logconfig Documentation

Release 0.4.0

Derrick Gilland

February 13, 2017

Contents

Simple helper moudle for configuring Python logging.

CHAPTER 1

Requirements

1.1 Compatibility

- Python 2.6
- Python 2.7
- Python 3.3
- Python 3.4

1.2 Dependencies

- PyYAML
- logutils (if using Python 2)

CHAPTER 2

Installation

```
pip install logconfig
```

CHAPTER 3

Overview

This simple library exposes several helper methods for configuring the standard library's `logging` module. There's nothing fancy about it. Under the hood `logconfig` uses `logging.config` to load various configuartion formats.

In addition to configuration loading, `logconfig` provides helpers for easily converting a configured logger's handlers to utilize a queue.

3.1 Supported Configuration Formats

- JSON
- YAML
- ConfigParser
- Python Dict

CHAPTER 4

Quickstart

4.1 Configuration Loading

```
import logconfig
import logging

# Load config from JSON file
logconfig.from_json('path/to/file.json')

# Load config from YAML file
logconfig.from_yaml('path/to/file.yml')

# Load config from ConfigParser file
logconfig.from_file('path/to/file.cfg')

# Load config from dict
logconfig.from_dict(config_dict)

log = logging.getLogger()
log.debug('Configuration loaded using logconfig')
```

4.2 Queue Utilization

```
import logconfig
import logging

logconfig.from_dict({
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'DEBUG'
        }
    },
    'loggers': {
        'mylogger': {
            'handlers': ['console']
        }
    }
```

```
})

# Convert logger's handlers to utilize a queue
queue = logconfig.Queue(-1)
listener = logconfig.QueueListener(queue)
handler = logconfig.QueueHandler(queue)

mylogger = logging.getLogger('mylogger')

# You can also pass in the logger name instead of the actual logger.
# logconfig.queuify_logger('mylogger', handler, listener)
logconfig.queuify_logger(mylogger, handler, listener)

assert isinstance(mylogger.handlers[0], logconfig.QueueHandler)

# Start the listener.
listener.start()

# When finished, stop the listener.
# This is optional, but not doing so may prevent some logs from being processed.
listener.stop()
```

CHAPTER 5

Usage

Use `logconfig` to easily load `logging` configurations. For more details on configuring `logging`, visit https://docs.python.org/library/logging.config.html.

```
import logconfig
```

5.1 Configuration from JSON

Configure logging using JSON file.

```
logconfig.from_json(filename)
```

Example JSON file:

```
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(asctime)s. - %(name)s - %(levelname)s - %(message)s"
        }
    },
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "level": "DEBUG",
            "formatter": "simple",
            "stream": "ext://sys.stdout"
        }
    },
    "root": {
        "level": "DEBUG",
        "handlers": ["console"]
    }
}
```

5.2 Configuration from YAML

Configure logging using YAML file.

```
logconfig.from_yaml(filename)
```

Example YAML file:

```
version: 1
disable_existing_loggers: False
formatters:
  simple:
    format: "%(asctime)s. - %(name)s - %(levelname)s - %(message)s"
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
root:
  level: DEBUG
  handlers: [console]
```

5.3 Configuration from ConfigParser File

Configure logging using ConfigParser compatible file.

```
logconfig.from_file(filename)
```

Example CFG file:

```
[loggers]
keys=root

[handlers]
keys=console

[formatters]
keys=simple

[logger_root]
level=DEBUG
handlers=console

[handler_console]
class=StreamHandler
level=DEBUG
formatter=simple
args=(sys.stdout,)

[formatter_simple]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
```

5.4 Configuration from Dict

Configure logging using Python dictionary.

```
logconfig.from_dict(dct)
```

Example dict:

```
{
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'simple': {
            'format': '%(asctime)s. - %(name)s - %(levelname)s - %(message)s'
        }
    },
    'handlers': {
        'console': {
            'formatter': 'simple',
            'class': 'logging.StreamHandler',
            'level': 'DEBUG',
            'stream': 'ext://sys.stdout'
        }
    },
    'root': {
        'handlers': ['console'],
        'level': 'DEBUG'
    }
}
```

5.5 Configuration from Autodetection

If, for whatever reason, you do not know what the source of the configuration will be (or if you're just feeling lucky), then you can try to coerce logging configuration using one of the autodetection methods:

```
logconfig.from_filename(filename)
logconfig.from_autodetect(filename_or_dict)
```

```
try:
    logconfig.from_filename(filename)
    logconfig.from_autodetect(filename_or_dict)
except logconfig.LogConfigException as ex:
    # Unrecognized configuration argument.
    pass
```

These methods will try to dispatch the function argument to the proper configuration loader or fail trying.

5.6 Configuration from Environment Variable

Configure logging using filename provided via environment variable.

```
logconfig.from_env(variable_name)
```

NOTE: Environment variable value will be passed to `from_filename()`.

CHAPTER 6

Guide

6.1 Installation

logconfig requires Python >= 2.6 or >= 3.3.

To install from PyPi:

```
pip install logconfig
```

CHAPTER 7

API Reference

Includes links to source code.

7.1 API Reference

7.1.1 Loaders

Functions that load `logging` configurations from various sources.

`logconfig.`**`from_autodetect`**(*obj*)
Dispatch logging configuration based on autodetecting object type.

Parameters **obj** (*mixed*) – Object to load configuration from.

New in version 0.1.0.

`logconfig.`**`from_dict`**(*dct*)
Configure logging module using dict object.

Parameters **dct** (*dict*) – Dict to load configuration from.

New in version 0.0.1.

`logconfig.`**`from_env`**(*var*)
Dispatch logging configuration based on filename provided through environment variable.

