logquacious Documentation

Release 0.5.0

Tony S Yu

May 07, 2019

CONTENTS:

CHAPTER

ONE

LOGQUACIOUS

Logquacious is a set of simple logging utilities to help you over-communicate. (Logorrhea would've been a good name, if it didn't sound so terrible.)

Good application logging is easy to overlook, until you have to debug an error in production. Logquacious aims to make logging as easy as possible. You can find read more at the official ReadTheDocs documentation.

1.1 Quick start

To get started, first make sure logquacious is installed:

```
$ pip install logquacious
```

You'll also need to set up logging for your application. For this example, we'll use a really simple configuration:

```
import logging

logging.basicConfig(format='%(levelname)s: %(message)s',
                    level=logging.DEBUG)
```

Note that this simple configuration is used for demonstration purposes, only. See the Logging Cookbook in the official Python docs for examples of options used for real logging configuration.

The main interface to `logquacious` is the `LogManager`, which can be used for normal logging:

```
import logquacious

log = logquacious.LogManager(__name__)
```

```
log.debug('Nothing to see here.')
```

Due to our simplified logging format defined earlier, that would output:

```
DEBUG: Nothing to see here.
```

That isn't a very interesting example. In addition to basic logging, `LogManager` has a `context` attribute for use as a context manager:

```
>>> with log.context.debug('greetings'):
...     print('Hello!')
DEBUG: Enter greetings
Hello!
DEBUG: Exit greetings
```

The same attribute can be used as a decorator, as well:

```
@log.context.info
def divide(numerator, denominator):
    if denominator == 0:
        log.warning('Attempted division by zero. Returning None')
        return None
    return numerator / denominator

>>> divide(1, 0)
INFO: Call `divide()`
WARNING: Attempted division by zero. Returning None
INFO: Return from `divide`
```

Even better, you can log input arguments as well:

```
@log.context.info(show_args=True, show_kwargs=True)
def greet(name, char='-'):
    msg = 'Hello, {name}'.format(name=name)
    print(msg)
    print(char * len(msg))

>>> greet('Tony', char='~')
INFO: Call `greet('Tony', char='~')`
Hello, Tony
~~~~~~~~~~~
INFO: Return from `greet`
```

There's also a special context manager for suppressing errors and logging:

```
with log.and_suppress(ValueError, msg="It's ok, mistakes happen"):
    raise ValueError('Test error')
```

```
[ERROR] It's ok, mistakes happen
Traceback (most recent call last):
  File "/Users/tyu/code/logquacious/logquacious/log_manager.py", line 103, in and_
↪suppress
    yield
  File "scripts/example.py", line 26, in <module>
    raise ValueError('Test error')
ValueError: Test error
```

Note the traceback above is logged, not streamed to stderr.

1.2 Configuration

The message templates used by `LogManager.context` can be configured to your liking by passing a `context_templates` argument to `LogManager`:

```
log = logquacious.LogManager(__name__, context_templates={
    'context.start': '=============== Enter {label} ===============',
    'context.finish': '=============== Exit {label} ===============',
})

with log.context.debug('greetings'):
    print('Hello!')
```

```
[DEBUG] =============== Enter greetings ===============
Hello!
[DEBUG] =============== Exit greetings ===============
```

The general format for `context_templates` keys is:

```
[CONTEXT_TYPE.]('start'|'finish')[.LOG_LEVEL_NAME]
```

where square-brackets marks optional fields.

`CONTEXT_TYPE` can be any of the following:

- `function`: Template used when called as a decorator.
- `context`: Template used when called as a context manager.

`LOG_LEVEL_NAME` can be any of the following logging levels:

- `DEBUG`
- `INFO`
- `WARNING`
- `ERROR`
- `CRITICAL`

For example, consider the cascade graph for `function.start.DEBUG`, which looks like:

```
     function.start.DEBUG
          /       \
start.DEBUG       function.start
          \       /
            start
```

The cascade is performed using a breadth-first search. If `function.start.DEBUG` is not defined, check `start.DEBUG` then check `function.start` *BEFORE* checking `start`.

The default configuration is:

