Twiggy Documentation

Release 0.4.7

Peter Fein

Sep 06, 2018

Contents

CHAPTER 1

Who, What, When, Where

Twiggy is a more Pythonic logger. It aims to be easy to setup:

```
>>> from twiggy import quick_setup
>>> quick_setup()
```

And fun to use!

```
>>> from twiggy import log
>>> log.name('frank').fields(number=42).info("hello {who}, it's a {0} day", 'sunny',␣
↪who='world')
INFO:frank:number=42|hello world, it's a sunny day
```


author Peter Fein

email pete@wearpants.org

twitter @petecode

homepage http://twiggy.readthedocs.io/en/latest/

hosting https://github.com/wearpants/twiggy

IRC irc://irc.freenode.net/#twiggy

license BSD

Python 2.6, 2.7

Twiggy was born at Pycon 2010 after I whined about the standard library's logging and Jesse Noller "invited" me to do something about it.

Install straight with distutils from the Cheeseshop or:

```
pip install Twiggy

easy_install -U Twiggy
```

Get the latest version:

```
git clone https://github.com/wearpants/twiggy.git
```

CHAPTER 2

Why Twiggy Should Be Your New Logger

You should use Twiggy because it is awesome. For more information, see this blog post.

Warning: Twiggy works great, but is not rock solid (yet); do not use for nuclear power plants, spaceships or mortgage derivatives trading (not that it'd matter).

CHAPTER 3

Documentation

3.1 Logging Messages

This part describes how user code can log messages with twiggy.

To get started quickly, use *quick_setup()*.:

```
>>> import twiggy
>>> twiggy.quick_setup()
```

See also:

Full details on *Configuring Output*.

3.1.1 The Magic log

The main interface is the the magic *log*.

```
>>> from twiggy import log
>>> log
<twiggy.logger.Logger object at 0x...>
```

It works out of the box, using typical *levels*. Arbitrary levels are *not* supported. Note that when logging, you never need to refer to any level object; just use the methods on the log.

```
>>> log.debug('You may not care')
DEBUG|You may not care
>>> log.error('OMFG! Pants on fire!')
ERROR|OMFG! Pants on fire!
```

The log can handle messages in several styles of *format strings*, defaulting to new-style.

```
>>> log.info('I wear {0} on my {where}', 'pants', where='legs')
INFO|I wear pants on my legs
```

You can name your loggers.

```
>>> mylog = log.name('alfredo')
>>> mylog.debug('hello')
DEBUG:alfredo|hello
```

3.1.2 Better output

Twiggy's default output strives to be user-friendly and to avoid pet peeves.

Newlines are suppressed by default; that can be turned off per-message.

```
>>> log.info('user\ninput\nannoys\nus')
INFO|user\ninput\nannoys\nus
>>> log.options(suppress_newlines=False).info('we\ndeal')
INFO|we
deal
```

Exceptions are prefixed by `TRACE`. By default, *`tracing`* will use the current exception, but you can also pass an exc_info tuple.

```
>>> try:
...     1/0
... except:
...     log.trace('error').warning('oh noes')
WARNING|oh noes
TRACE Traceback (most recent call last):
TRACE   File "<doctest better-output[...]>", line 2, in <module>
TRACE     1/0
TRACE ZeroDivisionError: division by zero
```

See also:

How to fold exceptions to a single line

3.1.3 Structured Logging

I like this method chaining style a lot.

```
>>> log.name('benito').info('hi there')
INFO:benito|hi there
```

It makes *structured logging* easy. In the past, fielded data was stuffed in the text of your message:

```
>>> log.info('Going for a walk. path: {0} roads: {1}', "less traveled", 42)
INFO|Going for a walk. path: less traveled roads: 42
```

Instead, you can use *`fields()`* to add arbitrary key-value pairs. Output is easily parseable.

```
>>> log.fields(path="less traveled", roads=42).info('Going for a walk')
INFO:path=less traveled:roads=42|Going for a walk
```

The *`struct()`* is a short cut for *only* logging fields. This is great for runtime statistics gathering.

```
>>> log.struct(paths=42, dolphins='thankful')
INFO:dolphins=thankful:paths=42|
```

3.1.4 Partial Binding

Each call to *fields()* or *options()* creates a new, independent log instance that inherits all of the data of the parent. This incremental binding can be useful for webapps.

```
>>> ## an application-level log
... webapp_log = log.name("myblog")
>>> ## a log for the individual request
... current_request_log = webapp_log.fields(request_id='12345')
>>> current_request_log.fields(rows=100, user='frank').info('frobnicating database')
INFO:myblog:request_id=12345:rows=100:user=frank|frobnicating database
>>> current_request_log.fields(bytes=5678).info('sending page over tubes')
INFO:myblog:bytes=5678:request_id=12345|sending page over tubes
>>> ## a log for a different request
... another_log = webapp_log.fields(request_id='67890')
>>> another_log.debug('Client connected')
DEBUG:myblog:request_id=67890|Client connected
```

Chained style is awesome. It allows you to create complex yet parsable log messages in a concise way.

```
>>> log.name('donjuan').fields(pants='sexy').info("hello, {who} want to {what}?", who=
↪'ladies', what='dance')
INFO:donjuan:pants=sexy|hello, ladies want to dance?
```

