get_instance()*. In the case you rapidly initialize the handler with some files values, and then you see that some of these values are not good or that some files are missing, you can adjust by passing new values or new files with this method. The data structure is the almost the same.

Parameters

- **file_path** (*str*) – The relative path of the file or folder you want to add.
- **file_references** (*list*) – A list of 3 values in the following order: `file_hash`, `file_size`, `file_mtime`

Returns `None`

get_available_events () → list
Returns a list of high-level post-treated and ready events. Ready in the sense that the *POSTTREATMENT_TIME_LIMIT* has been reached without any new low-level events coming...

run ()
Threading module method, that is executed when calling `start()` method. The thread is running in a loop until you call the *stop()* method. Until then, it just check regularly if there is any new queued events emitted by the watchdog oberver. If any, it post-treats it calling the *posttreat_lowlevel_event()* method.

2.1.3 lazydog.events

module lazydog.events

synopsis Definitions of the high-level lazydog events, based on the low-level watchdog ones, which are now aggregable and also convertible to copy or move events.

author Clément Warneys <clement.warneys@gmail.com>

Possible type of lazydog events:

- *EVENT_TYPE_CREATED* for the creation of a file or folder
- `EVENT_TYPE_MODIFIED` for the creation of a file or folder (whatever the modification concerns: metadata or content)
- *EVENT_TYPE_MOVED* for the move of a file or folder
- *EVENT_TYPE_COPIED* for the copy of a file or folder
- *EVENT_TYPE_DELETED* for the deletion of a file or folder

Note: Some kind of events such as Moved and Copied have 2 path attributes: *path* for the origin path, and *to_path* for the destination path. Other kinds have only the *path* attribute.

The *ref_path* attribute always refers to the current location of the file (*to_path* if any, else *path*). All the paths are always relative to the main watched directory.

Lazydog has the ability to aggregate related low-level events. For example, in the case of multiple deletion events, each of one under the same parent directory, the lazydog handler will emit only one deletion event, with the path of the common parent directory.

Lazydog is also able to correlate almost simultaneous deletion and creation events into a unique moved event, if the low-level events are related. Or mutiple creation events into a unique copied event, if the new files and folders were already existing elsewhere in the main watched folder.

All these correlations are mainly done by the *HighlevelEventHandler* class, but some helper methods are defined in the *LazydogEvent* class such as *add_source_paths_and_transforms_into_copied_event()* or *update_main_event()*.

class lazydog.events.**LazydogEvent**(*event: watchdog.events.FileSystemEvent*, *local_states: lazydog.states.LocalState*)

Main class of *lazydog.events* module. Initialization with a low-level watchdog event that is then converted into high-level lazydog event.

Note: The local path of the event is referenced as a relative path starting from the absolute path of the watched directory. For this mechanism, the Lazydog event needs a reference, which is given at the initialisation with a `LocalState` reference.

Parameters

- **event** (`FileSystemEvent`) – A low-level watchdog event.
- **local_states** (*LocalState*) – The reference state of the local files in the watched directory. Including the absolute path of the watched directory, thus allowing to manage high-level event with relative path.

Returns A high-level lazydog event (converted from low-level watchdog event).

Return type *LazydogEvent*

EVENT_TYPE_CREATED = 'created'
Created event type, imported from `watchdog` module

EVENT_TYPE_DELETED = 'deleted'
Deleted event type, imported from `watchdog` module

EVENT_TYPE_MOVED = 'moved'
Moved event type, imported from `watchdog` module

EVENT_TYPE_C_MODIFIED = 'modified'
Content modified event type, imported from `lazydog.revised_watchdog` module

EVENT_TYPE_M_MODIFIED = 'metadata'
Metadata modified event type, imported from `lazydog.revised_watchdog` module

EVENT_TYPE_COPIED = 'copied'
New kind of event, that does not exist in watchdog python module. Copied event can only be obtained by transforming Created events. The transformation decision is made by the *HighlevelEventHandler* and is based on the existing files or folders in the watched directory.

