alogging Documentation

Release 0.6.2

Adrian Likins

May 26, 2021

Contents

Contents:

CHAPTER 1

alogging

Python logging tools and utils.

1.1 Usage

To use alogging in a project:

```
import alogging
```

1.2 Examples

Basic use of alogging:

```
import alogging

# create a logging.Logger object, will use the __name__ of the
# module by default. Equilivent to 'log = logging.getLogger(__name__)'
log = alogging.get_logger()

log.debug('created a Logger object so use it for a debug msg')

if __name__ = '__main__':
    main_log = alogging.app_setup(name='example.main')
    main_log.debug('started main')
```

More advanced:

```
import alogging

# local alias for alogging.a()
a = alogging.a

log = alogging.get_logger()

class ThingToDo(object):
    def __init__(self, requirement, priority=None, assigner=None):
        # get a Logger named 'example.ThingToDo'
        self.log = alogging.get_class_logger(self)

        self.log.info('Task as assigned: req=%s, pri=%s, ass=%s', requirement,␣
↪priority, assigner)

        priority = priority or 'never'

        self.log.info('Task reprioritized: req=%s, pri=%s, ass=%s', requirement,␣
↪priority, assigner')


# alogging.t decorator will log when the decorated method is called,
# what args it was passed, and what it's return value was

@alogging.t
def space_out_for_while(duration=None):
    # space out for 10 minutes by default
    duration = duration or 600

    # return the total amount of work accomplished
    return 0

def find_coffee(coffee_places):
    log.debug('looking for coffee')
    return None

def do_startup_stuff():
    coffee_places = ['piehole', 'mug_on_desk', 'coffee_machine', 'krankies']
    # log the the args to find_coffee as it is called
    has_coffee = a(find_coffee(coffee_places))

    work_accomplished = space_out_for_while(duration=300)

def do_work():
    next_task = TaskToDo('finish TODO list', assigner='Lumberg')
    if not next_task:
        return

    # oh no, work...
    log.error("I'm slammed at the moment, I can't do %s', next_task)
    raise Exception()

if __name__ = '__main__':
    # use some reasonable defaults for setting up logging.
    # - log to stderr
    # - use a default format:
```

(continues on next page)

(continued from previous page)

```
    #   """%(asctime)s,%(msecs)03d %(levelname)-0.1s %(name)s %(processName)s:
↪%(process)d %(funcName)s:%(lineno)d - %(message)s"""
    main_log = alogging.app_setup(name='example.main')
    main_log.debug('Log to logging "example.main"')

    do_startup_stuff()

    try:
        do_work()
    except Exception as exc:
        # gruntle a bit and continue
        log.exception(exc)

    return 0
```

1.3 License

- Free software: MIT license

1.4 Features

- TODO

1.5 Authors

- Adrian Likins

CHAPTER 2

Installation

2.1 Stable release

To install alogging, run this command in your terminal:

```
$ pip install alogging
```

This is the preferred method to install alogging, 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 alogging can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/alikins/alogging
```

Or download the tarball:

```
$ curl  -OL https://github.com/alikins/alogging/tarball/master
```

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

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

CHAPTER 3

Usage

To use alogging in a project:

```
import alogging
```

3.1 Examples

Basic use of alogging:

```
import alogging

# local alias for alogging.a()
a = alogging.a

# create a logging.Logger object, will use the __name__ of the
# module by default. Equilivent to 'log = logging.getLogger(__name__)'
log = alogging.get_logger()

log.debug('created a Logger object so use it for a debug msg')


class ThingToDo(object):
    def __init__(self, requirement, priority=None, assigner=None):
        # get a Logger named 'example.ThingToDo'
        self.log = alogging.get_class_logger(self)

        self.log.info('Task as assigned: req=%s, pri=%s, ass=%s', requirement,␣
↪priority, assigner)

        priority = priority or 'never'

        self.log.info('Task reprioritized: req=%s, pri=%s, ass=%s', requirement,␣
↪priority, assigner')
```

(continues on next page)

(continued from previous page)

```
# alogging.t decorator will log when the decorated method is called,
# what args it was passed, and what it's return value was