3.1.5 Sample Output

Routed to a *file*, the above produces the following:

```
2010-03-28T14:23:34Z:DEBUG:You may not care
2010-03-28T14:23:34Z:ERROR:OMFG! Pants on fire!
2010-03-28T14:23:34Z:INFO:I like bikes
2010-03-28T14:23:34Z:INFO:I wear pants on my legs
2010-03-28T14:23:34Z:DEBUG:alfredo:hello
2010-03-28T14:23:34Z:INFO:user\ninput\nannoys\nus
2010-03-28T14:23:34Z:INFO:we
deal
2010-03-28T14:23:34Z:WARNING:oh noes
TRACE Traceback (most recent call last):
TRACE   File "<doctest better-output[...]>", line 35, in <module>
TRACE ZeroDivisionError: integer division or modulo by zero
2010-03-28T14:23:34Z:INFO:benito:hi there
2010-03-28T14:23:34Z:INFO:Going for a walk. path: less traveled roads: 42
2010-03-28T14:23:34Z:INFO:path=less traveled:roads=42:Going for a walk
2010-03-28T14:23:34Z:INFO:dolphins=thankful:paths=42:
2010-03-28T14:23:34Z:INFO:myblog:request_id=12345:rows=100:user=frank:frobnicating␣
↪database
2010-03-28T14:23:34Z:INFO:myblog:bytes=5678:request_id=12345:sending page over tubes
2010-03-28T14:23:34Z:INFO:myblog:request_id=67890:Client connected
2010-03-28T14:23:34Z:INFO:donjuan:pants=sexy:hello, ladies want to dance?
2010-03-28T14:23:34Z:INFO:myblog:request_id=12345:rows=100:user=frank:frobnicating␣
↪database
2010-03-28T14:23:34Z:INFO:myblog:bytes=5678:request_id=12345:sending page over tubes
2010-03-28T14:23:34Z:DEBUG:myblog:request_id=67890:Client connected
```

3.2 Configuring Output

This part discusses how to configure twiggy's output of messages. You should do this once, near the start of your application's `__main__`. It's particularly important to set up Twiggy *before spawning new processes*.

3.2.1 Quick Setup

quick_setup() quickly configures output with reasonable defaults. Use it when you don't need a lot of customizability or as the default configuration that the user can override via programatic configuration or *dict_config()*.

The defaults will emit log messages of `DEBUG` level or higher to `stderr`:

```
from twiggy import quick_setup
quick_setup()
```

See also:

The API docs for complete information on *quick_setup()*'s parameters.

3.2.2 twiggy_setup.py

Twiggy's output side features modern, loosely coupled design. The easiest way to understand what that means is to look at how to configure twiggy programmatically.

Note: Prior to Twiggy 0.5, by convention twiggy was programmatically set up in a separate file in your application called `twiggy_setup.py` in a function called `twiggy_setup()`. This allowed sites to override the configuration via their configuration management systems by replacing the file. In Twiggy 0.5 and later, the *dict_config()* function provides a more natural way for to allow users to override the logging configuration using a config file.

Programmatically configuring Twiggy involves creating an *output* which defines where the log messages will be sent and then creating an *Emitter* which associates a subset of your application's logs with the output. Here's what an example `twiggy_setup()` function would look like:

```
from twiggy import add_emitters, outputs, levels, filters, formats, emitters # import␣
↪* is also ok
def twiggy_setup():
    alice_output = outputs.FileOutput("alice.log", format=formats.line_format)
    bob_output = outputs.FileOutput("bob.log", format=formats.line_format)

    add_emitters(
        # (name, min_level, filter, output),
        ("alice", levels.DEBUG, None, alice_output),
        ("betty", levels.INFO, filters.names("betty"), bob_output),
        ("brian.*", levels.DEBUG, filters.glob_names("brian.*"), bob_output),
        )

# near the top of your __main__
twiggy_setup()
```

In this example, we create two log *Outputs*: `alice_output` and `bob_output`. These outputs are `twiggy.outputs.FileOutput`s. They tell twiggy to write messages directed to the output into the named file, in this case, ``alice.log` and `bob.log`. All outputs have a formatter associated with them. The formatter is responsible for turning Twiggy's *Structured Logging* calls into

a suitable form for the output. In this example, both `alice_output` and `bob_output` use *`twiggy.formats.line_format()`* to format their messages.

`Emitters` associate *Outputs* with a set of messages via *`levels`* and *Filtering Output*. Here we configure three emitters to two outputs. `alice_output` will receive all messages and `bob_output` will receive two sets of messages:

- messages with the name field equal to `betty` and level >= `INFO`
- messages with the name field glob-matching `brian.*`

The convenience function, *`add_emitters()`*, takes the emitter information as a tuple of emitter name, minimum log level, optional filters, and the output that the logs should be written to. It creates the `Emitters` from that information and populates the *`emitters`* dictionary:

```
>>> sorted(emitters.keys())
['alice', 'betty', 'brian.*']
```

`Emitters` can be removed by deleting them from this dict. *`filter`* and *`min_level`* may be modified during the running of the application, but outputs *cannot* be changed. Instead, remove the emitter and re-add it.

```
>>> # bump level
... emitters['alice'].min_level = levels.WARNING
>>> # change filter
... emitters['alice'].filter = filters.names('alice', 'andy')
>>> # remove entirely
... del emitters['alice']
```

We'll examine the various parts in more detail below.

Note: Remember to import and run `twiggy_setup` near the top of your application.

3.2.3 Outputs

Outputs are the destinations to which log messages are written (files, databases, etc.). Several *`implementations`* are provided. Once created, outputs cannot be modified. Each output has an associated *`format`*.

Asynchronous Logging

Many outputs can be configured to use a separate, dedicated process to log messages. This is known as *asynchronous logging* and is enabled with the *`msg_buffer`* argument. Asynchronous mode dramatically reduces the cost of logging, as expensive formatting and writing operations are moved out of the main thread of control.

3.2.4 Formats

`Formats` transform a log message into a form that can be written by an output. The result of formatting is output dependent; for example, an output that posts to an HTTP server may take a format that provides JSON, whereas an output that writes to a file may produce text.