path
Origin path of the event.

to_path
Destination path of the event, if any, else `None`.

ref_path
Refers to the current location of the file or the event, which is *to_path* if any, else *path*.

parent_rp
Refers to the directory name of the event. If the directory name is already the main watched directory, `None` is returned.

basename
Returns the filename or directory name of the related file or dir.

absolute_ref_path
Returns the absolute path of the current location of the file or dir.

is_directory() → bool
Returns `True` if the event is related to a directory.

is_moved_event() → bool
Returns `True` if the event is a file or dir move.

is_dir_moved_event() → bool
Returns `True` if the event is a dir move.

is_deleted_event() → bool
Returns `True` if the event is a file or dir deletion.

is_dir_deleted_event() → bool
Returns `True` if the event is a dir deletion.

is_created_event() → bool
Returns `True` if the event is a file or dir creation.

is_dir_created_event() → bool
Returns `True` if the event is a dir creation.

is_file_created_event() → bool
Returns `True` if the event is a file creation.

is_copied_event() → bool
Returns `True` if the event is a file or dir copy.

is_modified_event() → bool
Returns `True` if the event is a file or dir modification.

is_meta_modified_event() → bool
Returns `True` if the event is a file or dir modification of the metadata only.

is_data_modified_event() → bool
Returns `True` if the event is a file or dir modification of the content.

is_file_modified_event() → bool
Returns `True` if the event is a file modification.

is_meta_file_modified_event() → bool
Returns `True` if the event is a file modification of the metadata only.

is_data_file_modified_event() → bool
Returns `True` if the event is a file modification of the content.

is_dir_modified_event() → bool
Returns `True` if the event is a dir modification.

has_dest() → bool
Returns `True` if the event has a destination path (i.e. if it's a Moved or Copied event).

has_same_mtime_than(*previous_event*) → bool
Returns `True` if the event has the same modification time than the event in parameter.

has_same_size_than(*event*) → bool
Returns `True` if the event has the same size than the event in parameter.

has_same_path_than(*event*) → bool
Returns `True` if the event has the same *ref_path* than the event in parameter.

If both events have destination path, source paths are compared too.

has_same_src_path_than(*event*) → bool
Returns `True` if the *path* of event is the same than the *ref_path* of the event in parameter.

static p1_comes_after_p2(*p1: str*, *p2: str*) → bool
p1 and p2 are both paths (str format). This method is a basic comparison method to check if the first parameter p1 is stricly a parent path of the second parameter p2.

Returns `False` if both paths are identical.

static p1_comes_before_p2(*p1: str*, *p2: str*) → bool
Same than *p1_comes_after_p2()* method, but opposite result.

comes_before(*event*) → bool
Same than *comes_after()* method, but opposite result.

same_or_comes_before(*event*) → bool
Same than *comes_before()* method, but also `True` when both events have identical paths.

comes_after(*event*, *complete_check: bool = True*) → bool
Same result than *p1_comes_after_p2()*, comparing current event *ref_path* path (as p1), to the *ref_path* path of the event in parameter (as p2).

If both events have a destination path, source paths are compared too.

Returns `False` if both paths are identical.

same_or_comes_after(*event*) → bool
Same than *comes_after()* method, but also `True` when both events have identical paths.

static datetime_difference_from_now(*dt: datetime.datetime*) → datetime.datetime
Returns `datetime.datetime` object representing time difference between the datetime in parameter, and now.

idle_time() → datetime.datetime
Returns time difference between last time this event has been updated and now.

Note: Event updates occur when the event is aggregated to another related event, or also when the event is transformed into a copied or a moved one...

---

file_hash
Returns the file hash of the file related to the event if any, else `None`. File hash value is saved into a private variable, in order to avoid useless computation time...

static count_files_in(*absolute_dir_path: str*) → int
Counts all non-empty (file size > 0) files in `absolute_dir_path` directory and all its sub-directories. Returns `None` if the `absolute_dir_path` is not a directory.