```
DEFAULT_TEMPLATES = {
    'start': 'Enter {label}',
    'finish': 'Exit {label}',
    'function.start': 'Call `{label}({arguments})`',
    'function.finish': 'Return from `{label}`',
}
```

Note that custom configuration *updates* these defaults. For example, if you want to if you want to skip logging on exit for all context managers and decorators, you'll have set *both* `'finish'` and `'function.finish'` to `None` or an empty string.

As you can see above, two template variables may be passed to the template string: `label` and `arguments`. When called as a context manager, the `label` is the first argument to the context manager and `arguments` is always

empty. When called as a decorator, `label` is the function's `__name__` and `arguments` a string representing input arguments, if `show_args` or `show_kwargs` parameters are `True`.

1.3 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install logquacious, run this command in your terminal:

```
$ pip install logquacious
```

This is the preferred method to install logquacious, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for logquacious can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/tonysyu/logquacious
```

Or download the tarball:

```
$ curl  -OL https://github.com/tonysyu/logquacious/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

THREE

USAGE

To use logquacious in a project:

```
import logquacious
```

Note: Coming Soon: This page is under construction.

CHAPTER

FOUR

LOGQUACIOUS

4.1 logquacious package

4.1.1 Submodules

4.1.2 logquacious.backport_configurable_stacklevel module

Backport of configurable stacklevel for logging added in Python 3.8.

See https://github.com/python/cpython/pull/7424

class logquacious.backport_configurable_stacklevel.**PatchedLoggerMixin**(**args*, ***kwargs*)

Bases: object

Mixin adding *temp_monkey_patched_logger* that allows stacklevel kwarg.

Classes that include this mixin have a *temp_monkey_patched_logger* context manager that allows the use of the *stacklevel* keyword argument from Python 3.8.

Classes using this mixin must have a *logging.Logger* instance as an attribute of the class. By default, this is assumed to be named *logger*, but you can override the *logger_attribute* class attribute with the name of a different attribute.

logger_attribute = 'logger'
Name of logger instance on the class inheriting this mixin.

temp_monkey_patched_logger()
Temporarily monkey patch logger to allow overriding log records.

The monkey patching is reset so that the behavior change is limited to the scope of this logger.

logquacious.backport_configurable_stacklevel.**patch_logger**(*logger_class*)
Return logger class patched with stacklevel keyword argument.

4.1.3 logquacious.cascading_config module

class logquacious.cascading_config.**CascadingConfig**(*config_values=None*, *cascade_map=None*)

Bases: collections.abc.Mapping

Cascading configuration values.

This class allows you to define parameter names that can match exactly, but if it doesn't, parameter names will be searched as defined by *cascade_map*. *cascade_map* basically defines edges of a dependency graph, which is then used for a breadth-first search of parameter values.

Parameters

- **config_values** (*dict or list of (key, value) pairs*) – Default values for a configuration, where keys are the parameter names and values are the associated value.
- **cascade_map** (*dict*) – Dictionary defining cascading defaults. If a parameter name is not found, indexing *cascade_map* with the parameter name will return the parameter to look for.
- **kwargs** (*dict*) – Keyword arguments for initializing dict.

cascade_list(*name*)
Return list of cascade hierarchy for a given configuration name.

cascade_path(*name*)
Return string of describing cascade.

get(*name*, *default=None*, *_prev=None*)
Return best matching config value for *name*.

Get value from configuration. The search for *name* is in the following order:

- *self* (Value in global configuration)
- *default*
- Alternate name specified by *self.cascade_map*

This method supports the pattern commonly used for optional keyword arguments to a function. For example:

```
>>> def print_value(key, **kwargs):
...     print(kwargs.get(key, 0))
>>> print_value('a')
0
>>> print_value('a', a=1)
1
```

Instead, you would create a config class and write:

```
>>> config = CascadingConfig({'a': 0})
>>> def print_value(key, **kwargs):
...     print(kwargs.get(key, config.get(key)))
>>> print_value('a')
0
>>> print_value('a', a=1)
1
>>> print_value('b')
None
>>> config.cascade_map['b'] = 'a'
>>> print_value('b')
0
```

See examples below for a demonstration of the cascading of configuration names.

Parameters

- **name** (*str*) – Name of config value you want.
- **default** (*object*) – Default value if name doesn't exist in instance.

Examples

```
>>> config = CascadingConfig({'size': 0},
...                          cascade_map={'arrow.size': 'size'})
>>> config.get('size')
0
>>> top_choice = {'size': 1}
>>> top_choice.get('size', config.get('size'))
1
>>> config.get('non-existent', 'unknown')
'unknown'
>>> config.get('arrow.size')
0
>>> config.get('arrow.size', 2)
2
>>> top_choice.get('size', config.get('arrow.size'))
1
```

4.1.4 logquacious.constants module

4.1.5 logquacious.context_templates module

class logquacious.context_templates.**ContextTemplates**(*config_dict=None*)
Bases: *logquacious.cascading_config.CascadingConfig*

classmethod resolve(*templates*)

4.1.6 logquacious.log_context module

class logquacious.log_context.**LogContext**(*logger*, *templates=None*)
Bases: object

Manager for context managers/decorators used for logging.

debug
Decorator/context-manager with level *logging.DEBUG*.

info
Decorator/context-manager with level *logging.INFO*.

warning
Decorator/context-manager with level *logging.WARNING*.

error
Decorator/context-manager with level *logging.ERROR*.

fatal
Decorator/context-manager with level *logging.CRITICAL*.

4.1.7 logquacious.log_manager module

class logquacious.log_manager.**LogManager**(*name=None*, *context_templates=None*)
Bases: *logquacious.backport_configurable_stacklevel.PatchedLoggerMixin*

Logging manager for use as a logger, decorator, or contextmanager.

```
>>> log = LogManager(__name__)
>>> log.info('Normal logging statement')
```

[INFO] Normal logging statement

```
>>> @log.context.info
... def logged_function():
...     log.info('Inside logged_function')
```

[DEBUG] Start logged_function [INFO] Inside logged_function [DEBUG] Finish logged_function

```
>>> with log.context.info("context manager"):
...     log.info("Inside context manager")
```

[DEBUG] Start context manager [INFO] Inside context manger [DEBUG] Finish context manager

and_reraise(*allowed_exceptions*, *msg='Logging error and reraising'*, *level=40*, *exc_info=True*, *stacklevel=3*)
Context manager that logs and reraises given error.

Parameters

- **allowed_exceptions** – Exception(s) to log before reraising.
- **msg** – Message logged for exceptions.
- **level** – Logging level for logging exceptions.
- **exc_info** – If True, include exception info.
- **stacklevel** – Stacklevel of logging statement. Defaults to level 3 since this method (level=2) defers functionality to a helper utility (level=1), but logging should use the context where this is called (level=3).

and_suppress(*allowed_exceptions*, *msg='Suppressed error and logging'*, *level=40*, *exc_info=True*, *stacklevel=3*)
Context manager that logs and suppresses given error.

Parameters

- **allowed_exceptions** – Exception(s) to log and suppress.
- **msg** – Message logged for exceptions.
- **level** – Logging level for logging exceptions.
- **exc_info** – If True, include exception info.
- **stacklevel** – Stacklevel of logging statement. Defaults to level 3 since this method (level=2) defers functionality to a helper utility (level=1), but logging should use the context where this is called (level=3).

4.1.8 logquacious.utils module

class logquacious.utils.**HandleException**(*handled_exceptions*, *on_exception*)
Bases: contextlib.ContextDecorator

handled_exceptions = ()

on_exception()

logquacious.utils.**format_function_args**(*args*, *kwargs*, *show_args=False*, *show_kwargs=False*)

logquacious.utils.**get_logger**(*name_or_logger*)

logquacious.utils.**is_sequence**(*value*)
Return True if value is a non-string sequence.

See https://stackoverflow.com/a/1835259/260303

logquacious.utils.**is_string**(*value*)

4.1.9 Module contents

Top-level package for logquacious.

class logquacious.**LogManager**(*name=None*, *context_templates=None*)
Bases: *logquacious.backport_configurable_stacklevel.PatchedLoggerMixin*

Logging manager for use as a logger, decorator, or contextmanager.

