Burin

Release 0.2.0

William Foster

Feb 28, 2024

BURIN DOCUMENTATION

Burin (/ˈbʏʊər ɪn, ˈbɜr-/byoor-in, bur-/) is a logging library that is meant to add features and simplify usage compared to the Python standard library `logging` package. It can be used as a direct replacement in most cases.

The name Burin is based on the (originally French) name of a handheld chisel like tool used for engraving.

Currently Python 3.7, 3.8, 3.9, 3.10, 3.11, and 3.12 are all supported. There are no dependencies or additional requirements for Burin and it should work on any platform that Python does.

Warning: Python 3.7 support is deprecated and will be removed in a future release.

An important aspect of Burin is an easy migration that allows changing from the `logging` package to Burin without anything breaking in most use cases. While class names may need to be changed this generally should work well. Although some situations may require other small changes due to the added features of Burin.

Using Burin to replace `logging` use in a program can be done gradually or all at once. Burin should not interfere with `logging` usage as its internal references are all managed independently. However; it's best to ensure that they are not trying to log to the same file as this may cause issues.

Note: While some classes in Burin inherit from classes in the Python standard `logging` package they cannot be used interchangeably.

Using classes from `logging` in Burin or vice-versa may cause exceptions or other issues.

Note: Burin is still in early development and may change in backwards incompatible ways between minor release versions. This should be rare as general compatibility with `logging` is desired to ease switching, but it is a good idea check the release notes when upgrading between minor (0.X.0) releases.

CHAPTER

ONE

WHAT'S DIFFERENT IN BURIN?

The following make up the current major differences compared to the Python standard `logging` module.

- Extra arguments and changes to *`basic_config()`* (page 14) allow it to be used in more situations and when setting up common logging configurations.
- Built-in support for deferred formatting of `str.format()` and `string.Template` style logging messages.
- Library level logging calls (eg. *`burin.log()`* (page 17)) can specify a logger to use other than the root logger, so calling *`get_logger()`* (page 18) isn't necessary first.
- Logging features from newer versions of Python (eg. *`burin.config.logAsyncioTasks`* (page 12) in 3.12) are implemented in Burin and available in all supported Python versions.
- Everything that should be needed is available at the top level of the library; no more extra imports of `logging.handlers` and `logging.config`.
- Multiple log record factory classes are supported at the same time, and which is used can be set per logger instance to allow for greater customisation.
- *`BurinLoggerAdapter`* (page 30) instances will merge *extra* values from logging calls with the pre-set values from instantiation; nesting built-in adapters can actually be useful now.
- All handlers within Burin support a *level* parameter during initialization so an extra call *`BurinHandler.set_level()`* (page 40) isn't needed
- *`BurinSocketHandler`* (page 46) and *`BurinDatagramHandler`* (page 36) by default use pickling protocol version **4** instead of **1**. This can be set to a different protocol version when creating the handler.
- All methods and functions are *underscore_separated*, but *camelCase* aliases are available for an easier transition from the standard library.
- Logging configuration attributes `logMultiprocessing`, `logProcesses`, `logThreads`, and `raiseExceptions` are on a `burin.config` object instead of directly on the module.
- Deprecated methods such as `fatal` and `warn` are not implemented.

There are several other differences which are more minor or are internal to Burin and not documented in this list. If you are going to create subclasses or use internal classes and methods, then just make sure to read the documentation or docstrings within the code.

CHAPTER

TWO

WHAT CAN'T BURIN DO?

Burin is still in early development and does not yet support some use cases that are supported by Python `logging`. These features are planned to be implemented before Burin reaches its first stable major (1.0.0) release.

- Advanced configuration functions like those from `logging.config` (`dictConfig`, `fileConfig`, and `listen`) are not yet implemented.
- Custom log levels are not yet supported.

CHAPTER

THREE

USING BURIN

Below are a few examples of using the features of Burin. Read through the rest of the documentation to see the full information on the functionality of Burin.

Note: All Burin functions and methods are *underscore_separated*, however to ease changing from the standard library they all also have *camelCase* aliases.

Throughout this documentation the *underscore_separated* names are used, but every time you see a function or method call in Burin just remember that the *camelCase* name (like what is in `logging`) will also work.

Burin can be imported and used similar to the `logging` standard library package.

```
import burin
burin.basic_config()
logger = burin.get_logger()
logger.info("I am a log message.")
```

What is above would do the exact same thing with both Burin and `logging`.

3.1 A Not So "Basic" Config

However compared to the standard `logging` package; using Burin can be much simpler for certain things, or even allow some functionality that would otherwise require custom wrapper utilities or overriding logging subclasses.

For example a common logging setup may be to output info level logs to a rotating file with a specific format, and also output warning level logs to `sys.stderr` in a different format.

With Burin setting this up can be accomplished with 2 imports and 1 call to *`basic_config()`* (page 14).

```
import sys
import burin
burin.basic_config(filename="prog.log", filelevel="INFO", filerotate=True,
                   fileformat="{asctime} - {levelname} :{name}: {message}",
                   filerotatesize=1048576, filerotatecount=9, level="INFO",
                   stream=sys.stderr, streamlevel="WARNING",
                   streamformat="{levelname}: {message}", style="{")
```

Whereas with `logging` this takes 3 imports and 12 lines.

```
import sys
import logging
from logging.handlers import RotatingFileHandler
```

(continues on next page)

(continued from previous page)

```
fileForm = logging.Formatter("{asctime} - {levelname} :{name}: {message}",
                             style="{")
fileHand = RotatingFileHandler("prog.log", maxBytes=1048576, backupCount=9)
fileHand.setFormatter(fileForm)
fileHand.setLevel("INFO")
streamForm = logging.Formatter("{levelname}: {message}", style="{")
streamHand = logging.StreamHandler(sys.stderr)
streamHand.setFormatter(streamForm)
streamHand.setLevel("WARNING")
rootLogger = logging.getLogger()
rootLogger.addHandler(fileHand)
rootLogger.addHandler(streamHand)
rootLogger.setLevel("INFO")
```

3.2 Deferred Formatting Styles

Burin also supports deferred formatting with log messages using `str.format()` and `string.Template` style strings, as well as the '%' style formatting that the standard library does. Which formatting is used is set by the *BurinLogger.msgStyle* (page 25) property on a logger which can also be specified when calling *get_logger()* (page 18).

```
formatLogger = burin.get_logger("formatLogger", "{")
formatLogger.debug("This is a {} event in {}", "DEBUG", "Burin")
templateLogger = burin.get_logger("templateLogger", msgStyle="$")
templateLogger.debug("This is a ${lvl} event in ${prog}", lvl="DEBUG",
                     prog="Burin")
```

Setting this on the root logger will set the default style for new loggers as well.

```
rootLogger = burin.get_logger(msgStyle="{")
newLogger = burin.get_logger("new")
newLogger.debug("This is a {lvl} event in {prog}", lvl="DEBUG",
                prog="Burin")
```

Deferred formatting means that all of the extra formatting is only done if a message will be logged, so this can be more efficient than doing the formatting on the string beforehand.

For a bit more information about the deferred logging see *BurinLogger.log()* (page 28).

3.3 Customisable Log Records

Setting the `msgStyle` of a logger actually sets the log record factory that is used. While the default built-in factories are focused on formatting, you can actually add any other custom factories that may be useful in your program. These factories can then just be used where needed instead of for all log messages as in the standard library.

This can be incredibly useful when you need a log to display values in a specific way, but only want that extra processing to run if the log message will actually be output.

To add your own factory simply create a subclass of *BurinLogRecord* (page 55) and then set it to a *msgStyle* with *set_log_record_factory()* (page 20).

```
class HexRecord(burin.BurinLogRecord):
    """
    Converts all int argument values to hex strings for log output.
    """

    def get_message(self):
        msg = str(self.msg)
        if self.args or self.kwargs:
            hexArgs = []
            hexKwargs = {}

            for eachArg in self.args:
                if isinstance(eachArg, int):
                    eachArg = hex(eachArg)
                hexArgs.append(eachArg)

            for eachKey, eachValue in self.kwargs.items():
                if isinstance(eachValue, int):
                    eachValue = hex(eachValue)
                hexKwargs[eachKey] = eachValue

            msg = msg.format(*hexArgs, **hexKwargs)
        return msg

burin.set_log_record_factory(HexRecord, "hex")
```

In this example you would now be able to use `hex` as a *msgStyle* for any loggers where you want int *args* and *kwargs* converted to a hexadecimal string when the log message is output.