Note: Be careful: `absolute_dir_path` has to represent absolute path (not a relative one).

---

dir_files_qty
Counts all non-empty (file size > 0) files in the related path of the event, and all its sub-directories. Returns `None` if the event is not related to a directory.

static get_file_size(*absolute_file_path: str*) → int
Returns the size of the file at the specified absolute path if any, else `None`.

file_size
Size of the file related to the event if any, else `None`. File size value is saved in a private variable, in order to avoid useless sollicitation of file-system.

is_empty() → bool
Returns `True` if the event is related to an empty directory, or if the event is related to an empty file (size = 0).

file_mtime
Last modification time of the file related to the event if any, else `None`. File modification time value is saved in a private variable, in order to avoid useless sollicitation of file-system.

file_inode
Inode of the file related to the event if any, else `None`. Inode value is saved in a private variable, in order to avoid useless sollicitation of file-system.

Note: This property seems now useless, and could be deprecated.

---

update_main_event(*main_event*)
High level helper method to facilitate the work of the *HighlevelEventHandler*. When different events are identified as related ones, this method is merging the current event in the main one (in parameter).

General idea is to update paramters of the main event, such as *file_inode*, *file_mtime*, *file_size*, *file_hash*, and also the dates of occurence (which are needed to manage an aggregation time limit).

Each related events, including the main event itself, are all listed in `related_events` list, to keep track of them.

add_source_paths_and_transforms_into_copied_event(*src_paths: set*)
High level helper method to facilitate the work of the *HighlevelEventHandler*. When a creation event is actually identified as a copied one, this method is transforming the current event in a copied one.

The old *path* attribute is converted into a *to_path* one. And the *path* id filled with one of the identified possible source paths (this identification is the job of the *HighlevelEventHandler*).

To get prepared to potential future aggregation of multiple copied events (for example in the case of a copied directory), we need to keep track of all the possible source paths which are then saved into a `possible_src_paths` attribute.

2.1.4 lazydog.queues

module lazydog.queues

synopsis Bufferizes lazydog events pending for a possible aggregation with other simultaneous events.

author Clément Warneys <clement.warneys@gmail.com>

class lazydog.queues.**DatedlocaleventQueue**(*local_states: lazydog.states.LocalState*)
Basically accumulates all the events emited by a watchdog oberver. It inherits from `FileSystemEventHandler`, so it is compatible with watchdog oberver. The *on_any_event()* catches the low-level event and adds them to the queue, after transorming them to *LazydogEvent*, which will further allow them to be post-treated by a *HighlevelEventHandler*.

The *DatedlocaleventQueue* has to be initialized with a *LocalState* object.

on_any_event(*event*)
Catch-all event handler.

Parameters **event** (`watchdog.events.FileSystemEvent`) – The event object representing the file system event.

next()
Provides with the oldest event that has been queued, removing it from the queue in the same time.

size()
Returns an integer corresponding to the current size of the queue.

is_empty()
`True` if the queue size is 0.

2.1.5 lazydog.states

module lazydog.states

synopsis Keeps track of the current state of the watched local directory. The idea is to save computational time, avoiding recomputing file hashes or getting size and modification time of each watched files (depending on the requested method), thus accelerating identification of copy events.

author Clément Warneys <clement.warneys@gmail.com>

class lazydog.states.**DualAccessMemory**
Helper class, used by *LocalState*. Sort of double-entry dictionary. When you save one tuple {key, value}, you can then access it both way:

- either from the key, using *get()*, or using accessor `object[key]`
- or from value, using *get_by_value()*. In this case, you will get a set of all the corresponding keys that references to this specific value.

To register a new key, you can either use *save()* method, or the accessor `object[key] = value`.