Line-oriented formatting

LineFormat formats messages for text-oriented outputs such as a file or standard error. It uses a *ConversionTable* to stringify the arbitrary fields in a message. To customize, copy the default *line_format* and modify:

```
# in your twiggy_setup
import copy
my_format = copy.copy(formats.line_format)
my_format.conversion.add(key = 'address', # name of the field
                         convert_value = hex, # gets original value
                         convert_item = "{0}={1}".format, # gets called with: key,␣
↪converted_value
                         required = True)

# output messages with name 'memory' to stderr
add_emitters(('memory', levels.DEBUG, filters.names('memory'), outputs.
↪StreamOutput(format = my_format)))
```

3.2.5 Filtering Output

The messages output by an emitter are determined by its *min_level* and filter (a *function* which take a *Message* and returns bool). These attributes may be changed while the application is running. The *filter* attribute of emitters is *intelligent*; you may assign strings, bools or functions and it will magically do the right thing. Assigning a list indicates that *all* of the filters must pass for the message to be output.

```
e = emitters['memory']
e.min_level = levels.WARNING
# True allows all messages through (None works as well)
e.filter = True
# False blocks all messages
e.filter = False
# Strings are interpreted as regexes (regex objects ok too)
e.filter = "^mem.*y$"
# functions are passed the message; return True to emit
e.filter = lambda msg: msg.fields['address'] > 0xDECAF
# lists are all()'d
e.filter = ["^mem.y$", lambda msg: msg.fields['address'] > 0xDECAF]
```

See also:

Available *filters*

3.2.6 dict_config()

Twiggy 0.5 features a new convenience method, *dict_config()* for configuring *Emitters* that takes a a dictionary with the configuration information. The dictionary can be constructed programmatically, loaded from a configuration file, or hardcoded into an application. This allows the programmer to easily set defaults and allow the user to override those from a configuration file. Here's an example:

```
from twiggy import dict_config

twiggy_config = {'version': '1.0',
                 'outputs': {
```

(continues on next page)

(continued from previous page)

```
                    'alice_output': {
                        'output': 'twiggy.outputs.FileOutput',
                        'args': ['alice.log']
                    },
                    'bob_output': {
                        'output': 'twiggy.outputs.FileOutput',
                        'args': ['bob.log'],
                        'format': 'twiggy.formats.line_format'
                    }
                },
                'emitters': {
                    'alice': {
                        'level': 'DEBUG',
                        'output_name': 'alice_output'
                    },
                    'betty': {
                        'level': 'INFO',
                        'filters': [ {
                            'filter': 'twiggy.filters.names',
                            'args': ['betty']
                            }
                        ],
                        'output_name': 'bob_output'
                    },
                    'brian.*': {
                        'level': 'DEBUG',
                        'filters': [ {
                            'filter': 'twiggy.filters.glob_names',
                            'args': ['brian.*']
                            }
                        ],
                        'output_name': 'bob_output'
                    }
                }
            }

dict_config(twiggy_config)
```

In this example, the programmer creates a twiggy configuration in the application's code and uses it to configure twiggy. The configuration closely mirrors the objects that were created in the *twiggy_setup.py* section. The `outputs` field contains definitions of `alice_output` and `bob_output` that write to the `alice.log` and `bob.log` files respectively. The `emitters` field defines three emitters, their levels and filters to output to the

The configuration should be done near the start of your application. It's particularly important to set up Twiggy *before spawning new processes*.

With this configuration, *`twiggy.dict_config()`* will create two log destinations (*Outputs*): `alice.log` and `bob.log`. These *Outputs* are then associated with the set of messages that they will receive in the `emitters` section. `alice.log` will receive all messages and `bob.log` will receive two sets of messages:

- messages with the name field equal to `betty` and level >= `INFO`
- messages with the name field glob-matching `brian.*`

See the *Twiggy Config Schema* documentation for details of what each of the fields in the configuration dictionary mean.

User Overrides

Each site that runs an application is likely to have different logging needs. Using *dict_config* it is easy to let the user override the configuration specified by the program. For instance, the application could have a yaml configuration file with a `logging_config` section:

```
import yaml
config = yaml.safe_load('config_file.yml')
if 'logging_config' in config:
    try:
        twiggy.dict_config(config['logging_config'])
    except Exception as e:
        print('User provided logging configuration was flawed: {0}'.format(e))
```

Twiggy Config Schema

The dict taken by *twiggy.dict_config()* may contain the following keys:

version Set to the value representing the schema version as a string. Currently, the only valid value is "1.0".

incremental (*Optional*) If True, the dictionary will update any existing configuration. If False, this will override any existing configuration. This allows user defined logging configuration to decide whether to override the logging configuration set be the application or merely supplement it. The default is False.

outputs (*Optional*) Mapping of output names to outputs. Outputs consist of

output A *twiggy.outputs.Output* or the string representation with which to import a *Output*. For instance, to use the builtin, *twiggy.outputs.FileOutput* either set output directly to the class or the string `twiggy.outputs.FileOutput`.

args (*Optional*) A list of arguments to pass to the `Twiggy.outputs.Output` class constructor. For instance, *FileOutput* takes the filename of a file to log to. So `args` could be set to: `["logfile.log"]`.

kwargs (*Optional*) A dict of keyword arguments to pass to the `Twiggy.outputs.Output` class constructor. For instance, *StreamOutput* takes a stream as a keyword argument so `kwargs` could be set to: `{"stream": "ext://sys.stdout"}`.

format (*Optional*) A formatter function which transforms the log message for the output. This can either be a string name of the formatter of the formatter itself. The default is *twiggy.formats.line_format()*

If both `outputs` and `emitters` are None and `incremental` is False then *twiggy.emitters* will be cleared.

emitters (*Optional*) Mapping of emitter names to emitters. Emitters consist of:

level String name of the log level at which log messages will be passed to this emitter. May be one of (In order of severity) `CRITICAL`, `ERROR`, `WARNING`, `NOTICE`, `INFO`, `DEBUG`, `DISABLED`.

output_name The name of an output in this configuration dict.

filters (*Optional*) A list of filters which filter out messages which will go to this emitter. Each filter is a mapping which consists of:

filter Name for a twiggy filter function. This can either be a string name for the function or the function itself.

args (*Optional*) A list of arguments to pass to the `Twiggy.outputs.Output` class constructor. For instance, *FileOutput* takes the filename of a file to log to. So `args` could be set to: `["logfile.log"]`.

kwargs (*Optional*) A dict of keyword arguments to pass to the `Twiggy.outputs.Output` class constructor. For instance, *`StreamOutput`* takes a stream as a keyword argument so `kwargs` could be set to: `{"stream": "ext://sys.stdout"}`.