Parameters **var** (*str*) – Environtment variable name to load configuration from.

New in version 0.1.0.

`logconfig.`**`from_file`**(*filename*, ***kargs*)
Configure logging module using configparser-format file.

Parameters **filename** (*str*) – String filename to load configuration from.

Keyword Arguments

- **defaults** (*dict, optional*) – Defaults to be passed to the ConfigParser.
- **disable_existing_loggers** (*bool, optional*) – Whether to disable existing loggers. Defaults to `True`. The default `True` is what the default `logging.config.fileConfig` uses.

New in version 0.0.1.

`logconfig.`**`from_filename`**(*filename*)
Dispatch logging configuration based on filename.

Parameters **filename** (*str*) – String filename to load configuration from. Supported filename extensions: *.json*, *yml*, *.yaml*, *.cfg*, *.ini*, *.conf*, *.config*.

Raises `LogConfigException` – Raised if unsupported filename extension is used.

New in version 0.1.0.

`logconfig.`**`from_json`**(*filename*)
Configure logging module using JSON file.

Parameters **filename** (*str*) – String filename to load configuration from.

New in version 0.0.1.

`logconfig.`**`from_yaml`**(*filename*)
Configure logging module using YAML file.

Parameters **filename** (*str*) – String filename to load configuration from.

New in version 0.0.1.

7.1.2 Utilities

Basic utility functions and classes.

`logconfig.`**`get_all_loggers`**()
Return `dict` of all loggers than have been accessed.

New in version 0.3.0.

class `logconfig.`**`Queue`**(*maxsize=0*)
Create a queue object with a given maximum size.

If maxsize is <= 0, the queue size is infinite.

`empty`()
Return True if the queue is empty, False otherwise (not reliable!).

`full`()
Return True if the queue is full, False otherwise (not reliable!).

`get`(*block=True*, *timeout=None*)
Remove and return an item from the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until an item is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Empty exception if no item was available within that time. Otherwise ('block' is false), return an item if one is immediately available, else raise the Empty exception ('timeout' is ignored in that case).

`get_nowait`()
Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the Empty exception.

`join`()
Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

put(*item*, *block=True*, *timeout=None*)
Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

put_nowait(*item*)
Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

qsize()
Return the approximate size of the queue (not reliable!).

task_done()
Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

class logconfig.**QueueHandler**(*queue*)
This handler sends events to a queue. Typically, it would be used together with a multiprocessing Queue to centralise logging to file in one process (in a multi-process application), so as to avoid file write contention between processes.

Parameters **queue** – The queue to send *LogRecords* to.

emit(*record*)
Emit a record.

Writes the LogRecord to the queue, preparing it for pickling first.

Parameters **record** – The record to emit.

enqueue(*record*)
Enqueue a record.

The base implementation uses `put_nowait()`. You may want to override this method if you want to use blocking, timeouts or custom queue implementations.

Parameters **record** – The record to enqueue.

prepare(*record*)
Prepares a record for queuing. The object returned by this method is enqueued.

The base implementation formats the record to merge the message and arguments, and removes unpickleable items from the record in-place.

You might want to override this method if you want to convert the record to a dict or JSON string, or send a modified copy of the record while leaving the original intact.

Parameters **record** – The record to prepare.

class logconfig.**QueueListener**(*queue*, **handlers*)
Extension of default `QueueListener` that respects `handler.level` when handling records.

New in version 0.3.0.

handle(*record*)
Delegate handling of log records to listened handlers if record's log level is greater than or equal to handler's level.

logconfig.**queuify_logger**(*logger*, *queue_handler*, *queue_listener*)
Replace logger's handlers with a queue handler while adding existing handlers to a queue listener.

This is useful when you want to use a default logging config but then optionally add a logger's handlers to a queue during runtime.

Parameters

- **logger** (*mixed*) – Logger instance or string name of logger to queue-ify handlers.
- **queue_handler** (*QueueHandler*) – Instance of a `QueueHandler`.
- **queue_listener** (*QueueListener*) – Instance of a `QueueListener`.

New in version 0.3.0.

7.1.3 Exceptions

Custom exceptions used within module.

exception logconfig.**LogConfigException**
Base exception for logconfig module.

CHAPTER 8

Project Info

8.1 License

8.2 Changelog

8.2.1 v0.4.0 (2014-12-24)

- Rename project to `logconfig`.
- Rename `ConfiglogException` to `LogConfigException`. (**breaking change**)

8.2.2 v0.3.1 (2014-12-23)

- Rename `ConfigLogException` to `ConfiglogException`. (**breaking change**)

8.2.3 v0.3.0 (2014-12-23)

- Add support for Python 2.6
- Add `queuify_logger()` for moving logger's handlers to a queue.

- Add `get_all_loggers()` which returns a `dict` of all loggers that have been accessed.
- Proxy access to `QueueHandler` and `QueueListener` from either `logutils` if on Python 2 or `logging.handlers` if on Python 3.
- Extend base `QueueListener` class to respect log level of handler. The current stdlib version doesn't do this.

8.2.4 v0.2.0 (2014-12-23)

- Rename project to `configlog`. (**breaking change**)
- Rename `ConfigException` to `ConfigLogException`. (**breaking change**)

8.2.5 v0.1.1 (2014-07-16)

Brown bag release.

8.2.6 v0.1.0 (2014-07-16)

- Add `from_filename()`.
- Add `from_autodetect()`.
- Add `from_env()`.

8.2.7 v0.0.1 (2014-07-15)

- First release

8.3 Authors

8.3.1 Lead

- Derrick Gilland, dgilland@gmail.com, dgilland@github

8.3.2 Contributors

None

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

E

F

G

H

J

L

P

Q

T