@alogging.t
def space_out_for_while(duration=None):
    # space out for 10 minutes by default
    duration = duration or 600

    # return the total amount of work accomplished
    return 0

def find_coffee(coffee_places):
    log.debug('looking for coffee')
    return None

def do_startup_stuff():
    coffee_places = ['piehole', 'mug_on_desk', 'coffee_machine', 'krankies']
    # log the the args to find_coffee as it is called
    has_coffee = a(find_coffee(coffee_places))

    work_accomplished = space_out_for_while(duration=300)

def do_work():
    next_task = TaskToDo('finish TODO list', assigner='Lumberg')
    if not next_task:
        return

    # oh no, work...
    log.error("I'm slammed at the moment, I can't do %s', next_task)
    raise Exception()

if __name__ = '__main__':
    # use some reasonable defaults for setting up logging.
    # - log to stderr
    # - use a default format:
    #   """%(asctime)s,%(msecs)03d %(levelname)-0.1s %(name)s %(processName)s:
↪%(process)d %(funcName)s:%(lineno)d - %(message)s"""
    main_log = alogging.app_setup(name='example.main')
    main_log.debug('Log to logging "example.main"')

    do_startup_stuff()

    try:
        do_work()
    except Exception as exc:
        # gruntle a bit and continue
        log.exception(exc)
```

CHAPTER 4

alogging

4.1 alogging package

alogging.**pf**(*obj*)

alogging.**pp**(*obj*)

alogging.**echo**(*value*)

alogging.**a**(**args*)
Log the args of 'a' and returns the args.

Basically, log info about whatever it wraps, but returns it so it can continue to be callled.

Parameters **args** (*tuple*) – The args to pass through to whatever is wrapped

Returns: (tuple): The args that were passed in.

alogging.**t**(*func*)
Decorate a callable (class or method) and log it's args and return values

The loggers created and used should reflect where the object is defined/used.

ie, 'mycode.utils.math.Summer.total' for calling 'total' method on an instance of mycode.utils.math.Summer

alogging.**app_setup**(*name=None*)
Call this to setup a default logging setup in a script or apps __main__

This will create a root logger with some default handlers, as well as a logger for 'name' if provided.

Parameters **name** – (str): If provided, create a logging.Logger with this name

alogging.**module_setup**(*name=None*, *use_root_logger=False*)
Call this to setup a default log setup from a library or module.

ie, where the app itself may be setting up handlers, root logger, etc

alogging.**setup**(*name=None*, *stream_handler=None*, *file_handler=None*, *use_root_logger=False*)

alogging.**setup_root_logger**(*root_level=None*, *handlers=None*)

alogging.**get_class_logger**(*obj*, *depth=2*)
Use to get a logger with name equiv to module.Class

in a regular class __init__, use like:

self.log = alogging.get_class_logger(self)

In a metaclass __new__, use like:

log = alogging.get_class_logger(cls)

alogging.**get_class_logger_name**(*obj*, *depth=None*)
Use to get a logger name equiv to module.Class

alogging.**get_logger**(*name=None*, *depth=2*)
Use to get a logger with name of callers __name__

Can be used in place of:

import logging log = logging.getLogger(__name__)

That can be replaced with

import alogging log = alogging.get_logger()

Parameters

- **name** (*str*) – Optional logger name to use to override the default one chosen automatically.
- **depth** (*int*) – Optional depth of stack to influence where get_logger looks to automatically choose a logger name. Default is 2.

Returns A logger

Return type logging.Logger

alogging.**get_logger_name**(*depth=None*)

alogging.**get_method_logger**(*depth=2*)

alogging.**get_method_logger_name**(*depth=None*)

alogging.**get_stack_size**()
Get stack size for caller's frame.

alogging.**env_log_level**(*var_name*)

4.1.1 Subpackages

alogging.filters package

Submodules

alogging.filters.default_fields module

class alogging.filters.default_fields.**DefaultFieldsFilter**(*name=''*, *defaults=None*)
Bases: logging.Filter

Make sure log records have a default value for the provided field/attribute

ie, if you want to use a default format string with a 'request_id' or 'sql' attribute, but not all records get those attributes added, then you could add this filter to add them