If both `emitters` and `output` are None and `incremental` is False then *`twiggy.emitters`* will be cleared.

Sometimes you want to have an entry in `args` or `kwargs` that is a python object. For instance, *`StreamOutput`* takes a stream keyword argument so you may want to give `sys.stdout` to it. If you are building the configuration dictionary in Python code you can simply use the actual object. However, if you are writing in a text configuration file, you can specify existing objects by prefixing the string with `ext://`. When Twiggy sees that the string starts with `ext://` it will strip off the prefix and then try to import an object with the rest of the name.

Here's an example config that you might find in a YAML config file:

```
version: '1.0'
outputs:
    alice_output:
        output: 'twiggy.outputs.FileOutput'
        args:
            - 'alice.log'
    bob_output:
        output: 'twiggy.outputs.StreamOutput'
        kwargs:
            stream: 'ext://sys.stdout'
        format: 'twiggy.formats.line_format'
emitters:
    alice:
        level: 'DEBUG'
        output_name: 'alice_output'
    betty:
        level: 'INFO'
        filters:
            filter: 'twiggy.filters.names'
            args:
                - 'betty'
        output_name: 'bob_output'
    brian.*:
        levels: 'DEBUG'
        filters:
            filter: 'twiggy.filters.glob_names'
            args:
                -'brian.*'
        output_name: 'bob_output'
```

3.3 Reference Guide

3.3.1 Dynamic Logging

Any functions in message args/fields are called and the value substitued.

```
>>> import os
>>> from twiggy.lib import thread_name
>>> thread_name()
'MainThread'
```

(continues on next page)

(continued from previous page)

```
>>> log.fields(pid=os.getpid).info("I'm in thread {0}", thread_name)
INFO:pid=...|I'm in thread MainThread
```

This can be useful with partially-bound loggers, which lets us do some cool stuff. Here's a proxy class that logs which thread accesses attributes.

```
class ThreadTracker(object):
    """a proxy that logs attribute access"""
    def __init__(self, obj):
        self.__obj = obj
        # a partially bound logger
        self.__log = log.name("tracker").fields(obj_id=id(obj), thread=thread_name)
        self.__log.debug("started tracking")
    def __getattr__(self, attr):
        self.__log.debug("accessed {0}", attr)
        return getattr(self.__obj, attr)

class Bunch(object):
    pass
```

Let's see it in action.

```
>>> foo = Bunch()
>>> foo.bar = 42
>>> tracked = ThreadTracker(foo)
DEBUG:tracker:obj_id=...:thread=MainThread|started tracking
>>> tracked.bar
DEBUG:tracker:obj_id=...:thread=MainThread|accessed bar
42
>>> import threading
>>> t=threading.Thread(target = lambda: tracked.bar * 2, name = "TheDoubler")
>>> t.start(); t.join()
DEBUG:tracker:obj_id=...:thread=TheDoubler|accessed bar
```

If you really want to log a callable, `repr()` it or wrap it in lambda.

See also:

procinfo feature

3.3.2 Features!

Features are optional additons of logging functionality to the *log*. They encapsulate common logging patterns. Code can be written using a feature, enhancing what information is logged. The feature can be disabled at *runtime* if desired.

> **Warning:** Features are currently deprecated, pending a reimplementation in version 0.5

```
>>> from twiggy.features import socket as socket_feature
>>> log.addFeature(socket_feature.socket)
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.connect(('www.python.org', 80))
```

(continues on next page)

(continued from previous page)

```
>>> log.socket(s).debug("connected")
DEBUG:host=...:ip_addr=...:port=80:service=http|connected
>>> # turn off the feature - the name is still available
... log.disableFeature('socket')
>>> log.socket(s).debug("connected")
DEBUG|connected
>>> # use a different implementation
... log.addFeature(socket_feature.socket_minimal, 'socket')
>>> log.socket(s).debug("connected")
DEBUG:ip_addr=...:port=80|connected
```

3.3.3 Stays Out of Your Way

Twiggy tries to stay out of your way. Specifically, an error in logging should **never** propogate outside the logging subsystem and cause your main application to crash. Instead, errors are trapped and reported by the *internal_log*.

Instances of *InternalLogger* only have a single *Output* - they do not use emitters. By default, these messages are sent to standard error. You may assign an alternate ouput (such as a file) to `twiggy.internal_log.output` if desired, with the following conditions:

- the output should be failsafe - any errors that occur during internal logging will be dumped to standard error, and suppressed, causing the original message to be discarded.
- accordingly, networked or asynchronous outputs are not recommended.
- make sure someone is reading these log messages!

3.3.4 Concurrency

Locking in twiggy is as fine-grained as possible. Each individual output has its own lock (if necessary), and only holds that lock when writing. Using redundant outputs (ie, pointing to the same file) is not supported and will cause logfile corruption.

Asynchronous loggers never lock.

3.3.5 Use by Libraries

Libraries require special care to be polite and usable by application code. The library should have a single bound in its top-level package that's used by modules. Library logging should generally be silent by default.

```
# in mylib/__init__.py
log = twiggy.log.name('mylib')
log.min_level = twiggy.levels.DISABLED

# in mylib/some_module.py
from . import log
log.debug("hi there")
```

This allows application code to enable/disable all of library's logging as needed.

```
# in twiggy_setup
import mylib
mylib.log.min_level = twiggy.levels.INFO
```

In addition to *min_level*, loggers also have a *filter*. This filter operates *only on the format string*, and is intended to allow users to selectively disable individual messages in a poorly-written library.

```
# in mylib:
for i in xrange(1000000):
    log.warning("blah blah {0}", 42)

# in twiggy_setup: turn off stupidness
mylib.log.filter = lambda format_spec: format_spec != "blah blah {0}"
```

Note that using a filter this way is an optimization - in general, application code should use *emitters* instead.