CHAPTER

FOUR

THE BURIN MODULE

Module Contents

4.1 Overview

Within Burin everything needed for normal usage is available on the top-level `burin` module. There is typically no need to import any other packages or modules from within Burin.

Formatters (page 51), *Handlers* (page 33), *Loggers and Logger Adapters* (page 23), and *Filters and Filterers* (page 59) are all documented in their own sections. This page will focus on the *Constants* (page 12), *Config Attributes* (page 12), and *Functions* (page 13) that are available directly on the `burin` module.

4.2 Constants

The logging levels are integer values that correspond to a kind of seriousness of a logging event; with higher values being more serious.

Whenever you need to pass a logging level these values can be used directly such as `burin.DEBUG`, as string representations like `"DEBUG"`, or as straight integer values.

Note: Burin does not currently support customisation or adding of your own log levels. This is planned to be added in a future release.

`burin.`**`CRITICAL = 50`**

`burin.`**`ERROR = 40`**

`burin.`**`WARNING = 30`**

`burin.`**`INFO = 20`**

`burin.`**`DEBUG = 10`**

`burin.`**`NOTSET = 0`**

4.3 Config Attributes

There are a handful of attributes that control some aspects of logging. These can be configured through the `burin.config` object.

Most of these control whether some data is available for inclusion in logs or not.

Note: This differs slightly from `logging` where the attributes are directly on the module.

`burin.config.`**`logAsyncioTasks = True`**

Whether `asyncio.Task` names should be available for inclusion in logs. Whatever value is set for this will be automatically converted using `bool()`.

Note: In Python 3.12 this was added to the standard `logging` module; it is supported here for all versions of Python compatible with Burin (including versions below 3.12).

However; names were added to `asyncio.Task` objects in Python 3.8, so in Python 3.7 the *taskName* attribute on a log record will always be **None**.

`burin.config.`**`logMultiprocessing = True`**

Whether multiprocessing info should be available for inclusion in logs. Whatever value is set for this will be automatically converted using `bool()`.

`burin.config.`**`logProcesses = True`**

Whether process info should be available for inclusion in logs. Whatever value is set for this will be automatically converted using `bool()`.

burin.config.**logThreads = True**

Whether threading info should be available for inclusion in logs. Whatever value is set for this will be automatically converted using `bool()`.

burin.config.**raiseExceptions = True**

Whether exceptions during handling should be propagated or ignored. Whatever value is set for this will be automatically converted using `bool()`.

4.4 Functions

There are many top-level functions within the Burin module. These provide ways to configure logging, get loggers, log directly, or add customised functionality.

Note: Many of these functions will have slight changes from the standard `logging` module due to added functionality.

In most use cases though calling any of these functions in the same way as the `logging` module should work exactly the same way.

Note: All of the functions with an *underscore_separated* name also have a *camelCase* alias name which matches the names used in the standard `logging` library.

Below is a summary list of all functions in the module; and then further below are the full details of the functions grouped into categories based on their general purpose.

basic_config (page 14)	Does a basic configuration of the Burin root logger.
capture_warnings (page 21)	Enables or disables capturing of warnings through logs instead.
critical (page 16)	Logs a message with the *CRITICAL* (page 12) level.
debug (page 16)	Logs a message with the *DEBUG* (page 12) level.
disable (page 20)	Provides a way to easily disable a level for all loggers.
error (page 16)	Logs a message with the *ERROR* (page 12) level.
exception (page 16)	Logs a message with the *ERROR* (page 12) level and exception information.
get_handler_by_name (page 19)	Gets a handler with the specified name.
get_handler_names (page 19)	Gets all known handler names as an immutable set.
get_level_name (page 21)	Return the textual or numeric representation of a logging level.
get_level_names_mapping (page 21)	Gets the current log levels name to level mapping.
get_log_record_factory (page 19)	Gets the log record factory class for the specified style.
get_logger (page 18)	Get a logger with the specified name and msgStyle.
get_logger_class (page 18)	Gets the class that is used when instantiating new loggers.
info (page 16)	Logs a message with the *INFO* (page 12) level.
log (page 17)	Logs a message with the specified *level*.
make_log_record (page 20)	Creates a new log record from a dictionary.
set_log_record_factory (page 20)	Sets the log record class to use as a factory.
set_logger_class (page 18)	Sets a class to be used when instantiating new loggers.
shutdown (page 22)	Cleans up by flushing and closing all handlers.
warning (page 17)	Logs a message with the *WARNING* (page 12) level.

4.4.1 Configuration

These functions are used to configure the logging setup.

Note: Burin does not yet support functionality similar to the `logging.config` `dictConfig`, `fileConfig`, and `listen` functions. This is planned to be added in a future release.

`burin.`**`basic_config`**(*, *datefmt=None*, *encoding=None*, *errors='backslashreplace'*, *filedatefmt=None*, *filedelay=False*, *fileformat=None*, *filelevel=None*, *filemode='a'*, *filename=None*, *filerotate=False*, *filerotatecount=4*, *filerotatesize=1048576*, *force=False*, *format=None*, *handlers=None*, *level='WARNING'*, *msgstyle='%'*, *stream=None*, *streamdatefmt=None*, *streamformat=None*, *streamlevel=None*, *style='%'*)

Does a basic configuration of the Burin root logger.

This function will configure handlers for the root logger. This is a convenience method intended to cover several common logging use cases.

Note: All arguments to this function are optional and must be passed as keyword arguments, no positional arguments are supported.

With this function a file handler (or rotating file hander) and stream handler can both be added to the root logger. An iterable of other *handlers* can also be added. This differs from the standard `logging.basicConfig()` where only one of these can be configured.

The file and stream handlers can each have a different *format*, *datefmt*, and *level* using the parameters prefixed with *file* or *stream*. The general *format*, *datefmt*, and *level* parameters are used if file or stream specific values are not set.

Any handler within *handlers* that does not have a formatter will have a formatter set for them using the general *format* and *datefmt*.

If the root logger already has handlers this function can still be used to add additional *handlers*, configure new file and/or stream handlers for the root logger, and change other settings of the root logger. This differs from the standard `logging.basicConfig()` where nothing would be done if the root logger already has handlers.

If *handlers*, *filename*, and *stream* are all **None**, and the root logger does not have any handlers then a stream handler using `sys.stderr` is created and added to the root logger. If these arguments are all **None** and the root logger does have handlers then the other configuration values are still applied if set (*level* and *msgstyle*).

Parameters

- **`datefmt`** (*`str`*) – The date/time format to use (as accepted by `time.strftime()`).
- **`encoding`** (*`str`*) – If specified with a filename this is passed to the handler and used when the file is opened.
- **`errors`** (*`str`*) – If specified with filename this is passed to the handler and used when the file is opened. (Default = 'backslashreplace')
- **`filedatefmt`** (*`str`*) – The date/time format to use specifically for the file handler. If this is **None** than the general *datefmt* argument is used instead.
- **`filedelay`** (*`bool`*) – Whether to delay opening the file until the first record is emitted. (Default = **False**)
- **`fileformat`** (*`str`*) – The format string to use specifically for the file handler. If this is **None** than the general *format* argument is used instead.