Finally you can check if a key is existing using the accessor `key in object`.

get(*key*)
Returns the value corresponding to the key in parameter, same behaviour as a dictionary. `None` if key is unknowned. You can also access it with `object[key]`.

get_by_value(*value*) → set
Returns a set of key corresponding to the value in parameter. Empty `set()` if value is not referenced.

save(*key: str*, *value*)
Registers the tuple {key, value} in order it is easily accessible both way. If key already exists with another value, the value is first removed, before registering the new one.

delete(*delete_key: str*)
Considering the *DualAccessMemory* has been designed to handle path key, this method not only deletes the `delete_key` in parameter, but it also deletes every children keys corresponding to the children paths of the parameter path `delete_key`.

move(*src_key: str*, *dst_key: str*)
Considering the *DualAccessMemory* has been designed to handle path key, this method not only moves the `src_key` in parameter to `dst_key` key, but it also moves every children keys corresponding to the children paths of the parameter path `src_key` to the related children path under the parameter path `dst_key`.

class lazydog.states.**LocalState**(*absolute_root_folder*, *custom_hash_function=None*, *custom_intializing_values: dict = None*)
Keeps track of the current state of the watched local directory, by listing every sub-files and sub-directories, and associating each of them with their size, modification time, and hash values.

When managing large directory, it can become very long to retrieves this information. But we need it very fast in order to be able to correlate *Created* event into *Copied* ones. Indeed, for this kind of correlation, we need to rapidly find every other file or folder that are having the same characteristics (that will then be eligible to be the source file or folder).

LocalState is keeping tracks of files with two *DualAccessMemory* objects. The first one keeping tracks of couple `(size, modification time)`, and the second one of single `hash` value.

Hash values are computed depending on a default hashing function. This default method is based on the Dropbox hashing algorithm, but you can define your own one. You only have to respect the same parameter and return. See `_default_hashing_function()` method to see the needed parameters names and types and the return type.

In order to accelerate the initialization of *LocalState* when watching large diectory, you can initialize it with pre-computed initializing values of your own (that you have to know in the first place, for example by keeping track of them in a hard backup file, or if you already have to compute them in other place of your application, no need that the hash values have to be computed again... just send them at the initialization). Please looke at the `custom_intializing_values` parameter for more information.

Parameters

- **absolute_root_folder** (*str*) – Absolute path of the folder you need to keep track of. Note that ever sub-file and sub-folder will then be referenced with relative paths.
- **custom_hash_function** (*function*) – *Optional*. Default value is `_default_hashing_function()` is used, which is based on the Dropbox hashing algorithm. But you can also provides your own hashing function, as long as your respect the format of the default one.
- **custom_intializing_values** (*dict*) – *Optional*. If not provided or `None`, all sub-folders will be browsed at initialization, and for each file and folder, the file size, file modification time and file hash will be retrieves and computed (this operation can take a long time, depending on the number and size of the files, and on the hashing function). To accelerate this initialization process, you can provide __init__ method with pre-computed initializing values under a dictionary format with `key=file_path` and `value=[file_hash, file_size, file_time]`. You do not need to know the exact content of the main directory at the initialization, and if you later notice unexpected modifications compared to the initial values you sent, you can still correct each of them using the *save()* method.

Returns An initialized object representing local state of the aimed folder.

Return type *LocalState*

DEFAULT_DIRECTORY_VALUE = 'DIR'
Default hash value for directory (since directory are not hashed, and that we want to reserve `None` value to non existing directories).

absolute_local_path(*relative_path: str*) → str
Computes the absolute local path from a relative one.

Parameters **relative_path** (*str*) – Relative local path of the file or folder.

Returns Absolute local path of the same file or folder

Return type str

relative_local_path(*absolute_path: str*) → str
Same as *absolute_local_path()*, but opposite.

get_hash(*key: str*, *compute_if_none: bool = True*) → str
Gets the `file_hash` value of the file at the `key` relative path. If the file is unknown (and so the hash value is not yet computed), by default the hash value will be computed. This behaviour can be cancelled using `compute_if_none` parameter.