3.3.6 Tips And Tricks

Alternate Styles

In addition to the default new-style (braces) format specs, twiggy also supports old-style (percent, aka printf) and templates (dollar). The aliases {}, % and $ are also supported.

```
>>> log.options(style='percent').info('I like %s', "bikes")
INFO|I like bikes
>>> log.options(style='dollar').info('$what kill', what='Cars')
INFO|Cars kill
```

Use Fields

Use *fields()* to include key-value data in a message instead of embedding it the human-readable string.

```
# do this:
log.fields(key1='a', key2='b').info("stuff happenend")

# not this:
log.info("stuff happened. key1: {0} key2: {1}", 'a', 'b')
```

3.3.7 Technical Details

Independence of logger instances

Each log instance created by partial binding is independent from each other. In particular, a logger's *name()* has no relation to the object; it's just for human use.

```
>>> log.name('bob') is log.name('bob')
False
```

Optimizations

Twiggy has been written to be fast, minimizing the performance impact on the main execution path. In particular, messages that will cause no output are handled as quickly as possible. Users are therefore encouraged to add lots of logging for development/debugging purposes and then turn them off in production.

The emit methods can be hidden behind an appropriate `assert`. Python will eliminate the statement entirely when run with bytecode optimization (`python -O`).

```
assert log.debug("This goes away with python -O") is None
assert not log.debug("So does this")
```

Note: The author doesn't particularly care for code written like this, but likes making his users happy more.

3.3.8 Extending Twiggy

When developing extensions to twiggy, use the *devel_log*. An *InternalLogger*, the devel_log is completely separate from the main *log*. By default, messages logged to the devel_log are discarded; assigning an appropriate *Output* to its `output` attribute before using.

Writing Features

Warning: Features are currently deprecated, pending a reimplementation in version 0.5

Features are used to encapsulate common logging patterns. They are implemented as methods added to the *Logger* class. They receive an instance as the first argument (ie, `self`). *Enable the feature* before using.

Features come in two flavors: those that add information to a message's fields or set options, and those that cause output.

Features which only add fields/set options should simply call the appropriate method on `self` and return the resultant object.:

```
def dimensions(self, shape):
    return self.fields(height=shape.height, width=shape.width)
```

Features can also emit messages as usual. Do not return from these methods.:

```
def sayhi(self, lang):
    if lang == 'en':
        self.info("Hello world")
    elif lang == 'fr':
        self.info("Bonjour tout le monde")
```

If the feature should add fields *and* emit in the same step (like *struct()*), use the *emit()* decorators. Here's a prototype feature that dumps information about a WSGI environ.:

```
from twiggy.logger import emit

@emit.info
def dump_wsgi(self, wsgi_environ):
    keys = ['SERVER_PROTOCOL', 'SERVER_PORT', 'SERVER_NAME', 'CONTENT_LENGTH',
↪'CONTENT_TYPE', 'QUERY_STRING', 'PATH_INFO', 'SCRIPT_NAME', 'REQUEST_METHOD']
    d = {}
    for k in keys:
        d[k] = wsgi_environ.get(k, '')

    for k, v in wsgi_environ.iteritems():
        if k.startswith('HTTP_'):
            k = k[5:].title().replace('_', '-')
```

(continues on next page)

(continued from previous page)

```
            d[k] = v

    # if called on an unnamed logger, add a name
    if name not in self._fields:
        self = self.name('dumpwsgi')

    return self.fields_dict(d)
```

Writing Outputs and Formats

Outputs do the work of writing a message to an external resource (file, socket, etc.). User-defined outputs should inherit from *Output* or *AsyncOutput* if they wish to support *asynchronous logging* (preferred).

An Output subclass's __init__ should take a *format* and any parameters needed to acquire resources (filename, hostname, etc.), but *not the resources themselves*. These are created in *_open()*. Implementations supporting asynchronous logging should also take a *msg_buffer* argument.

Outputs should define the following:

Output.**_open**()
: Acquire any resources needed for writing (files, sockets, etc.)

Output.**_close**()
: Release any resources acquired in *_open*

Output.**_write**(*x*)
: Do the work of writing

  Parameters **x** – an implementation-dependent object to be written.

If the output requires locking to be thread-safe, set the class attribute *use_locks* to True (the default). Turning off may give slightly higher throughput.

The *format* callable is Output-specific; it should take a *Message* and return an appropriate object (string, database row, etc.) to be written. **Do not modify** the received message - it is shared by all outputs.

ConversionTables are particulary useful for formatting fields. They are commonly used with *LineFormat* to format messages for text-oriented output.

```
from twiggy.lib.converter import ConversionTable
conversion = ConversionTable()

fields = {'shape': 'square',
          'height': 10,
          'width': 5,
          'color': 'blue'}

# hide shape field name
# uppercase value
# make mandatory
conversion.add(key = 'shape',
               convert_value = str.upper,
               convert_item = '{1}'.format, # stringify 2nd item (value)
               required = True)

# format height value with two decimal places
# show as "<key> is <value>"
```

(continues on next page)

(continued from previous page)

```
conversion.add('height', '{0:.2f}'.format, "{0} is {1}".format)

# separate fields in final output by colons
conversion.aggregate = ':'.join

# unknown items are sorted by key

# unknown values are stringified
conversion.generic_value = str

# show unknown items as "<key>=<value>"
conversion.generic_item = "{0}={1}".format

# convert!
print(conversion.convert(fields))
```

```
SQUARE:height is 10.00:color=blue:width=5
```

3.4 API Reference

3.4.1 Global Objects

twiggy.**log**
the magic log object

twiggy.**internal_log**
InternalLogger for reporting errors within Twiggy itself

twiggy.**devel_log**
InternalLogger for use by developers writing extensions to Twiggy

twiggy.**emitters**
the global *emitters* dictionary, tied to the *log*

3.4.2 Configuration

twiggy.**add_emitters**(**tuples*)
Add multiple emitters

`tuples` should be `(name_of_emitter, min_level, filter, output)`. The last three are passed to *Emitter*.

twiggy.**dict_config**(*config*)
Configure twiggy logging via a dictionary

Parameters **config** – a dictionary which configures twiggy's outputs and emitters. See `TWIGGY_CONFIG_SCHEMA` for details of the format of the dict.

