logaware

Release 0.1.0

Oct 31, 2018

Contents

Contents:

CHAPTER 1

Overview

1.1 logaware

docs	
tests	
package	

Python Logger that is context aware

- Free software: BSD license

1.1.1 Installation

```
pip install logaware
```

1.1.2 Documentation

https://logaware.readthedocs.org/

1.1.3 Development

To run the all tests:

```
tox
```

CHAPTER 2

Installation

At the command line:

```
pip install logaware
```

CHAPTER 3

Usage

To use logaware in a project:

```
import logaware
```

CHAPTER 4

Reference

4.1 logaware

4.1.1 Logger

class logaware.logger.**AwareLogger**

Similar to a `logging.Logger` but is context aware. There is only one `ContextLogger` per context and it automatically figures out the module that is being logged from. Other information about the context can also be injected into the log message.

The `ContextLogger` will use `logging.getLogger()` to get a Logger based on which module the logger is being called from. For example, if this was being called in `logaware.logger`, the log message would include `logaware.logger` as the logger name:

```
>>> import logging
>>> logging.getLogger().setLevel(logging.DEBUG)
>>> log = AwareLogger()
>>> log.info('Test message').module
'...logaware.logger'
```

CRITICAL = <LogLevel CRITICAL (50)>

CRITICAL Log level

DEBUG = <LogLevel DEBUG (10)>

ERROR = <LogLevel ERROR (40)>

FATAL = <LogLevel CRITICAL (50)>

INFO = <LogLevel INFO (20)>

WARN = <LogLevel WARNING (30)>

WARNING = <LogLevel WARNING (30)>

critical(**args*, ***kwargs*)

Log CRITICAL level message. See *AwareLogger.log()* for argument info.

debug (**args*, ***kwargs*)
Log DEBUG level message. See *AwareLogger.log()* for argument info.

error (**args*, ***kwargs*)
Log ERROR level message. See *AwareLogger.log()* for argument info.

exception (**args*, ***kwargs*)
Log ERROR level message with traceback. See *AwareLogger.log()* for argument info.

fatal (**args*, ***kwargs*)
Alias for *AwareLogger.critical()*

get_level_name (*level*)
Get the textual representation of logging `level`.

Parameters **level** (*int*) – Logging level

Returns Name of logging level

Return type unicode

info (**args*, ***kwargs*)
Log INFO level message. See *AwareLogger.log()* for argument info.

isEnabledFor (*level*)
Is this logger enabled for level 'level'?

Note: Wrapper around native logger method.

Parameters **level** (*int*) – Logging level

Returns Whether or not logging is enabled for *level*.

Return type boolean

log (*level*, *msg*, ***kwargs*)
Log a message at specified `level`

Parameters

- **level** (*int*) – Logging level
- **msg** (*unicode*) – Log message
- ****kwargs** – Extra logging parameters and substitution parameters for log message.

Returns LogRecord sent to logging handler

Return type logging.LogRecord

warn (**args*, ***kwargs*)
Alias for *AwareLogger.warning()*

warning (**args*, ***kwargs*)
Log WARNING level message. See *AwareLogger.log()* for argument info.

exception logaware.logger.**LogFormatException** (*message*, *original*)
Exception raised if there is an error processing the substitution format of a message.

class logaware.logger.**LogLevel** (*level*, *name*, *traceback=False*)
A logging level

Parameters

- **level** (*int*) – Log level
- **name** (*unicode*) – Level name

- **traceback** (*bool*) – Include traceback when logging

class logaware.logger.**LoggerMetaClass**
Metaclass that sets up log levels

logaware.logger.**log_method_factory**(*name*, *level*, *traceback=False*)
Create a method that will log at the specified level

Parameters

- **name** (*bytes*) – Method name
- **level** (LogLevel) – Logging level
- **traceback** (*bool*) – Include traceback when logging

4.1.2 MetaLogger

class logaware.metalogger.**LogMeta**(***kwargs*)
Read only meta data for a log message.

Values must be JSON serializable.

Automatically serializes as JSON when stringified.

to_dict()

Returns Dictionary of meta data

Return type dict

class logaware.metalogger.**LogMetaManager**(*meta=None*)
Track additional metadata for logging. The metadata is stored on this instance so this instance can not be re-used for multiple requests.

To add other information to the log output, use set_meta:

```
>>> import logging
>>> logging.getLogger().setLevel(logging.DEBUG)
>>> meta = LogMetaManager()
>>> meta.set_meta(user='foo', nothing=None)
<LogMeta {"user":"foo"}>
>>> log = MetaAwareLogger(getter=lambda: meta.get_meta())
>>> log.info('Test message').meta
<LogMeta {"user":"foo"}>
```

get_meta()

Returns Current meta data

Return type *LogMeta*

set_meta(***kwargs*)
Add metadata to the current meta context

Parameters ****kwargs** – Meta data to add to log records. Must be JSON serializable.

Returns Current meta

Return type dict

class logaware.metalogger.**MetaAwareLogger**(*getter*)
Similar to a AwareLogger and also track additional metadata.

Parameters **getter** (*callable*) – Callable to get the current LogMeta. It's up to the framework to decide how meta is obtained.

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

logaware could always use more documentation, whether as part of the official logaware docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/six8/logaware/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.4 Development

To set up *logaware* for local development:

1. Fork logaware on GitHub.
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/logaware.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.

It will be slower though ...

CHAPTER 6

Authors

- Mike Thornton - http://devdetails.com/

CHAPTER 7

Changelog

7.1 0.2.1

- Implemented *AwareLogger.isEnabledFor* to make more compatible as a drop-in replacement for native loggers.

7.2 0.1.0

- Initial implementation

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index