watchdog Documentation

Release 0.8.3

Yesudeep Mangalapilly

Aug 24, 2021

Contents

Python API library and shell utilities to monitor file system events.

CHAPTER 1

Directory monitoring made easy with

- A cross-platform API.
- A shell tool to run commands in response to directory changes.

Get started quickly with a simple example in *Quickstart*.

CHAPTER 2

Easy installation

You can use pip to install `watchdog` quickly and easily:

```
$ pip install watchdog
```

Need more help with installing? See *Installation*.

CHAPTER 3

User's Guide

3.1 Installation

watchdog requires Python 2.6 or above to work. If you are using a Linux/FreeBSD/Mac OS X system, you already have Python installed. However, you may wish to upgrade your system to Python 2.7 at least, because this version comes with updates that can reduce compatibility problems. See a list of *Dependencies*.

3.1.1 Installing from PyPI using pip

```
$ pip install watchdog
```

3.1.2 Installing from source tarballs

```
$ wget -c http://pypi.python.org/packages/source/w/watchdog/watchdog-0.8.3.
↪tar.gz
$ tar zxvf watchdog-0.8.3.tar.gz
$ cd watchdog-0.8.3
$ python setup.py install
```

3.1.3 Installing from the code repository

```
$ git clone --recursive git://github.com/gorakhargosh/watchdog.git
$ cd watchdog
$ python setup.py install
```

3.1.4 Dependencies

watchdog depends on many libraries to do its job. The following is a list of dependencies you need based on the operating system you are using.

Operating system Dependency (row)	Windows	Linux 2.6	**Mac OS X/** Darwin	BSD
XCode			Yes	
PyYAML	Yes	Yes	Yes	Yes
argh	Yes	Yes	Yes	Yes
argparse	Yes	Yes	Yes	Yes
select_backport (Python 2.6)			Yes	Yes
pathtools	Yes	Yes	Yes	Yes

Installing Dependencies

The `watchmedo` script depends on PyYAML which links with LibYAML. On Mac OS X, you can use homebrew to install LibYAML:

```
brew install libyaml
```

On Linux, use your favorite package manager to install LibYAML. Here's how you do it on Ubuntu:

```
sudo aptitude install libyaml-dev
```

On Windows, please install PyYAML using the binaries they provide.

3.1.5 Supported Platforms (and Caveats)

`watchdog` uses native APIs as much as possible falling back to polling the disk periodically to compare directory snapshots only when it cannot use an API natively-provided by the underlying operating system. The following operating systems are currently supported:

> **Warning:** Differences between behaviors of these native API are noted below.

Linux 2.6+ Linux kernel version 2.6 and later come with an API called inotify that programs can use to monitor file system events.

> **Note:** On most systems the maximum number of watches that can be created per user is limited to `8192`. `watchdog` needs one per directory to monitor. To change this limit, edit `/etc/sysctl.conf` and add:
>
> ```
> fs.inotify.max_user_watches=16384
> ```

Mac OS X The Darwin kernel/OS X API maintains two ways to monitor directories for file system events:

- kqueue
- FSEvents

`watchdog` can use whichever one is available, preferring FSEvents over `kqueue(2)`. `kqueue(2)` uses open file descriptors for monitoring and the current implementation uses Mac OS X File System Monitoring Performance Guidelines to open these file descriptors only to monitor events, thus allowing OS X to unmount volumes that are being watched without locking them.

Note: More information about how `watchdog` uses `kqueue(2)` is noted in *BSD Unix variants*. Much of this information applies to Mac OS X as well.

BSD Unix variants BSD variants come with kqueue which programs can use to monitor changes to open file descriptors. Because of the way `kqueue(2)` works, `watchdog` needs to open these files and directories in read-only non-blocking mode and keep books about them.

`watchdog` will automatically open file descriptors for all new files/directories created and close those for which are deleted.

Note: The maximum number of open file descriptor per process limit on your operating system can hinder `watchdog`'s ability to monitor files.