- **filelevel** – The level to set specifically for the file handler. If this is **None** then the general *level* argument is used instead.
- **filemode** (*str*) – If specified with filename then this is the mode with which the file is opened. (Default = 'a')
- **filename** (*str | pathlib.Path*) – Specifies that a file handler is to be created and the file path to write to.
- **filerotate** (*bool*) – Whether a rotating file handler or normal file handler should be created. (Default = **False**)
- **filerotatecount** (*int*) – If *filerotate* is **True** then this is how many extra log files should be kept after rotating. (Default = 4)
- **filerotatesize** (*int*) – If *filerotate* is **True** then this sets the size of the log file in bytes before it should be rotated. (Default = 1048576 (1MB))
- **force** (*bool*) – Whether all existing handlers on the root logger should be removed and closed. (Default = **False**)
- **format** (*str*) – The format string to use for the handlers. If this is **None** then a default format string will be used that has the level, logger name, and message.
- **handlers** (*list[*BurinHandler (page 38)*]*) – This can be an iterable of handlers that were already created and should be added to the root logger. Any handler within that doesn't have a formatter will have the general formatter assigned to it.
- **level** (*int | str*) – The level to set on the root logger. You can set separate levels for the file and stream handlers by using the *filelevel* and *streamlevel* parameters. (Default = 'WARNING')
- **msgstyle** (*str*) – This sets the style that is used for deferred formatting of log messages on the root logger. This will also change the default style for any new loggers created afterwards. Built in possible values are '%' for %-formatting, '{' for `str.format()` formatting, and '$' for `string.Template` formatting. Other values can be used if custom log record factories are added using *set_log_record_factory()* (page 20). (Default = '%')
- **stream** (*io.TextIOBase*) – Specifies that a stream handler is to be created with the passed stream for output.
- **streamdatefmt** (*str*) – The date/time format to use specifically for the stream handler. If this is **None** than the general *datefmt* argument is used instead.
- **streamformat** (*str*) – The format string to use specifically for the stream handler. If this is **None** than the general *format* argument is used instead.
- **streamlevel** (*int*) – The level to set specifically for the stream handler. If this is **None** then the general *level* argument is used instead.
- **style** (*str*) – The type of formatting to use for the format strings. Possible values are '%' for %-formatting, '{' for `str.format()` formatting, and '$' for `string.Template` formatting. (Default = '%')

Raises

- ***ConfigError*** (page 63) – If *msgstyle* does not match a type of log record factory.
- ***FormatError*** (page 63) – If there are errors with the *format*, *fileformat*, or *streamformat* strings or *style*.

4.4.2 Logging

These functions can be used directly to log messages with either the root logger or another logger by using the *logger* parameter to specify the name.

burin.**critical**(*msg*, **args*, *logger=None*, ***kwargs*)

Logs a message with the *CRITICAL* (page 12) level.

A specific logger can be used passing its name with the *logger* keyword argument; otherwise the root logger is used.

Additional arguments are interpreted the same way as *log()* (page 17).

Parameters

- **msg** (*str*) – The message to log.
- **logger** (*str*) – The name of the logger to log the event with.

burin.**debug**(*msg*, **args*, *logger=None*, ***kwargs*)

Logs a message with the *DEBUG* (page 12) level.

A specific logger can be used passing its name with the *logger* keyword argument; otherwise the root logger is used.

Additional arguments are interpreted the same way as *log()* (page 17).

Parameters

- **msg** (*str*) – The message to log.
- **logger** (*str*) – The name of the logger to log the event with.

burin.**error**(*msg*, **args*, *logger=None*, ***kwargs*)

Logs a message with the *ERROR* (page 12) level.

A specific logger can be used passing its name with the *logger* keyword argument; otherwise the root logger is used.

Additional arguments are interpreted the same way as *log()* (page 17).

Parameters

- **msg** (*str*) – The message to log.
- **logger** (*str*) – The name of the logger to log the event with.

burin.**exception**(*msg*, **args*, *exc_info=True*, *logger=None*, ***kwargs*)

Logs a message with the *ERROR* (page 12) level and exception information.

A specific logger can be used passing its name with the *logger* keyword argument; otherwise the root logger is used.

This should normally be called only within an exception handler.

Additional arguments are interpreted the same way as *log()* (page 17).

Parameters

- **msg** (*str*) – The message to log.
- **logger** (*str*) – The name of the logger to log the event with.

burin.**info**(*msg*, **args*, *logger=None*, ***kwargs*)

Logs a message with the *INFO* (page 12) level.

A specific logger can be used passing its name with the *logger* keyword argument; otherwise the root logger is used.

Additional arguments are interpreted the same way as *log()* (page 17).

Parameters

- **msg** (*str*) – The message to log.
- **logger** (*str*) – The name of the logger to log the event with.

burin.**log**(*level*, *msg*, **args*, *exc_info=None*, *extra=None*, *logger=None*, *stack_info=False*, *stacklevel=1*, ***kwargs*)

Logs a message with the specified *level*.

Note: The arguments *exc_info*, *extra*, *logger*, *stack_info*, and *stacklevel* are all keyword only arguments. These cannot be passed as positional arguments.

A specific logger can be used passing its name with the *logger* keyword argument; otherwise the root logger is used.

Any additional *args* and *kwargs* will be kept with the message and used for deferred formatting before output. Deferred formatting allows you to pass in a format string for the message and the values as additional arguments. The message then will only be formatted if it is going to be emitted by a handler.

How this formatting is done is determined by the log record factory used. This is controlled by the *BurinLogger.msgStyle* (page 25) property of the logger. See *BurinLogger.log()* (page 28) for examples of the different *msgStyle* deferred formatting options.

Additional or customised log record factories can be used by adding them with the *set_log_record_factory()* (page 20) function.

Parameters

- **level** (*int | str*) – The level to log the message at.
- **msg** (*str*) – The message to log.
- **exc_info** (*Exception | tuple(type, Exception, traceback) | bool*) – Exception information to be added to the logging message. This should be an exception instance or an exception tuple (as returned by `sys.exc_info()`) if possible; otherwise if it is any other value that doesn't evaluate as **False** then the exception information will be fetched using `sys.exc_info()`.
- **extra** (*dict{str: Any}*) – A dictionary of extra attributes that are applied to the log record's *__dict__*. These can be used to populate custom fields that you set in your format string for *BurinFormatter* (page 51). The keys in this dictionary must not interfere with the built in fields/keys in the log record.
- **logger** (*str*) – The name of the logger to log the event with. By default and when this is **None** then the root logger is used. If this is not **None** and the named logger doesn't exist then it is created first.
- **stack_info** (*bool*) – Whether to get the stack information from the logging call and add it to the log record. This allows for getting stack information for logging without an exception needing to be raised. (Default = **False**)
- **stacklevel** (*int*) – If this is greater than 1 then the corresponding number of stack frames are skipped back through when getting the logging caller's information (like filename, lineno, and funcName). This can be useful when the log call was from helper/wrapper code that doesn't need to be included in the log record.

Raises

KeyError – If any key in *extra* conflicts with a built in key of the log record.

burin.**warning**(*msg*, **args*, *logger=None*, ***kwargs*)

Logs a message with the *WARNING* (page 12) level.

A specific logger can be used passing its name with the *logger* keyword argument; otherwise the root logger is used.

Additional arguments are interpreted the same way as *log()* (page 17).

Parameters

- **msg** (*str*) – The message to log.
- **logger** (*str*) – The name of the logger to log the event with.

4.4.3 Loggers

These are the top-level functions you can use to get logger instances or to customise the logger class used.

burin.**get_logger**(*name=None*, *msgStyle=None*)

Get a logger with the specified name and msgStyle.

If *name* is **None** then the root logger will be returned. Otherwise it will try to find any previously created logger with the specified *name*.

If no existing logger with *name* is found then a new *BurinLogger* (page 24) instance is created with that *name*. When creating a new logger if *msgStyle* is **None** then the *msgStyle* of the root logger will be used.

A *name* can be any almost text value the developer likes. Any periods '.' within the names though will be treated as separators for a hierarchy of loggers. For example a name of 'a.b.c' will get or create logger 'a.b.c' which is a child of logger 'a.b'; logger 'a.b' is also a child of logger 'a'. Any parts of the hierarchy that don't exist already are constructed with placeholder classes that are replaced with actual loggers if ever fetched by name.

Children in the hierarchy typically propagate logging events up to parents which allows for handlers further up the hierarchy to emit these log records.

If *msgStyle* is not **None** then it will be set as the *msgStyle* on the retrieved logger. If this is the root logger then that will become the default *msgStyle* for all new loggers created afterwards.

Parameters

- **name** (*str*) – The name of the logger the get. If this logger doesn't exist already then it will be created. If this is **None** then the root logger will be returned.
- **msgStyle** – If this is not **None** then it is set as the *msgStyle* on the retrieved logger. If that is the root logger then this will also change the default *msgStyle* for any new loggers created afterwards. Built in possible values are '%' for %-formatting, '{' for `str.format()` formatting, and '$' for `string.Template` formatting. Other values can be used if custom log record factories are added using *set_log_record_factory()* (page 20).