See also:

dict_config() for a thorough explaination of the outputs and emitters concepts from the dictionary

twiggy.**quick_setup**(*min_level=<LogLevel DEBUG>*, *file=None*, *msg_buffer=0*)
Quickly set up *emitters*.

quick_setup() quickly sets up logging with reasonable defaults and minimal customizablity. Quick setup is limited to sending all messages to a file, `sys.stdout` or `sys.stderr`. A timestamp will be prefixed when logging to a file.

Parameters

- **min_level** (*LogLevel*) – lowest message level to cause output
- **file** (*string*) – filename to log to, or `sys.stdout`, or `sys.stderr`. `None` means standard error.
- **msg_buffer** (*int*) – number of messages to buffer, see `outputs.AsyncOutput.msg_buffer`

3.4.3 Features

Optional additions of logging functionality

procinfo

Logging feature to add information about process, etc.

`twiggy.features.procinfo.`**procinfo**(*self*)

Adds the following fields:

Hostname current hostname

Pid current process id

Thread current thread name

socket

Logging feature to add information about a socket

`twiggy.features.socket.`**socket**(*self*, *s*)

Adds the following fields:

ip_addr numeric IP address

port port number

host peer hostname, as returned by `getnameinfo()`

service the human readable name of the service on `port`

Parameters **s** (*socket*) – the socket to extract information from

`twiggy.features.socket.`**socket_minimal**(*self*, *s*)

Like *socket*, but only log `ip_addr` and `port`

3.4.4 Filters

`twiggy.filters.`**filter**(*msg : Message*) → bool

A *filter* is any function that takes a *Message* and returns True if it should be *emitted*.

twiggy.filters.**msg_filter**(*x*) → filter
create a *filter* intelligently

You may pass:

None, True the filter will always return True

False the filter will always return False

string compiled into a regex

regex `match()` against the message text

callable returned as is

list apply *msg_filter* to each element, and `all()` the results

Return type *filter* function

twiggy.filters.**names**(**names*) → filter
create a *filter*, which gives True if the messsage's name equals any of those provided

`names` will be stored as an attribute on the filter.

Parameters **names** (*strings*) – names to match

Return type *filter* function

twiggy.filters.**glob_names**(**names*) → filter
create a *filter*, which gives True if the messsage's name globs those provided.

`names` will be stored as an attribute on the filter.

This is probably quite a bit slower than *names()*.

Parameters **names** (*strings*) – glob patterns.

Return type *filter* function

class twiggy.filters.**Emitter**
Hold and manage an *Output* and associated *filter()*

min_level
only emit if greater than this *LogLevel*

filter
arbitrary *filter()* on message contents. Assigning to this attribute is *intelligent*.

_output
Output to emit messages to. Do not modify.

3.4.5 Formats

Formats are single-argument callables that take a *Message* and return an object appropriate for the *Output* they are assigned to.

class twiggy.formats.**LineFormat**(*separator=':'*, *traceback_prefix='\nTRACE'*, *conversion=line_conversion*)

separator
string to separate line parts. Defaults to `:`.

traceback_prefix
string to prepend to traceback lines. Defaults to `\nTRACE`.

Set to `'\\n'` (double backslash n) to roll up tracebacks to a single line.

conversion
ConversionTable used to format *fields*. Defaults to *line_conversion*

format_text(*msg*)
format the text part of a message

format_fields(*msg*)
format the fields of a message

format_traceback(*msg*)
format the traceback part of a message

twiggy.formats.**line_conversion**
a default line-oriented *ConversionTable*. Produces a nice-looking string from *fields*.

Fields are separated by a colon (`:`). Resultant string includes:

time in iso8601 format (required)

level message level (required)

name logger name

Remaining fields are sorted alphabetically and formatted as `key=value`

twiggy.formats.**line_format**
a default *LineFormat* for output to a file. *Sample output*.

Fields are formatted using *line_conversion* and separated from the message *text* by a colon (`:`). Traceback lines are prefixed by `TRACE`.

twiggy.formats.**shell_conversion**
a default line-oriented *ConversionTable* for use in the shell. Returns the same string as *line_conversion* but drops the `time` field.

twiggy.formats.**shell_format**
a default *LineFormat* for use in the shell. Same as *line_format* but uses *shell_conversion* for *fields*.

3.4.6 Levels

Levels include (increasing severity): `DEBUG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `CRITICAL`, `DISABLED`

class twiggy.levels.**LogLevel**(*name*, *value*)
A log level. Users should *not* create new instances.

Levels are opaque; they may be compared to each other, but nothing else.

class twiggy.levels.**LogLevelMeta**
Metaclass that aids in making comparisons work the same in Python2 and Python3

Python3 raises TypeError when unorderable types are compared via lt, gt, le, ge. Python2 picks an order but it doesn't always make much sense.

In Python3, we only need the rich comparison operators to get this behaviour.

In Python2, we use the __cmp__ function to raise TypeError for lt, gt, le, and ge. We define __eq__ and __ne__ on their own since those should just say that a LogLevel is never equal to a non-LogLevel.

twiggy.levels.**name2level**(*name*)
return a *LogLevel* from a case-insensitve string

3.4.7 Library

twiggy.lib.**iso8601time**(*gmtime=None*)
convert time to ISO 8601 format - it sucks less!