```
>>> log = LogManager(__name__)
>>> log.info('Normal logging statement')
```

[INFO] Normal logging statement

```
>>> @log.context.info
... def logged_function():
...     log.info('Inside logged_function')
```

[DEBUG] Start logged_function [INFO] Inside logged_function [DEBUG] Finish logged_function

```
>>> with log.context.info("context manager"):
...     log.info("Inside context manager")
```

[DEBUG] Start context manager [INFO] Inside context manger [DEBUG] Finish context manager

and_reraise(*allowed_exceptions*, *msg='Logging error and reraising'*, *level=40*, *exc_info=True*, *stacklevel=3*)
Context manager that logs and reraises given error.

Parameters

- **allowed_exceptions** – Exception(s) to log before reraising.
- **msg** – Message logged for exceptions.
- **level** – Logging level for logging exceptions.
- **exc_info** – If True, include exception info.
- **stacklevel** – Stacklevel of logging statement. Defaults to level 3 since this method (level=2) defers functionality to a helper utility (level=1), but logging should use the context where this is called (level=3).

and_suppress(*allowed_exceptions*, *msg='Suppressed error and logging'*, *level=40*, *exc_info=True*, *stacklevel=3*)
Context manager that logs and suppresses given error.

Parameters

- **allowed_exceptions** – Exception(s) to log and suppress.
- **msg** – Message logged for exceptions.
- **level** – Logging level for logging exceptions.
- **exc_info** – If True, include exception info.
- **stacklevel** – Stacklevel of logging statement. Defaults to level 3 since this method (level=2) defers functionality to a helper utility (level=1), but logging should use the context where this is called (level=3).

CHAPTER

FIVE

CONTRIBUTING

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

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/tonysyu/logquacious/issues.

If you are 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.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

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

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/tonysyu/logquacious/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 :)

5.2 Get Started!

Ready to contribute? Here's how to set up *logquacious* for local development.

1. Fork the *logquacious* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/logquacious.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv logquacious
$ cd logquacious/
$ pip install .[dev]
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass the tests:

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

6. 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
```

7. Submit a pull request through the GitHub website.

5.2.1 Building the documentation

As mentioned above, it's always helpful to have more documentation. To build the docs you can simply run:

```
$ make docs
```

It's also good practice to run the test suite for the docs:

```
$ python test_docs.py
```

Currently, this only runs tests of the ReadMe file. Also, these tests aren't currently integrated with the main test suite because it relies on overriding the output stream of the logger to function.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/tonysyu/logquacious/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ py.test tests.test_logquacious
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
$ make release
$ make clean-build
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

SIX

CREDITS

6.1 Development Lead

- Tony S Yu <tsyu80@gmail.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER

SEVEN

HISTORY

7.1 0.5.0 (2019-05-05)

- Backport *stacklevel* keyword argument from Python 3.8 and configure stacklevel such that logging utilities report the context (e.g. filename and line number) where *logquacious* utilities are called.

7.2 0.4.0 (2018-10-05)

- Fix config override behavior to extend rather than replace default templates

7.3 0.3.0 (2018-10-05)

- Add decorator support for *log.and_suppress* and *log.and_reraise* context managers
- Suppress logging for null/empty log message templates

7.4 0.2.0 (2018-10-03)

Changed default templates. In 0.1.0, the templates were:

```
DEFAULT_TEMPLATES = {
    'start': 'Start {label}',
    'finish': 'Finish {label}',
}
```

These defaults have been changed to:

```
DEFAULT_TEMPLATES = {
    'start': 'Enter {label}',
    'finish': 'Exit {label}',
    'function.start': 'Call `{label}({arguments})`',
    'function.finish': 'Return from `{label}`',
}
```

7.5 0.1.0 (2018-10-03)

- First release on PyPI.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX

T

W