Returns

The logger with the specified *name*.

Return type

BurinLogger (page 24)

Raises

FactoryError (page 63) – If *msgStyle* doesn't match any known log record factory.

burin.**get_logger_class**()

Gets the class that is used when instantiating new loggers.

Returns

The class new loggers are created with.

Return type

BurinLogger (page 24)

burin.**set_logger_class**(*newClass*)

Sets a class to be used when instantiating new loggers.

The class being set must be derived from *BurinLogger* (page 24).

Parameters
newClass (BurinLogger (page 24)) – A new class to be used when instantiating new loggers. This must be a subclass of *BurinLogger* (page 24).

Raises
TypeError – If the received class is not a subclass of *BurinLogger* (page 24).

4.4.4 Handlers

These are top-level functions that can be used to get handler names or a specific handler by name if a name has be set on the handler.

burin.**get_handler_by_name**(*name*)

Gets a handler with the specified name.

If no handler exists with the name then **None** is returned.

Parameters
name (*str*) – The name of the handler to get.

Returns
The handler with the specified name or **None** if it doesn't exist.

Return type
BurinHandler (page 38) | None

burin.**get_handler_names**()

Gets all known handler names as an immutable set.

Returns
A frozenset of the handler names.

Return type
frozenset

4.4.5 Log Records

The log record classes are used to represent the log event values and format the passed log message before output. These are referred to as factories when logger instances create an instance of the record for an event.

The built-in log record factories provide different formatting options as demonstrated in *Deferred Formatting Styles* (page 8). However custom log record factories can also be added by using the *set_log_record_factory()* (page 20). An example of this is shown in *Customisable Log Records* (page 8).

burin.**get_log_record_factory**(*msgStyle='%'*)

Gets the log record factory class for the specified style.

If no log record factory exists for the *msgStyle* then **None** is returned.

Parameters
msgStyle (*str*) – The style to get the associated log record factory for. (Default = '%')

Returns
The log record factory class associated with the *msgStyle* or **None** if no factory exists for that style.

Return type
BurinLogRecord (page 55) | None

burin.**make_log_record**(*recordDict*, *msgStyle='%'*)

Creates a new log record from a dictionary.

This is intended for rebuilding log records that were pickled and sent over a socket.

Typically *msgStyle* won't matter here as the msg formatting is done before a record is pickled and sent. It is provided as a parameter here for special use cases.

Parameters

- **recordDict** (*dict{str: Any}*) – The dictionary of the log record attributes.
- **msgStyle** (*str*) – The *msgStyle* of which log record factory to use when rebuilding the record. (Default = '%')

Returns
The reconstructed log record.

Return type
BurinLogRecord (page 55)

burin.**set_log_record_factory**(*factory*, *msgStyle='%'*)

Sets the log record class to use as a factory.

The factory can be set to any type of *msgStyle*. If a factory is already set for that *msgStyle* it is replaced, otherwise the new factory is simply added without impacting the other factories.

Once a factory has been set to a *msgStyle* then the same style can be used as the *msgStyle* on loggers to use that specific log record factory.

Parameters

- **factory** (BurinLogRecord (page 55)) – The new log record class to use as a factory. This should be a subclass of *BurinLogRecord* (page 55).
- **msgStyle** (*str*) – The style and key used to reference the factory for loggers. (Default = '%')

4.4.6 Log Levels

These functions are meant to help with some situations dealing with logging levels. For example disabling all logging of specific levels with one simple call, and providing a helper method that may be useful for some wrappers.

Note: Burin does not currently support customisation or adding of your own log levels. This is planned to be added in a future release.

---

burin.**disable**(*level='CRITICAL'*)

Provides a way to easily disable a level for all loggers.

This can be helpful for situations where you need to throttle logging output throughout an entire application quickly.

All logging events of *level* or below will not be processed.

To reset this back to normal it can simply be called again with *NOTSET* (page 12) as *level*.

Parameters
level (*int | str*) – The level where all logging events and below should not be processed. (Default = "CRITICAL")

burin.**get_level_name**(*level*)

Return the textual or numeric representation of a logging level.

If a numeric value corresponding to one of the defined levels (*CRITICAL* (page 12), *ERROR* (page 12), *WARNING* (page 12), *INFO* (page 12), *DEBUG* (page 12)) is passed in, the corresponding string representation is returned.

If a string representation of the level is passed in, the corresponding numeric value is returned.

Note: Unlike the standard library `logging.getLevelName()` a lower case name can also be used; all level name checks are automatically converted to uppercase.

If no matching numeric or string value is passed in, the string `f'Level {level}'` level is returned.

Parameters
level (*int | str*) – The logging level to get the text or numeric representation of.

Returns
If *level* is an *int* then a string representation of the level is returned; otherwise, if *level* is a *str* then an integer representation of the level is returned.

Return type
int | str

burin.**get_level_names_mapping**()

Gets the current log levels name to level mapping.

Note: In Python 3.11 `logging.getLevelNamesMapping()` was added to the standard library; it is supported here for all versions of Python compatible with Burin (including versions below 3.11).

Returns
A dictionary of the current logging level names mapped to the level values.

Return type
dict{str: int}

4.4.7 Warnings Integration

Like the Python standard `logging` package Burin also supports some integration with the `warnings` module.

burin.**capture_warnings**(*capture*)

Enables or disables capturing of warnings through logs instead.

When this is enabled `warnings.showwarning()` is overridden with a function that will automatically log all warnings that are called through `warnings.showwarning()`.

Parameters
capture (*bool*) – Enable or disabled capturing of warnings for log output.

4.4.8 Clean Up

Burin will handle cleaning up of all handlers automatically when the program exits, so there shouldn't be any need for manual cleanup. However; if you want Burin to clean up handlers before then you can call the *shutdown()* (page 22) function.

burin.**shutdown**(*handlerList=None*)

Cleans up by flushing and closing all handlers.

This is automatically registered with atexit.register() and therefore shouldn't need to be called manually when an application closes.

Note: In Python 3.12 this was changed to check if a handler has a *flushOnClose* property set to **False** to prevent flushing during shutdown (targetting logging.MemoryHandler). This is supported here for all versions of Python compatible with Burin (including versions below 3.12). The check was also left generic so any custom handlers that may not want to flush when closed can benefit.

Parameters

handlerList (*list[*BurinHandler (page 38)*]*) – The handlers to be cleaned up. If this is **None** then it will default to an internal list of all Burin handlers. This should not need to be changed in almost all circumstances. However; if you only want to clean up a specific set of handlers then pass them here.

CHAPTER

FIVE

LOGGERS AND LOGGER ADAPTERS

BurinLogger (page 24) instances are what actually handle and process logging events. Each logger will have a unique name and can have its own handlers, formatters, and message style.

Loggers also exist in a hierarchy, by default loggers will propagate their logging events up through the hierarchy so handlers assigned to other loggers higher up will also receive the event. This simplifies things as handlers don't need to be set on every single logger.

BurinLoggerAdapter (page 30) instances allow you to predefine *extra* fields and values for a logger without needing to provide them in every logging call. This can be useful if you want to log an extra field every time. Also unlike the `logging.LoggerAdapter` any *extra* values that are passed in a call to the adapter will get merged with the defaults that were set; this mean you can also nest adapters if needed.

Note: All methods of the *BurinLogger* (page 24) and *BurinLoggerAdapter* (page 30) classes with an *underscore_separated* name also have a *camelCase* alias name which matches the names used in the standard `logging` library.

5.1 BurinLogger

Most of the methods on a logger are only called internally by other parts of Burin and do not need to be called directly. The most commonly used methods would be *BurinLogger.add_handler()* (page 25), *BurinLogger.critical()* (page 25), *BurinLogger.debug()* (page 25), *BurinLogger.error()* (page 25), *BurinLogger.exception()* (page 26), *BurinLogger.info()* (page 27), *BurinLogger.log()* (page 28), and *BurinLogger.warning()* (page 29).