Parameters **gmtime** (*time.struct_time*) – time tuple. If None, use time.gmtime() (UTC)

XXX timezone is not supported

twiggy.lib.**thread_name**()
return the name of the current thread

Converter

class twiggy.lib.converter.**Converter**(*key*, *convert_value*, *convert_item*, *required=False*)
Holder for *ConversionTable* items

Variables

- **key** – the key to apply the conversion to
- **convert_value** (*function*) – one-argument function to convert the value
- **convert_item** (*function*) – two-argument function converting the key and converted value
- **required** (*bool*) – is the item required to present. Items are optional by default.

twiggy.lib.converter.**same_value**(*v*)
return the value unchanged

twiggy.lib.converter.**same_item**(*k*, *v*)
return the item unchanged

twiggy.lib.converter.**drop**(*k*, *v*)
return None, indicating the item should be dropped

New in version 0.5.0: Add *same_value*, *same_item*, *drop*.

class twiggy.lib.converter.**ConversionTable**(*seq*)
Convert data dictionaries using *Converters*

For each item in the dictionary to be converted:

1. Find one or more corresponding converters `c` by matching key.
2. Build a list of converted items by calling `c.convertItem(item_key, c.convertValue(item_value))`. The list will have items in the same order as converters were supplied.
3. Dict items for which no converter was found are sorted by key and passed to *generic_value* / *generic_item*. These items are appended to the list from step 2.
4. If any required items are missing, `ValueError` is raised.
5. The resulting list of converted items is passed to *aggregate*. The value it returns is the result of the conversion.

Users may override *generic_value*/*generic_item*/*aggregate* by subclassing or assigning a new function on a ConversionTable instance.

Really, it's *pretty intuitive*.

__init__(*seq=None*)

Parameters **seq** – a sequence of Converters

You may also pass 3-or-4 item arg tuples or kwarg dicts (which will be used to create *Converters*)

convert(*d*)
do the conversion

Parameters **d** (*dict*) – the data to convert. Keys should be strings.

generic_value(*value*)
convert values for which no specific *Converter* is supplied

generic_item(*key*, *value*)
convert items for which no specific *Converter* is supplied

aggregate(*converteds*)
aggregate list of converted items. The return value of *convert*

copy()
make an independent copy of this ConversionTable

get(*key*)
return the *first* converter for key

get_all(*key*)
return a list of all converters for key

add(**args*, ***kwargs*)
Append a *Converter*.

`args` & `kwargs` will be passed through to its constructor

delete(*key*)
delete the *all* of the converters for key

3.4.8 Logger

Loggers should not be created directly by users; use the global *log* instead.

class `twiggy.logger.`**BaseLogger**(*fields=None*, *options=None*, *min_level=None*)
Base class for loggers

_fields
dictionary of bound fields for *structured logging*. By default, contains a single field `time` with value `time.gmtime()`. This function will be called for each message emitted, populating the field with the current `time.struct_time`.

_options
dictionary of bound *options*.

min_level
minimum *LogLevel* for which to emit. For optimization purposes only.

fields(***kwargs*) → bound Logger
bind fields for *structured logging*. `kwargs` are interpreted as names/values of fields.

fields_dict (*d*) → bound Logger
bind fields for structured logging. Use this instead of *fields* if you have keys which are not valid Python identifiers.

Parameters **d** (*dict*) – dictionary of fields. Keys should be strings.

options (***kwargs*) → bound Logger
bind *options* for message creation.

trace (*trace='error'*) → bound Logger
convenience method to enable *traceback logging*

name (*name*) → bound Logger
convenvience method to bind `name` field

struct (***kwargs*) → bound Logger
convenience method for *structured logging*. Calls *fields()* and emits at *info*

struct_dict (*d*) → bound Logger
convenience method for *structured logging*. Use instead of *struct* if you have keys which are not valid Python identifiers.

Parameters **d** (*dict*) – dictionary of fields. Keys should be strings.

The following methods cause messages to be emitted. `format_spec` is a template string into which `args` and `kwargs` will be substituted.

debug (*format_spec=''*, **args*, ***kwargs*)
Emit at `DEBUG` level

info (*format_spec=''*, **args*, ***kwargs*)
Emit at `INFO` level

notice (*format_spec=''*, **args*, ***kwargs*)
Emit at `NOTICE` level

warning (*format_spec=''*, **args*, ***kwargs*)
Emit at `WARNING` level

error (*format_spec=''*, **args*, ***kwargs*)
Emit at `ERROR` level

critical (*format_spec=''*, **args*, ***kwargs*)
Emit at `CRITICAL` level

class `twiggy.logger.`**Logger** (*fields=None*, *options=None*, *min_level=None*)
Logger for end-users. The type of the magic *log*

filter
Filter on `format_spec`. For optimization purposes only. Should have the following signature:

func (*format_spec : string*) → bool
Should the message be emitted.

classmethod addFeature (*func*, *name=None*)
add a feature to the class

Parameters

- **func** – the function to add
- **name** (*string*) – the name to add it under. If None, use the function's name.

classmethod disableFeature(*name*)
disable a feature.

A method will still exist by this name, but it won't do anything.

Parameters **name** (*string*) – the name of the feature to disable.

classmethod delFeature(*name*)
delete a feature entirely

Parameters **name** (*string*) – the name of the feature to remove

class twiggy.logger.**InternalLogger**(*output*, *fields=None*, *options=None*, *min_level=None*)
Special-purpose logger for internal uses

Sends messages directly to output, bypassing *emitters*.

Variables **output** (Output) – an output to write to

twiggy.logger.**emit**(*level*)
a decorator that emits at *level* after calling the method. The method should return a *Logger* instance.

For convenience, decorators for the various levels are available as `emit.debug`, `emit.info`, etc..

3.4.9 Message

class twiggy.message.**Message**(*level*, *format_spec*, *fields*, *options*, *args*, *kwargs*)
A logging message. Users never create these directly.

Changed in version 0.4.1: Pass args/kwargs as list/dict instead of via */** expansion. The constructor takes a dict of `options` to control message creation. In addition to *suppress_newlines*, the following options are recognized:

trace control traceback inclusion. Either a traceback tuple, or one of the strings `always`, `error`, in which case a traceback will be extracted from the current stack frame.

style the style of template used for `format_spec`. One of `braces`, `percent`, `dollar`. The aliases `{}`, `%` and `$` are also supported.