You should ensure this limit is set to at least **1024** (or a value suitable to your usage). The following command appended to your `~/.profile` configuration file does this for you:

```
ulimit -n 1024
```

Windows Vista and later The Windows API provides the ReadDirectoryChangesW. `watchdog` currently contains implementation for a synchronous approach requiring additional API functionality only available in Windows Vista and later.

Note: Since renaming is not the same operation as movement on Windows, `watchdog` tries hard to convert renames to movement events. Also, because the ReadDirectoryChangesW API function returns rename/movement events for directories even before the underlying I/O is complete, `watchdog` may not be able to completely scan the moved directory in order to successfully queue movement events for files and directories within it.

Note: Since the Windows API does not provide information about whether an object is a file or a directory, delete events for directories may be reported as a file deleted event.

OS Independent Polling `watchdog` also includes a fallback-implementation that polls watched directories for changes by periodically comparing snapshots of the directory tree.

3.2 Quickstart

Below we present a simple example that monitors the current directory recursively (which means, it will traverse any sub-directories) to detect changes. Here is what we will do with the API:

1. Create an instance of the `watchdog.observers.Observer` thread class.
2. Implement a subclass of `watchdog.events.FileSystemEventHandler` (or as in our case, we will use the built-in `watchdog.events.LoggingEventHandler`, which already does).
3. Schedule monitoring a few paths with the observer instance attaching the event handler.
4. Start the observer thread and wait for it generate events without blocking our main thread.

By default, an `watchdog.observers.Observer` instance will not monitor sub-directories. By passing `recursive=True` in the call to `watchdog.observers.Observer.schedule()` monitoring entire directory trees is ensured.

3.2.1 A Simple Example

The following example program will monitor the current directory recursively for file system changes and simply log them to the console:

```
import sys
import time
import logging
from watchdog.observers import Observer
from watchdog.events import LoggingEventHandler

if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO,
                        format='%(asctime)s - %(message)s',
                        datefmt='%Y-%m-%d %H:%M:%S')
    path = sys.argv[1] if len(sys.argv) > 1 else '.'
    event_handler = LoggingEventHandler()
    observer = Observer()
    observer.schedule(event_handler, path, recursive=True)
    observer.start()
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()
```

To stop the program, press Control-C.

3.3 API Reference

3.3.1 *watchdog.events*

3.3.2 *watchdog.observers.api*

3.3.3 *watchdog.observers*

3.3.4 *watchdog.observers.polling*

3.3.5 *watchdog.utils*

module watchdog.utils

synopsis Utility classes and functions.

author yesudeep@google.com (Yesudeep Mangalapilly)

Classes

class watchdog.utils.**BaseThread**

Bases: threading.Thread

Convenience class for creating stoppable threads.

daemon
A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isAlive()
Return whether the thread is alive.

This method is deprecated, use is_alive() instead.

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

on_thread_start()
Override this method instead of *start()*. *start()* calls this method.

This method is called right before this thread is started and this object's run() method is invoked.

on_thread_stop()
Override this method instead of *stop()*. *stop()* calls this method.

This method is called immediately after the thread is signaled to stop.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

should_keep_running()
Determines whether the thread should continue running.

start()
Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

stop()
Signals the thread to stop.

3.3.6 *watchdog.utils.dirsnapshot*

module watchdog.utils.dirsnapshot

synopsis Directory snapshots and comparison.

author yesudeep@google.com (Yesudeep Mangalapilly)

Where are the moved events? They "disappeared"

This implementation does not take partition boundaries into consideration. It will only work when the directory tree is entirely on the same file system. More specifically, any part of the code that depends on inode numbers can break if partition boundaries are crossed. In these cases, the snapshot diff will represent file/directory movement as created and deleted events.