Parameters

- **key** (*str*) – Relative local path of the file or folder.
- **compute_if_none** (*boolean*) – *Optional*. `True` by default, which means that if the file is unknown (and so it is for the hash value), the hash value will be computed. Use `False` if you want to cancel this bahaviour, so the returned value will be `None`.

Returns File or directory hash value, if path exists, else `None`.

Return type str

get_files_by_hash_key(*hash_key: str*) → set
Returns a set of every file or directory paths for which the hash value corresponds to the `hash_key` parameter.

get_sizetime(*key: str*, *compute_if_none: bool = True*)
Gets the couple `(file_size, file_modification_time)` value of the file at the `key` relative path. Same behaviour than *get_hash()* method.

Parameters

- **key** (*str*) – Relative local path of the file or folder.
- **compute_if_none** (*boolean*) – *Optional*. `True` by default, which means that if the file is unknown (and so it is for the file size and modification time value), the values will be computed. Use `False` if you want to cancel this bahaviour, so the returned value will be `None`.

Returns File or directory couple (file_size, file_modification_time) value, if path exists, else `None`.

Return type str

get_files_by_sizetime_key(*sizetime_key*) → set
Returns a set of every file or directory paths for which the couple (file_size, file_modification_time) value corresponds to the `sizetime_key` parameter.

save(*key: str*, *file_hash*, *file_size*, *file_mtime*)
Allows an external object to add a new file or folder reference to the local state object, by giving already

computed hash, size and modification time values. Note that the values will not be neither checked nor recomputed.

If you prefer that the *LocalState* class computes these values itself, and add the file or folder reference, you can just call the *get_hash()* or *get_sizetime()* method. Note that then the *LocalState* object just compute the needed values: it can compute the hash value without having any reference in its sizetime dictionary. These one will only be computed when calling the related method.

Parameters

- **key** (*str*) – Relative local path of the file or folder.
- **file_hash** (*str*) – File hash value of the file or folder.
- **file_size** (*int*) – File size value of the file or folder. For information the size is computed with `os.path.getsize()` method, so the size is the number of bytes of the file.
- **file_mtime** (*int*) – File modification time value of the file or folder. For information the modification time is computed with `os.path.getmtime()` method, rounded to the third decimal, so the time is a number giving the number of seconds since the epoch, precise at the millisecond.

Returns `None`

delete(*delete_key: str*)
Deletes key recursively. This method can be called internally when detecting a file or folder does not exists anymore, or by an external objects, that do not need to keep track of this path anymore.

move(*src_key: str*, *dst_key: str*)
Moves key recursively. This method can be called by an external object, when you know a file or folder has been moved and that you want to keep the already computed values in reference, without recomputing them all.

2.1.6 lazydog.dropbox_content_hasher

module lazydog.dropbox_content_hasher

synopsis Function to get hash of a file, based on dropbox api hasher.

author Dropbox, Inc.

author Clément Warneys <clement.warneys@gmail.com>

`lazydog.dropbox_content_hasher.`**default_hash_function**(*absolute_path: str*, *default_directory_hash: str = 'DIR'*)
Main function in this module that returns the dropbox-like hash of any local file. If the local path does not exist, `None` is returned. If the local path is a directory, the `default_directory_hash` parameter is returned, or the default string "DIR".

Parameters

- **absolute_path** (*str*) – The absolute local path of the file or directory.
- **default_directory_hash** – *Optional*. The returned value in case the absolute path is a directory.

Returns The hash of the file or directory located in `absolute_path`. The hash is computed based on the default Dropbox API hasher. `None` if absolute local path does not exist.

Return type str

class lazydog.dropbox_content_hasher.**DropboxContentHasher**

Computes a hash using the same algorithm that the Dropbox API uses for the the "content_hash" metadata field.