Here is a summary list of the methods for the *BurinLogger* (page 24) class; below that is a full description of the class, it attributes, and methods.

BurinLogger.add_handler (page 25)	Add the specified *handler* to this logger.
BurinLogger.call_handlers (page 25)	Passes a log record to all relevant handlers.
BurinLogger.critical (page 25)	Logs a message with the *CRITICAL* (page 12) level.
BurinLogger.debug (page 25)	Logs a message with the *DEBUG* (page 12) level.
BurinLogger.error (page 25)	Logs a message with the *ERROR* (page 12) level.
BurinLogger.exception (page 26)	Logs a message with the *ERROR* (page 12) level and exception information.
BurinLogger.find_caller (page 26)	Finds the logging event caller's information.
BurinLogger.get_child (page 26)	Gets a child of this logger.
BurinLogger.get_children (page 26)	Gets a set of loggers that are the immediate children of this logger.
BurinLogger.get_effective_level (page 27)	Gets the effective log level for this logger.
BurinLogger.handle (page 27)	Calls handlers for the record.
BurinLogger.has_handlers (page 27)	Checks if there are any available handlers for this logger.
BurinLogger.info (page 27)	Logs a message with the *INFO* (page 12) level.
BurinLogger.is_enabled_for (page 27)	Checks if the logger is enabled for the specified *level*.
BurinLogger.log (page 28)	Logs a message with the specified *level*.
BurinLogger.make_record (page 29)	Creates the log record and applies any extra fields to it.
BurinLogger.remove_handler (page 29)	Removes the specified *handler* from this logger.
BurinLogger.set_level (page 29)	Sets the level of this logger.
BurinLogger.warning (page 29)	Logs a message with the *WARNING* (page 12) level.

class burin.**BurinLogger**(*name*, *level='NOTSET'*, *msgStyle='%'*)

Loggers represent a logging channel within an application.

Note: While this is based off `logging.Logger` it is not a subclass of it and has a few differences and additions.

Deprecated methods like `logging.Logger.warn()` or `logging.Logger.fatal()` do not exist as methods for this class.

Other methods from `logging.Logger` can be called in the same way on this class without using any of the changes if desired.

This should never be instantiated directly during normal use; instead always use the *get_logger()* (page 18) function instead to create a new instance. Calling *get_logger()* (page 18) with the same name will always return the same logger instance.

What a logging channel encompasses is normally a specific area of the software and is up to each developer; it could be a class, module, package, process, etc.

Typically the name of the logger matches the area the logging channel represents; for example a common use case is `burin.get_logger(__name__)` which uses the module name for the logger.

BurinLoggers support a hierarchy similar to Python packages; so any periods '.' within a name represent multiple steps. An example is the name 'foo.bar.baz' which shows three different loggers at different places in the hierarchy. The logger 'foo' is higher up the hierarchy and is a parent of 'foo.bar', and then 'foo.bar' is subsquently a parent of 'foo.bar.baz'.

Children can propagate logging events up to parents above them in the hierarchy. This can simplify how handlers are setup as each logger doesn't need to have its own handlers added if somewhere up the line a parent has the desired handlers already attached.

Initialization of the logger sets it up to start processing log events.

Parameters

- **name** (*str*) – The name of the logger.
- **level** (*int | str*) – The logging level for the logger. (Default = 'NOTSET')
- **msgStyle** (*str*) – The style of deferred formatting to use for log messages. This determines the log record factory that is used when creating a new log record. Built in possible values are '%' for %-formatting, '{' for `str.format()` formatting, and '$' for `string.Template` formatting. Other values can be used if custom log record factories are added using *set_log_record_factory()* (page 20). (Default = '%')

Raises

FactoryError (page 63) – If *msgStyle* doesn't match any known log record factory.

property msgStyle

Determines the log record factory to use when creating new log records.

Built in possible values are '%' for %-formatting, '{' for `str.format()` formatting, and '$' for `string.Template` formatting. Other values can be used if custom log record factories are added using *set_log_record_factory()* (page 20).

Note: This will raise a *FactoryError* (page 63) if it is set to a value that doesn't match with any log record factory.

---

propagate = True

Whether logging events should be propagated up the logger hierarchy.

add_handler(*handler*)

Add the specified *handler* to this logger.

Parameters

handler (`BurinHandler` (page 38)) – The handler to add to the logger.

call_handlers(*record*)

Passes a log record to all relevant handlers.

This will call all handlers on this logger and then will move through parent loggers in the hierarchy calling their handlers based on propagation checks.

Parameters

record (`BurinLogRecord` (page 55)) – The log record to pass to the handlers.

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

Logs a message with the *CRITICAL* (page 12) level.

Additional arguments are interpreted the same way as *BurinLogger.log()* (page 28).

Parameters

msg (*str*) – The message to log.

debug(*msg*, **args*, ***kwargs*)

Logs a message with the *DEBUG* (page 12) level.

Additional arguments are interpreted the same way as *BurinLogger.log()* (page 28).

Parameters

msg (*str*) – The message to log.

error(*msg*, **args*, ***kwargs*)

Logs a message with the *ERROR* (page 12) level.

Additional arguments are interpreted the same way as *BurinLogger.log()* (page 28).

Parameters
msg (*str*) – The message to log.

exception(*msg*, **args*, *exc_info=True*, ***kwargs*)

Logs a message with the *ERROR* (page 12) level and exception information.

This should normally be called only within an exception handler.

Additional arguments are interpreted the same way as *BurinLogger.log()* (page 28).

Parameters
msg (*str*) – The message to log.

find_caller(*stack_info=False*, *stacklevel=1*)

Finds the logging event caller's information.

This will traverse back through frames until it is outside of the Burin library to find the caller of the logging event.

This will get the filename, line number, function name, and optionally the stack information of the caller.

Parameters

- **stack_info** (*bool*) – Whether the caller's stack information should be returned as well. (Default = **False**)
- **stacklevel** (*int*) – Allows stepping further back through stack frames in case the log call was from helper/wrapper code that should be ignored as well.

Returns
A tuple of the filename, line number, function name, and if *stack_info*=**True** the stack information.

Return type
tuple(str, int, str, str | None)

get_child(*suffix*)

Gets a child of this logger.

The suffix can have multiple steps down the hierarchy by including additional period separate names '.'; this will all be added as descendants of this logger instance.

Calling `burin.get_logger('abc').get_child('def.ghi')` would return the exact same logger as `burin.get_logger('abc.def.ghi')`.

If the requested logger already exists it is simply retrieved; otherwise it will be created.

Parameters
suffix (*str*) – The part of the child logger's name below this logger.

Returns
The child logger.

Return type
BurinLogger (page 24)

get_children()

Gets a set of loggers that are the immediate children of this logger.

Note: In Python 3.12 this method was changed on the standard `logging.Logger`; it is supported here for all versions of Python compatible with Burin (including versions below 3.12).

Returns
A set of loggers that are direct children of this logger.

Return type
set

get_effective_level()

Gets the effective log level for this logger.

This will check if a specific level is set on this logger and if not then it will check through its parents until it finds one. If no specific level is found then *NOTSET* (page 12) is returned.

Returns
The effective log level for this logger.

Return type
int

handle(*record*)

Calls handlers for the record.

This will check if the logger is disabled or any filters before calling handlers.

Parameters
record (BurinLogRecord (page 55)) – The log record to pass to the handlers.

has_handlers()

Checks if there are any available handlers for this logger.

This will check this logger and if it doesn't find any handlers it will move through parent loggers in the hierarchy looking for handlers based on propagation checks.

Returns
Whether this logger has any available handlers.

Return type
bool

info(*msg*, **args*, ***kwargs*)

Logs a message with the *INFO* (page 12) level.

Additional arguments are interpreted the same way as *BurinLogger.log()* (page 28).

Parameters
msg (*str*) – The message to log.

is_enabled_for(*level*)

Checks if the logger is enabled for the specified *level*.

Parameters
level (*int | str*) – The level to check on the logger.

Returns
If the logger is enabled for *level*.

Return type
bool

log(*level*, *msg*, **args*, *exc_info=None*, *extra=None*, *stack_info=False*, *stacklevel=1*, ***kwargs*)

Logs a message with the specified *level*.