Classes

class watchdog.utils.dirsnapshot.**DirectorySnapshot**(*path*, *recursive=True*, *walker_callback=<function DirectorySnapshot.<lambda>>*, *stat=<built-in function stat>*, *listdir=<built-in function listdir>*)

Bases: object

A snapshot of stat information of files in a directory.

Parameters

- **path** (str) – The directory path for which a snapshot should be taken.
- **recursive** (bool) – True if the entire directory tree should be included in the snapshot; False otherwise.
- **walker_callback** – Deprecated since version 0.7.2.
- **stat** – Use custom stat function that returns a stat structure for path. Currently only st_dev, st_ino, st_mode and st_mtime are needed.

  A function with the signature walker_callback(path, stat_info) which will be called for every entry in the directory tree.

- **listdir** – Use custom listdir function. See `os.listdir` for details.

inode(*path*)
Returns an id for path.

path(*id*)
Returns path for id. None if id is unknown to this snapshot.

paths
Set of file/directory paths in the snapshot.

stat_info(*path*)
Returns a stat information object for the specified path from the snapshot.

Attached information is subject to change. Do not use unless you specify *stat* in constructor. Use *inode()*, `mtime()`, `isdir()` instead.

Parameters **path** – The path for which stat information should be obtained from a snapshot.

class watchdog.utils.dirsnapshot.**DirectorySnapshotDiff**(*ref*, *snapshot*)
Bases: `object`

Compares two directory snapshots and creates an object that represents the difference between the two snapshots.

Parameters

- **ref** (*DirectorySnapshot*) – The reference directory snapshot.
- **snapshot** (*DirectorySnapshot*) – The directory snapshot which will be compared with the reference snapshot.

dirs_created
List of directories that were created.

dirs_deleted
List of directories that were deleted.

dirs_modified
List of directories that were modified.

dirs_moved
List of directories that were moved.

Each event is a two-tuple the first item of which is the path that has been renamed to the second item in the tuple.

files_created
List of files that were created.

files_deleted
List of files that were deleted.

files_modified
List of files that were modified.

files_moved
List of files that were moved.

Each event is a two-tuple the first item of which is the path that has been renamed to the second item in the tuple.

3.4 Contributing

Welcome hacker! So you have got something you would like to see in watchdog? Whee. This document will help you get started.

3.4.1 Important URLs

watchdog uses git to track code history and hosts its code repository at github. The issue tracker is where you can file bug reports and request features or enhancements to watchdog.

3.4.2 Before you start

Ensure your system has the following programs and libraries installed before beginning to hack:

1. Python
2. git
3. ssh
4. XCode (on Mac OS X)
5. select_backport (on BSD/Mac OS X if you're using Python 2.6)

3.4.3 Setting up the Work Environment

watchdog makes extensive use of zc.buildout to set up its work environment. You should get familiar with it.

Steps to setting up a clean environment:

1. Fork the code repository into your github account. Let us call you hackeratti for the sake of this example. Replace hackeratti with your own username below.
2. Clone your fork and setup your environment:

```
$ git clone --recursive git@github.com:hackeratti/watchdog.git
$ cd watchdog
$ python tools/bootstrap.py --distribute
$ bin/buildout
```

Important: Re-run bin/buildout every time you make a change to the buildout.cfg file.

That's it with the setup. Now you're ready to hack on watchdog.

3.4.4 Enabling Continuous Integration

The repository checkout contains a script called autobuild.sh which you must run prior to making changes. It will detect changes to Python source code or restructuredText documentation files anywhere in the directory tree and rebuild sphinx documentation, run all tests using nose, and generate coverage reports.

Start it by issuing this command in the watchdog directory checked out earlier:

```
$ tools/autobuild.sh
...
```

Happy hacking!

CHAPTER 4

Contribute

Found a bug in or want a feature added to `watchdog`? You can fork the official code repository or file an issue ticket at the issue tracker. You can also ask questions at the official mailing list. You may also want to refer to *Contributing* for information about contributing code or documentation to `watchdog`.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

W

Index