The digest() method returns a raw binary representation of the hash. The hexdigest() convenience method returns a hexadecimal-encoded version, which is what the "content_hash" metadata field uses.

How to use it:

```
hasher = DropboxContentHasher()
with open('some-file', 'rb') as f:
    while True:
        chunk = f.read(1024)  # or whatever chunk size you want
        if len(chunk) == 0:
            break
        hasher.update(chunk)
print(hasher.hexdigest())
```

2.2 Revised Watchdog

This inner package is overloading the original **watchdog** package by revising and completing it, resolving the current situation where the useful **watchdog** package is not maintained anymore...

Please read original **watchdog** project documentation for more information: https://pypi.org/project/watchdog/

2.2.1 revised_watchdog.events

module revised_watchdog.events

synopsis File system events and event handlers.

author yesudeep@google.com (Yesudeep Mangalapilly)

author Clément Warneys <clement.warneys@gmail.com>

This module is overloading the original `watchdog.events` module by revising and completing it. Please read original **watchdog** project documentation for more information: https://github.com/gorakhargosh/watchdog

This module imports some definitions of watchdog.events and keeps them unchanged:

- `FileModifiedEvent`
- `DirModifiedEvent`
- `FileSystemEvent`
- *`FileSystemEventHandler`*
- `EVENT_TYPE_MOVED`
- `EVENT_TYPE_CREATED`
- `EVENT_TYPE_DELETED`

It adds the following definitions, in order to add some granularity in the `watchdog.events.ModifiedEvent` definition, thus differentiating content modification from only metadata (access date, owner, etc.) modification:

- *`MetaFileModifiedEvent`*
- *`TrueFileModifiedEvent`*
- *`MetaDirModifiedEvent`*

- *TrueDirModifiedEvent*
- EVENT_TYPE_C_MODIFIED
- EVENT_TYPE_M_MODIFIED

Finally, it overloads the FileSystemEventHandler class, in order to manage the new granularity of modified events:

- *FileSystemEventHandler*

class lazydog.revised_watchdog.events.**MetaFileModifiedEvent**(*src_path*)
File system event representing metadata file modification on the file system.

class lazydog.revised_watchdog.events.**TrueFileModifiedEvent**(*src_path*)
File system event representing true file content modification on the file system.

class lazydog.revised_watchdog.events.**MetaDirModifiedEvent**(*src_path*)
File system event representing metadata directory modification on the file system.

class lazydog.revised_watchdog.events.**TrueDirModifiedEvent**(*src_path*)
File system event representing true directory content modification on the file system.

class lazydog.revised_watchdog.events.**FileSystemEventHandler**
Base file system event handler that you can override methods from. With modified dispatch method, added *on_data_modified()* and *on_meta_modified()* methods, thus covering specific needs of lazydog.

dispatch(*event*)
Dispatches events to the appropriate methods.

Parameters **event** (FileSystemEvent) – The event object representing the file system event.

on_data_modified(*event*)
Called when a file or directory true content is modified.

Parameters **event** (DirModifiedEvent or FileModifiedEvent) – Event representing file or directory modification.

on_meta_modified(*event*)
Called when a file or directory metadata is modified.

Parameters **event** (DirModifiedEvent or FileModifiedEvent) – Event representing file or directory modification.

2.2.2 revised_watchdog.observers.inotify

module revised_watchdog.observers.inotify

synopsis inotify(7) based emitter implementation, enhanced implementation of original watchdog one.

author Sebastien Martini <seb@dbzteam.org>

author Luke McCarthy <luke@iogopro.co.uk>

author yesudeep@google.com (Yesudeep Mangalapilly)

author Tim Cuthbertson <tim+github@gfxmonk.net>

author Clément Warneys <clement.warneys@gmail.com>

platforms Linux 2.6.13+.