Any callables passed in `fields`, `args` or `kwargs` will be called and the returned value used instead. See *dynamic messages*.

All attributes are read-only.

fields
dictionary of *structured logging* fields. Keys are string, values are arbitrary. A `level` item is required.

suppress_newlines
should newlines be escaped in output. Boolean.

traceback
a stringified traceback, or None.

text
the human-readable message. Constructed by substituting `args`/`kwargs` into `format_spec`. String.

__init__(*level*, *format_spec*, *fields*, *options*, *args*, *kwargs*)

Parameters

- **level** (LogLevel) – the level of the message
- **format_spec** (*string*) – the human-readable message template. Should match the `style` in options.

- **fields** (*dict*) – dictionary of fields for *structured logging*
- **args** (*tuple*) – substitution arguments for `format_spec`.
- **kwargs** (*dict*) – substitution keyword arguments for `format_spec`.
- **options** (*dict*) – a dictionary of *options* to control message creation.

3.4.10 Outputs

class twiggy.outputs.**Output**(*format=None*, *close_atexit=True*)

_format
a *callable* taking a *Message* and formatting it for output. None means return the message unchanged.

use_locks
Class variable, indicating that locks should be used when running in a synchronous, multithreaded environment. Threadsafe subclasses may disable locking for higher throughput. Defaults to True.

__init__(*format=None*, *close_atexit=True*)

Parameters

- **format** (*format*) – the format to use. If None, return the message unchanged.
- **close_atexit** (*bool*) – should *close()* be registered with `atexit`. If False, the user is responsible for closing the output.

New in version 0.4.1: Add the `close_atexit` parameter.

close()
Finalize the output.

The following methods should be implemented by subclasses.

_open()
Acquire any resources needed for writing (files, sockets, etc.)

_close()
Release any resources acquired in *_open*

_write(*x*)
Do the work of writing

Parameters **x** – an implementation-dependent object to be written.

class twiggy.outputs.**AsyncOutput**(*msg_buffer=0*)
An *Output* with support for *asynchronous logging*.

Inheriting from this class transparently adds support for asynchronous logging using the multiprocessing module. This is off by default, as it can cause log messages to be dropped.

Parameters **msg_buffer** (*int*) – number of messages to buffer in memory when using asynchronous logging. `0` turns asynchronous output off, a negative integer means an unlimited buffer, a positive integer is the size of the buffer.

class twiggy.outputs.**FileOutput**(*name*, *format*, *mode='a'*, *buffering=1*, *msg_buffer=0*, *close_atexit=True*)
Output messages to a file

`name`, `mode`, `buffering` are passed to `open()`

class twiggy.outputs.**StreamOutput**(*format*, *stream=sys.stderr*)
Output to an externally-managed stream.

The stream will be written to, but otherwise left alone (i.e., it will *not* be closed).

class twiggy.outputs.**NullOutput**(*format=None*, *close_atexit=True*)
An output that just discards its messages

class twiggy.outputs.**ListOutput**(*format=None*, *close_atexit=True*)
an output that stuffs messages in a list

Useful for unittesting.

Variables **messages** (*list*) – messages that have been emitted

Changed in version 0.4.1: Replace `DequeOutput` with more useful *ListOutput*.

3.5 Testing

This part discusses how to test Twiggy to ensure that Twiggy is built and installed correctly.

3.5.1 Requirements

The following need to be installed prior to testing:

- Python 2.7.1 or greater.
- The coverage module.
- sphinx 1.0.8 or greater. You'll need to get and build the sphinx source.
- Twiggy source.

3.5.2 Running Tests

Note: Tests **must** be run from the Twiggy root directory to work.

To run all tests (unittest and Sphinx doctests):

```
./scripts/run-twiggy-tests.sh
```

To run coverage tests, run:

```
./scripts/cover-twiggy-tests.sh discover -b
```

To run coverage tests on a specific module, run:

```
./scripts/cover-twiggy-tests.sh tests.test_levels
```

3.6 Glossary

asynchronous logging performance enhancement that moves formatting and writing messages to a separate process. See *Asynchronous Logging*.

structured logging logging information in easy-to-parse key-value pairs, instead of embedded in a human-readable message. See an *example*

3.7 Changelog

3.7.1 0.5.0

XXX Unreleased

- add a NOTICE level between INFO and WARNING
- add sameValue, sameItem, drop helper functions to lib.converter
- support {}, %, $ as style aliases.
- PEP8 name compliance
- add logging_compat module for compatibility with stdlib's logging
- add dict_config to configure logging from user configuration

3.7.2 0.4.7

03/09/2015 - add missing classifiers to setup.py

3.7.3 0.4.6

03/09/2015 - also suppress newlines in fields output - Python 3 support

3.7.4 0.4.5

03/18/2013 - documentation update, move to Github

3.7.5 0.4.4

07/12/2011 - support Python 2.6

3.7.6 0.4.3

12/20/2010 - add check for Python >= 2.7 to setup.py, to reduce invalid bug reports.

3.7.7 0.4.2

11/11/2010 - fix broken installer

3.7.8 0.4.1

11/8/2010

- full test coverage; numerous bug fixes
- add close_atexit parameter to Outputs
- replace DequeOutput with ListOutput
- deprecate features, pending a rewrite in 0.5
- minor internal API changes

3.7.9 0.4.0

10/18/2010

First serious public release

3.8 Contributors

Twiggy would not be possible without the support of the following people. You have our thanks.

- Peter Fein `pfein@pobox.com`
- Ian Foote `ianfoote@f2s.com`
- Kyle Rickey `kwkard@gmail.com`
- Lin O'Driscoll `nzlinus@gmail.com`

Python Module Index

t

Index

Symbols

A

B

C

D

E

F

G

I