Note: The arguments *exc_info*, *extra*, *stack_info*, and *stacklevel* are all keyword only arguments. These cannot be passed as positional arguments.

Any additional *args* and *kwargs* will be kept with the message and used for deferred formatting before output. Deferred formatting allows you to pass in a format string for the message and the values as additional arguments. The message then will only be formatted if it is going to be emitted by a handler.

How this formatting is done is determined by the log record factory used. This is controlled by the *BurinLogger.msgStyle* (page 25) property of the logger. See examples below.

% style:

```
# Positional format args
logger.log('DEBUG', 'This is a %s event in %s', 'DEBUG', 'Burin')
# Keyword format args in a dictionary
logger.log('DEBUG', 'This is a %(lvl)s event in %(prog)s',
           { 'lvl': 'DEBUG', 'prog': 'Burin'})
```

{ `str.format()` style:

```
# Positional format args
logger.log('DEBUG', 'This is a {} event in {}', 'DEBUG', 'Burin')
# Format args as keyword args
logger.log('DEBUG', 'This is a {lvl} event in {prog}', lvl='DEBUG',
           prog='Burin')
```

$ `string.Template` style:

```
# Format args as keyword args
logger.log('DEBUG', 'This is a ${lvl} event in ${prog}',
           lvl='DEBUG', prog='Burin')
```

Parameters

- **level** (*int | str*) – The level to log the message at.
- **msg** (*str*) – The message to log.
- **exc_info** (*Exception | tuple(type, Exception, traceback) | bool*) – Exception information to be added to the logging message. This should be an exception instance or an exception tuple (as returned by `sys.exc_info()`) if possible; otherwise if it is any other value that doesn't evaluate as **False** then the exception information will be fetched using `sys.exc_info()`.
- **extra** (*dict{str: Any}*) – A dictionary of extra attributes that are applied to the log record's *__dict__*. These can be used to populate custom fields that you set in your format string for *BurinFormatter* (page 51). The keys in this dictionary must not interfere with the built in fields/keys in the log record.
- **stack_info** (*bool*) – Whether to get the stack information from the logging call and add it to the log record. This allows for getting stack information for logging without an exception needing to be raised. (Default = **False**)

- **stacklevel** (*int*) – If this is greater than 1 then the corresponding number of stack frames are skipped back through when getting the logging caller's information (like filename, lineno, and funcName). This can be useful when the log call was from helper/wrapper code that doesn't need to be included in the log record.

Raises
KeyError – If any key in *extra* conflicts with a built in key of the log record.

make_record(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*, ***kwargs*)

Creates the log record and applies any extra fields to it.

The type of log record that is created is determined by this logger's *BurinLogger.msgStyle* (page 25) value.

Parameters

- **name** (*str*) – The name of the logger.
- **level** (*int*) – The level of the logging event.
- **fn** (*str*) – The filename of the log event caller.
- **lno** – The line number where the log event was called.
- **msg** (*str*) – The log message.
- **args** (*tuple(Any)*) – The additional positional arguments for the log event call.
- **exc_info** (*tuple(type, Exception, traceback)*) – The exception information if this log event is from an exception handler.
- **func** (*str*) – The name of the function where the log event was called.
- **extra** (*dict{str: Any}*) – Extra fields to be applied to the log record.
- **sinfo** (*str*) – The stack information for the log event call.

Returns
The newly created log record.

Return type
BurinLogRecord (page 55)

remove_handler(*handler*)

Removes the specified *handler* from this logger.

Parameters
handler (BurinHandler (page 38)) – The handler to remove from the logger.

set_level(*level*)

Sets the level of this logger.

Parameters
level (*int | str*) – The new level for the handler.

warning(*msg*, **args*, ***kwargs*)

Logs a message with the *WARNING* (page 12) level.

Additional arguments are interpreted the same way as *BurinLogger.log()* (page 28).

Parameters
msg (*str*) – The message to log.

5.2 BurinLoggerAdapter

Almost all of the methods of the logger adapter mirror the underlying logger or simply delegate directly to it. The only unique method to the adapter is *BurinLoggerAdapter.process()* (page 32) which is called automatically when a log call is made. This can be overridden though to customise an adapter.

class burin.**BurinLoggerAdapter**(*logger*, *extra=None*)

An adapter for easily passing contextual information in logging events.

Note: This differs slightly from the standard libraries logging.LoggerAdapter. Primarily the *extra* dictionary that is part of this adapter is merged with any *extra* dictionary that is part of each logging call instead of overwriting it.

This allows for more use cases and better nesting functionality. Also the *manager* property and *_log* method are not part of this class as they were unused.

Note: Almost all of the properties and non-logging methods of this class simply delegate to the underlying logger instance.

Using an adapter can simplify logging calls where specific contextual information would repeatedly need to be added to logging calls by instead automatically adding that contextual information for every logging event.

This is supported by essentially providing an *extra* value once when instantiating an adapter which is then added every time a logging method is called through the adapter.

The *extra* mapping is added to the log record's *__dict__*, so this can allow custom fields in the format string used in a *BurinFormatter* (page 51) to be populated with these values.

Initialization requires a *logger* and the optional *extra* mapping.

Parameters

- **logger** (BurinLogger (page 24)) – The logger to use when calling logging events.
- **extra** (*dict{str: Any}*) – The mapping to be added to the log record's *__dict__*.

property msgStyle

Gets or sets the *BurinLogger.msgStyle* (page 25) of the underlying logger.

See *BurinLogger.msgStyle* (page 25) for more information about how this is used and what it can be set to.

Note: This will raise a *FactoryError* (page 63) if it is set to a value that doesn't match with any log record factory.

---

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

Logs a message with the *CRITICAL* (page 12) level.

Additional arguments are interpreted the same way as *BurinLoggerAdapter.log()* (page 31).

Parameters

msg (*str*) – The message to log.

debug(*msg*, **args*, ***kwargs*)

Logs a message with the *DEBUG* (page 12) level.

Additional arguments are interpreted the same way as *BurinLoggerAdapter.log()* (page 31).

Parameters
msg (*str*) – The message to log.

error(*msg*, **args*, ***kwargs*)

Logs a message with the *ERROR* (page 12) level.

Additional arguments are interpreted the same way as *BurinLoggerAdapter.log()* (page 31).

Parameters
msg (*str*) – The message to log.

exception(*msg*, **args*, *exc_info=True*, ***kwargs*)

Logs a message with the *ERROR* (page 12) level with exception info.

This should normally be called only within an exception handler.

Additional arguments are interpreted the same way as *BurinLoggerAdapter.log()* (page 31).

Parameters
msg (*str*) – The message to log.

get_effective_level()

Gets the effective log level for the underlying logger.

Returns
The effective log level for the underlying logger.

Return type
int

has_handlers()

Checks if there are any available handlers for the underlying logger.

Returns
Whether the underlying logger has any available handlers.

Return type
bool

info(*msg*, **args*, ***kwargs*)

Logs a message with the *INFO* (page 12) level.

Additional arguments are interpreted the same way as *BurinLoggerAdapter.log()* (page 31).

Parameters
msg (*str*) – The message to log.

is_enabled_for(*level*)

Checks if the underlying logger is enabled for the specified *level*.

Parameters
level (*int | str*) – The level to check on the underlying logger.

Returns
If the underlying logger is enabled for *level*.

Return type
bool

log (*level*, *msg*, **args*, ***kwargs*)

Logs a message with the specified *level*.

Note: Unlike `BurinLogger.log()` (page 28) all keyword arguments (like *exc_info*, *extra*, *stack_info*, and *stacklevel*) are just handled as *kwargs* instead of specific arguments. This allows for more flexibility in any subclassed adapters as all of the *kwargs* are passed for processing as just a dictionary.

This will call `BurinLoggerAdapter.process()` (page 32) to add the *extra* values of this adapter with the logging call before calling the underlying logger.

Everything is passed to the underlying logger, so for more information about how it can be used and additional arguments please see `BurinLogger.log()` (page 28).

Parameters

- **level** (*int | str*) – The level to log the message at.
- **msg** (*str*) – The message to log.

process (*msg*, *kwargs*)

Processes the log event for the adapter.