This module is overloading the original `watchdog.observers.inotify` module by revising and completing it. Please read original **watchdog** project documentation for more information: https://github.com/gorakhargosh/watchdog

The main changes concern some methods in the *`InotifyEmitter`* class:

- `on_thread_start()` This method now uses revised *`InotifyBuffer`*.
- *`queue_events()`* This method has been simplified in order to reduce the number of emitted low-level events, in comparison with original watchdog module.

class lazydog.revised_watchdog.observers.inotify.**InotifyEmitter**(*event_queue*, *watch*, *timeout=1*)

inotify(7)-based event emitter. Revised package mainly concerns *`queue_events()`* method, thus covering specific needs of lazydog package.

Parameters

- **event_queue** (`watchdog.events.EventQueue`) – The event queue to fill with events.
- **watch** (`watchdog.observers.api.ObservedWatch`) – A watch object representing the directory to monitor.
- **timeout** (*`float`*) – Read events blocking timeout (in seconds).

queue_events(*timeout*, *full_events=False*)

This method is classifying the events received from Inotify into watchdog events type (defined in `watchdog.events` module).

Parameters

- **timeout** (*`float`*) – Unused param (from watchdog original package).
- **full_events** (*`boolean`*) – If `True`, then the method will report unmatched move events as separate events. This means that if `True`, a file move event from outside the watched directory will result in a `watchdog.events.FileMovedEvent` event, with no origin. Else (if `False`), it will result in a `watchdog.events.FileCreatedEvent` event. This behavior is by default only called by a `InotifyFullEmitter`.

class lazydog.revised_watchdog.observers.inotify.**InotifyObserver**(*timeout=1*, *generate_full_events=False*)

Observer thread that schedules watching directories and dispatches calls to event handlers.

Please note that his class remains unmodified in revised_watchdog package. Only the `__init__()` method is overided in order it uses the new definition of *`InotifyEmitter`* class.

2.2.3 revised_watchdog.observers.inotify_c

module revised_watchdog.observers.inotify_c

author yesudeep@google.com (Yesudeep Mangalapilly)

author Clément Warneys <clement.warneys@gmail.com>

This module is overloading the original `watchdog.observers.inotify_c` module by revising and completing it. Please read original **watchdog** project documentation for more information: https://github.com/gorakhargosh/watchdog

Fundamental changes and corrections have been brought to the original *Inotify* class, whose behaviour was not correct when moving or deleting sub-directories.

class lazydog.revised_watchdog.observers.inotify_c.**Inotify**(*path*, *recursive=False*, *event_mask=33556422*)

Linux inotify(7) API wrapper class.

With modified *read_events()* method, and added _remove_watch_bookkeeping() method, thus covering specifics needs of lazydog.

Parameters

- **path** (*bytes*) – The directory path for which we want an inotify object.
- **recursive** (*boolean*) – True if subdirectories should be monitored. False otherwise.

read_events(*event_buffer_size=81920*)

Reads events from inotify and yields them to the Inotify buffer. This method has been largely modified from original watchdog module... Thus preventing from unwanted behaviour.

2.2.4 revised_watchdog.observers.inotify_buffer

module revised_watchdog.observers.inotify_c

author Thomas Amland <thomas.amland@gmail.com>

author Clément Warneys <clement.warneys@gmail.com>

This module is overloading the original watchdog.observers.inotify_buffer module by revising and completing it. Please read original **watchdog** project documentation for more information: https://github.com/gorakhargosh/watchdog

The main change is in the *InotifyBuffer* class, whose InotifyBuffer.__init__() method now uses revised watchdog *Inotify* class.

class lazydog.revised_watchdog.observers.inotify_buffer.**InotifyBuffer**(*path*, *recursive=False*)

A wrapper for *Inotify* that holds events for *delay* seconds. During this time, IN_MOVED_FROM and IN_MOVED_TO events are paired.

Please note that his class remains unmodified in revised_watchdog package. Only the __init__() method is overrided in order it uses the new definition of *Inotify* class.

Python Module Index

l

r

Index

H

I

L

M

N

O

P

Q

R

S

T

U