filter(*record*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

alogging.filters.django_sql_celery module

class alogging.filters.django_sql_celery.**DjangoDbSqlCeleryFilter**
Bases: object

Filter to prevent logging celery periodtasks

filter(*record*)

alogging.filters.django_sql_excludes module

class alogging.filters.django_sql_excludes.**DjangoDbSqlExcludeFilter**(*name=''*, *excludes=None*)
Bases: logging.Filter

Filter to prevent logging misc queries

filter(*record*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

alogging.filters.django_sql_slow_queries module

class alogging.filters.django_sql_slow_queries.**DjangoDbSqlSlowQueriesFilter**(*name=''*, *min_duration=0.04*)
Bases: logging.Filter

Filter to log only "slow" sql queries

Default is to only show queries that take more than 40ms

The min_duration init arg is in seconds. Default is 0.04s (40ms)

Add this filter to handlers that get log records from 'django.db' loggers.

See django_sql_slow_queries_example.yaml for yaml setup.

filter(*record*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

alogging.filters.exclude module

class alogging.filters.exclude.**ExcludeFilter**(*name=''*, *excludes=None*, *operator=None*)
Bases: logging.Filter

Filter records with user provided values for record fields

ie, to exclude log records from loop polling records from the 'asyncio' module, with name='asyncio', module='base_events', func_name='_run_once'

excludes is a list of tuples (field_name, value)

check_value(*field_name*, *value*, *record*)

filter(*record*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

alogging.filters.pprint module

class alogging.filters.pprint.**PprintArgsFilter**(*name=''*, *defaults=None*)
Bases: logging.Filter

Use pprint/pformat to pretty the log message args

ie, log.debug("foo: %s", foo) this will pformat the value of the foo object.

filter(*record*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

alogging.filters.process_context module

class alogging.filters.process_context.**CurrentProcess**(*args=None*)
Bases: object

Info about current process.

logging.LogRecords include 'processName', but it is also fairly bogus (ie, 'MainProcess').

So check sys.argv for a better name. Also get current user.

class alogging.filters.process_context.**ProcessContextLoggingFilter**(*name=''*)
Bases: object

Filter that adds cmd_name, cmd_line, and user to log records

cmd_name is the basename of the executable running ('myscript.py') as opposed to log record field 'processName' which is typically something like 'MainProcess'.

cmd_line is the full cmdline. ie, sys.argv joined to a string,

user is the user the process is running as.

filter(*record*)

alogging.formatters package

Submodules

alogging.formatters.django_sql module

class alogging.formatters.django_sql.**DjangoDbSqlPlainFormatter**(*fmt=None*, *datefmt=None*, *options=None*, *style='%'*)

Bases: logging.Formatter

pretty print django.db sql

format(*record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

alogging.formatters.django_sql_color module

class alogging.formatters.django_sql_color.**DjangoDbSqlColorFormatter**(*fmt=None*, *datefmt=None*, *style='%'*, *options=None*, *pygments_lexer='postgres-console'*, *pygments_formatter='terminal256'* *pygments_style='default'*)

Bases: logging.Formatter

Pretty print django.db sql with color by pyments

Parameters

- **fmt** – (str): The logging.Formatter format string
- **datefmt** (*str*) – The logging.Formatter date format string
- **style** (*str*) – The logging.Formatter format string type
- **options** (*dict*) – Dict of options to pass to sqlparse.format()
- **pygments_lexer** (*str*) – The name of the pygments lexer to use. Examples include: 'postgres-console', 'postgres', 'rql', 'sql', 'sqlite3', 'mysql', 'plpgsql', 'tsql'
- **pygments_formatter** (*str*) – The name of the pygments formatter to use. Examples include: 'terminal256', 'terminal', 'terminal16m', 'text'
- **pygments_style** (*str*) – The name of the pygments formatter style to use.

format(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

alogging.formatters.pprint module

class alogging.formatters.pprint.**PPrintRecordFormatter**(*fmt=None*, *datefmt=None*, *indent=1*, *style='%'*)