This will add the *extra* values passed to the adapter when it was instantiated to the log event kwargs. If another *extra* dictionary was passed as part of the logging event then this will merge the *extra* values with the ones from the log event call taking precedence.

This can be overridden to provide other types of processing or customised adapters. The log *msg* and all *kwargs* from the logging call are passed in.

Parameters

- **msg** (*str*) – The log message.
- **kwargs** (*dict{str: Any}*) – All keyword arguments that were passed with the logging call.

Returns
The log message and keyword arguments to be sent to the underlying logger after processing.

Return type
tuple(str, dict{str: Any})

set_level (*level*)

Sets the level of the underlying logger.

Parameters
level (*int | str*) – The new level for the handler.

warning (*msg*, **args*, ***kwargs*)

Logs a message with the `WARNING` (page 12) level.

Additional arguments are interpreted the same way as `BurinLoggerAdapter.log()` (page 31).

Parameters
msg (*str*) – The message to log.

CHAPTER

SIX

HANDLERS

Handlers are responsible for emitting the log record to specific destination. All handlers within Burin are derived from the *BurinHandler* (page 38) class.

One feature of all Burin handlers is the ability to set the handler's log level when it is created. Every handler class has an optional `level` parameter for this so *BurinHandler.set_level()* (page 40) doesn't need to be called seperately. The default level for every handler is *NOTSET* (page 12).

Note: Even though many handlers in Burin inherit from handlers within the standard `logging` package, they cannot be used interchangeably.

Using `logging` handlers with Burin or Burin handlers with `logging` will cause issues and may result in exceptions or lost logs.

Note: Only methods defined within each Burin handler class are documented here. All handlers inherit from the *BurinHandler* (page 38) class and will also mention in their description if they inherit from any other handlers.

If a handler inherits from the `logging` package then methods that have not been changed are not documented here.

Additionally all methods of handler classes with an *underscore_separated* name also have a *camelCase* alias name which matches the names used in the standard `logging` library.

Below is a list of all handlers available within Burin. After that detailed descriptions of each handler is provided.

BurinBaseRotatingHandler (page 34)	Base class for handlers that rotate log files.
BurinBufferingHandler (page 35)	A handler that stores log records in a buffer.
BurinDatagramHandler (page 36)	A handler that writes log records to a datagram socket.
BurinFileHandler (page 37)	A handler for writing log records to a file.
BurinHandler (page 38)	Handlers emit logging events to specific destinations.
BurinHTTPHandler (page 40)	A handler that can send log records over HTTP to a Web server.
BurinMemoryHandler (page 41)	A handler which buffers log records in memory.
BurinNTEventLogHandler (page 42)	A handler which sends events to Windows NT Event Log.
BurinNullHandler (page 42)	A handler that doesn't do any formatting or output any log records.
BurinQueueHandler (page 43)	A handler that supports logging messages to a queue.
BurinQueueListener (page 43)	Listens for and processes log records queued by *BurinQueueHandler* (page 43).
BurinRotatingFileHandler (page 44)	A handler that rotates the file when it reaches a certain size.
BurinSMTPHandler (page 45)	A handler that can send emails over SMTP for logging events.
BurinSocketHandler (page 46)	A handler that writes pickled log records to a network socket.
BurinStreamHandler (page 47)	A handler that writes log records to a stream.
BurinSyslogHandler (page 47)	A handler that supports sending log records to a local or remote syslog.
BurinTimedRotatingFileHandler (page 48)	A handler that rotates the file at specific intervals.
BurinWatchedFileHandler (page 50)	A handler that watches for changes to the file.

6.1 BurinBaseRotatingHandler

This is the base rotating handler which can be used by any handlers that need to rotate files. This should not be used directly but instead can be inherited from to create custom handlers.

class burin.**BurinBaseRotatingHandler**(*filename*, *mode*, *encoding=None*, *delay=False*, *errors=None*, *level='NOTSET'*)

Base class for handlers that rotate log files.

This is derived from *BurinFileHandler* (page 37).

Note: This is a subclass of `logging.handlers.BaseRotatingHandler` and functions identically to it in normal use cases.

This should not be instantiated directly except within a subclass *__init__* method.

This will initialize the handler for outputting to a file.

Parameters

- **filename** (*str | pathlib.Path*) – The filename or path to write to.
- **mode** (*str*) – The mode that the file is opened with.
- **encoding** (*str*) – The text encoding to open the file with.

- **delay** (*bool*) – Whether to delay opening the file until the first record is emitted. (Default = **False**)
- **errors** (*str*) – Specifies how encoding errors are handled. See `open()` for information on the appropriate values.
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

do_rollover()

This method should perform the rotation of the file.

This should be implemented within a subclass and will only raise a `NotImplementedError` in this base class.

Raises
NotImplementedError – As this is not implemented in the base class.

emit(*record*)

Emits the record to the file.

This will check if the file should be rotated by calling *should_rollover* and if that returns **True** it calls *do_rollover* to perform the actual rotation.

Parameters
record (`BurinLogRecord` (page 55)) – The log record to emit.

should_rollover(*record*)

This method should check if the rotation of the file should be done.

This should be implemented within a subclass and will only raise a `NotImplementedError` in this base class.

Note: The *record* parameter is needed for the *BurinRotatingFileHandler* (page 44), so to ensure the signature is the same all subclasses should include it whether they use it or not.

Parameters
record (`BurinLogRecord` (page 55)) – The log record. (Not used for all subclasses)

Raises
NotImplementedError – As this is not implemented in the base class.

6.2 BurinBufferingHandler

This is a base buffering handler which can be used to create other handlers which requiring a buffering pattern. This should not be used directly but instead can be inherited from to create custom handlers.

class burin.**BurinBufferingHandler**(*capacity*, *level='NOTSET'*)

A handler that stores log records in a buffer.

Note: This is a subclass of `logging.handlers.BufferingHandler` and functions identically to it in normal use cases.

Each time a record is added to the buffer a check is done to see if the buffer should be flushed.

This class is intended to be subclassed by other handlers that need to use a buffering pattern and should not be instantiated directly except within a subclass *__init__* method.

The buffer will flush once *capacity* number of records are stored.

Parameters

- **capacity** (*int*) – The number of log records to hold in the buffer before flushing.
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

close()

Closes the handler and flush the buffer.

6.3 BurinDatagramHandler

This handler can be used to send logs through a datagram socket to another Python application.

class burin.**BurinDatagramHandler**(*host*, *port*, *pickleProtocol=4*, *level='NOTSET'*)

A handler that writes log records to a datagram socket.

The pickled data that is sent is just of the log records attribute dictionary (*__dict__*) so it can process the event in any way it needs and doesn't require Burin to be installed.

This is derived from *BurinSocketHandler* (page 46).

Note: The default pickle protocol version used in *BurinSocketHandler* (page 46) is different than what is used in logging.handlers.SocketHandler.

Since this is a subclass of the socket handler it is also impacted.

This should only cause issues if the receiving Python version is much older. However if needed the pickle protocol version used can be changed with the *pickleProtocol* parameter.

The *make_log_record()* (page 20) function can be used on the receiving end to recreate the log record from the pickled data if desired.

The *host* and *port* will set address and family of socket used.

If *port* is specified as **None** then the socket family will be socket.AF_UNIX; otherwise the socket family is socket.AF_INET.

Parameters

- **host** (*str*) – The address of the host to communicate with.
- **port** (*int*) – The port to communicate on.
- **pickleProtocol** (*int*) – The pickle protocol version to use. (Default = 4)
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

make_socket()

Makes the UDP (socket.SOCK_DGRAM) socket.

The socket family will be either socket.AF_UNIX or socket.AF_INET depending on the address that was passed in during initialization.

Returns

The UDP socket.

Return type
socket.socket

send(*msg*)

Sends the pickled log record through the socket.

This will try to create the socket first if it hasn't been created yet.

Parameters
msg (*str*) – The pickled string of the log record.

6.4 BurinFileHandler

This handler allows for simply writing logs out to a file.

class burin.**BurinFileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*, *level='NOTSET'*)

A handler for writing log records to a file.

This is derived from *BurinStreamHandler* (page 47).

Note: This is a subclass of `logging.FileHandler` and functions identically to it in normal use cases.