Bases: logging.Formatter

Pretty print the __dict__ of the log record.

format(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

alogging.record_factories package

Submodules

alogging.record_factories.apply_filters module

class alogging.record_factories.apply_filters.**ApplyFiltersRecordFactory**(**args*, *filters=None*, *base_factory=None*, ***kwargs*)

Bases: object

Apply each of the log record filter instances in *filters* in order on every log record created

Using logging.setLogRecordFactory(ApplyFiltersRecordFactory(filters=[...list of filter instances])) is equilivent to adding the set of filters to every logger instance

alogging.record_factories.apply_filters.**default_filters_factory**(**args*, ***kwargs*)

4.1.2 Submodules

alogging.echo module

alogging.echo.**echo**(*value*)

alogging.echo.**echo_format**(*value*, *depth=1*, *caller_name='echo_format'*)

alogging.logger module

alogging.logger.**a**(**args*)
Log the args of 'a' and returns the args.

Basically, log info about whatever it wraps, but returns it so it can continue to be callled.

Parameters **args** (*tuple*) – The args to pass through to whatever is wrapped

Returns: (tuple): The args that were passed in.

alogging.logger.**app_setup**(*name=None*)
Call this to setup a default logging setup in a script or apps __main__

This will create a root logger with some default handlers, as well as a logger for 'name' if provided.

Parameters **name** – (str): If provided, create a logging.Logger with this name

alogging.logger.**env_log_level**(*var_name*)

alogging.logger.**env_var**(*var_name*)
Fetch the env var by name

alogging.logger.**get_class_logger**(*obj*, *depth=2*)
Use to get a logger with name equiv to module.Class

in a regular class __init__, use like:

self.log = alogging.get_class_logger(self)

In a metaclass __new__, use like:

log = alogging.get_class_logger(cls)

alogging.logger.**get_class_logger_name**(*obj*, *depth=None*)
Use to get a logger name equiv to module.Class

alogging.logger.**get_file_handler**(*name*)

alogging.logger.**get_logger**(*name=None*, *depth=2*)
Use to get a logger with name of callers __name__

Can be used in place of:

import logging log = logging.getLogger(__name__)

That can be replaced with

import alogging log = alogging.get_logger()

Parameters

- **name** (*str*) – Optional logger name to use to override the default one chosen automatically.
- **depth** (*int*) – Optional depth of stack to influence where get_logger looks to automatically choose a logger name. Default is 2.

Returns A logger

Return type logging.Logger

alogging.logger.**get_logger_name**(*depth=None*)

alogging.logger.**get_method_logger**(*depth=2*)

alogging.logger.**get_method_logger_name**(*depth=None*)

alogging.logger.**get_stack_size**()
Get stack size for caller's frame.

alogging.logger.**get_stream_handler**(*name=None*)

alogging.logger.**module_setup**(*name=None*, *use_root_logger=False*)
Call this to setup a default log setup from a library or module.

ie, where the app itself may be setting up handlers, root logger, etc

alogging.logger.**setup**(*name=None*, *stream_handler=None*, *file_handler=None*, *use_root_logger=False*)

alogging.logger.**setup_root_logger**(*root_level=None*, *handlers=None*)

alogging.logger.**t**(*func*)
Decorate a callable (class or method) and log it's args and return values

The loggers created and used should reflect where the object is defined/used.

ie, 'mycode.utils.math.Summer.total' for calling 'total' method on an instance of mycode.utils.math.Summer

alogging.pp module

alogging.pp.**pf**(*obj*)

alogging.pp.**pp**(*obj*)

CHAPTER 5

Credits

5.1 Development Lead

- Adrian Likins <adrian@likins.com>

CHAPTER 6

History

6.1 0.4.3 (2020-06-23)

- use ‘color_bucket_logger’ if available for default setup

6.2 0.4.3 (2020-05-28)

- Docs improvements
- Setup readthedocs

6.3 0.4.2 (2020-04-25)

- Add pygments options to django_sql_color formatter
- Minor docs improvements

6.4 0.4.1 (2020-04-24)

- Split ‘default_setup’ to ‘app_setup’ and ‘module_setup’
- Add docs and examples

6.5 0.3.1 (2019-10-13)

- Add django_sql_color formatter

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

G

M

P

S

T