This will setup the handler using the absolute file path.

The file that is opened will grow indefinitely while being logged to. If this isn't desired consider using the *BurinRotatingFileHandler* (page 44) or *BurinTimedRotatingFileHandler* (page 48) instead.

Parameters

- **filename** (*str | pathlib.Path*) – The filename or path to write to.
- **mode** (*str*) – The mode that the file is opened with. (Default = 'a')
- **encoding** (*str*) – The text encoding to open the file with.
- **delay** (*bool*) – Whether to delay opening the file until the first record is emitted. (Default = **False**)
- **errors** (*str*) – Specifies how encoding errors are handled. See `open()` for information on the appropriate values.
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

close()

Flushes and closes the file.

emit(*record*)

Emits a log record to the file.

Parameters
record (BurinLogRecord (page 55)) – The log record to emit.

6.5 BurinHandler

This is the base handler class that all other handlers in Burin are derived from. This should not be used directly but instead can be inherited from to create custom handlers.

class burin.**BurinHandler**(*level='NOTSET'*)

Handlers emit logging events to specific destinations.

Note: This functions almost identically to `logging.Handler` but has some minor changes that allow it to work within Burin. These changes shouldn't impact normal usage when compared with the standard `logging` library.

This is not a subclass of `logging.Handler` and is not usable with the standard library `logging` module.

This is the base handler class for Burin and should not be used directly, but instead can be subclassed to create other handlers that work with Burin.

This will setup the basic instance values for the handler.

Typically this should be called within any subclasses *__init__* method to ensure all required handler instance attributes are created.

Parameters

level (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

property name

The name of the handler.

acquire()

Acquires the handlers internal thread lock.

It is recommended to use a handler's lock in a context manager using the **with** statement. The lock is simply accessible as `BurinHandler.lock` on any handler instance.

The *BurinHandler.acquire()* (page 38) and *BurinHandler.release()* (page 39) methods are primarily provided for improved compatibility with the standard library `logging.Handler`.

close()

Cleans up the handler.

This simply removes the handler from an internal library reference map, but any subclasses should ensure this is called in any overridden *close()* methods to ensure the reference to the handler is cleaned up.

create_lock()

Acquires a re-entrant lock for the handler for threading protection.

The lock is available through the instance `BurinHandler.lock` attribute or it can be used with *BurinHandler.acquire()* (page 38) and *BurinHandler.release()* (page 39).

This lock can then be used by subclasses to serialize access to I/O or any other places where protection of the instance across threads may be needed.

This also registers the handler to reinitialize the lock after a fork as otherwise it could prevent logging through the handler if fork is called while the lock is held.

flush()

Meant to ensure that all logging output is flushed.

This is intended to be implemented within subclasses as needed; this method on the base class does not do anything.

format(*record*)

Formats the received log record.

If the handler doesn't have a formatter a basic default formatter is used.

Parameters
record (BurinLogRecord (page 55)) – The log record to be formatted.

Returns
The formatted text of the log record.

Return type
str

handle(*record*)

Process the log record and possibly emit it.

This will check any filters that have been added to the handler and emit the record if no filters return **False**.

If the record passes all filters then the instance of the record that was emitted will be returned.

Note: In Python 3.12 the ability for this to return a record was added to the standard library; it is supported here for all versions of Python compatible with Burin (including versions below 3.12).

Parameters
record (BurinLogRecord (page 55)) – The log record to process.

Returns
An instance of the record that was emitted, or **False** if the record was not emitted.

Return type
BurinLogRecord (page 55) | bool

handle_error(*record*)

Handles errors which may occur during an *emit()* call.

This should be called from subclasses when an exception is encountered during an *emit()* call.

If `raiseExceptions` is **False** then the error will be silently ignored. This can be useful for a logging system as most users would be more concerned with application errors vs logging library errors.

However if `raiseExceptions` is **True** then information about the error will be output to `sys.stderr`.

Parameters
record (BurinLogRecord (page 55)) – The log record that was being processed when the error occurred.

release()

Releases the handler's internal thread lock.

It is recommended to use a handler's lock in a context manager using the **with** statement. The lock is simply accessible as `BurinHandler.lock` on any handler instance.

The *BurinHandler.acquire()* (page 38) and *BurinHandler.release()* (page 39) methods are primarily provided for improved compatibility with the standard library `logging.Handler`.

set_formatter(*fmt*)

Sets the formatter to be used by this handler.

Parameters
fmt (BurinFormatter (page 51)) – The formatter to use.

set_level (*level*)

Sets the logging level of this handler.

Parameters
level (*int | str*) – The new level for the handler.

6.6 BurinHTTPHandler

This handler can send logs to another service using HTTP.

class burin.**BurinHTTPHandler** (*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*, *level='NOTSET'*)

A handler that can send log records over HTTP to a Web server.

Note: This is a subclass of `logging.HTTPHandler` and functions identically to it in normal use cases.

Note: This has the *BurinHTTPHandler.get_connection()* (page 40) method (also aliased as `BurinHTTPHandler.getConnection()`); this was added to the standard library in Python 3.9 but is available here for all Python versions supported by Burin.

This will setup the handler and do some basic checks of parameters.

Only 'GET' or 'POST' are allowed as *method*. Also *context* must be **None** if *secure* is **False**.

Parameters

- **host** (*str*) – The host to connect to; this can be in the form of 'host:port' if non-standard HTTP/HTTPS ports are to be used.
- **url** (*str*) – The URL path on the host to use.
- **method** (*str*) – The HTTP method to use for the request. This must be either 'GET' or 'POST'. (Default = 'GET')
- **secure** (*bool*) – Whether to use HTTPS or not. (Default = **False**)
- **credentials** (*tuple(str, str)*) – If authentication is needed for the host then this should be a 2-tuple of (username, password). This will be placed into an HTTP 'Authorization' header for Basic Authentication support. If this is used you should also use *secure*=**True** so that the username and password are not sent in cleartext to the host.
- **context** (*ssl.SSLContext*) – A `ssl.SSLContext` instance to configured settings for an HTTPS connection. This must be **None** if *secure*=**False**.
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

Raises
ValueError – If *method* is not 'GET' or 'POST', or *context* is not **None** and *secure*=**False**.

get_connection (*host*, *secure*)

Gets the HTTP or HTTPS connection.

This can be overridden to change how the connection is created; for example if a proxy is required.

Parameters

- **host** (*str*) – The host to connect to.
- **secure** (*bool*) – Whether to use HTTPS or not.

Returns
The connection object.

Return type
http.client.HTTPConnection | http.client.HTTPSConnection

6.7 BurinMemoryHandler

This handler can buffer logs in memory until a specified capacity is reached.

class burin.**BurinMemoryHandler**(*capacity*, *flushLevel='ERROR'*, *target=None*, *flushOnClose=True*, *level='NOTSET'*)

A handler which buffers log records in memory.

This is derived from *BurinBufferingHandler* (page 35).

Note: This is a subclass of `logging.handlers.MemoryHandler` and functions identically to it in normal use cases.

This handler will flush when the buffer reaches the specified *capacity* or when a record of the specified *flushLevel* or above is emitted.

The *target* handler will be called when this flushes its buffer.

Parameters

- **capacity** (*int*) – The number of log records to hold in the buffer before flushing.
- **flushLevel** (*int / str*) – If a log record of this level is put in the buffer it will immediately flush the whole buffer. (Default = 'ERROR')
- **target** (BurinHandler (page 38)) – The handler which is called with the log records when the buffer is flushed.
- **flushOnClose** (*bool*) – Whether the buffer should be flushed when the handler is closed. (Default = **True**)
- **level** (*int / str*) – The logging level of the handler. (Default = 'NOTSET')

close()

Closes the handler.

This will also flush the buffer if *flushOnClose* was **True** when the handler was initialized.

6.8 BurinNTEventLogHandler

This handler can log to the Windows event log; this requires the *pywin32* package.

class burin.**BurinNTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*, *level='NOTSET'*)

A handler which sends events to Windows NT Event Log.

Note: This is a subclass of `logging.handlers.NTEventLogHandler` and functions identically to it in normal use cases.

To use this handler you must be on a Windows system and have the *pywin32* package installed.

This sets the application name and allows using a specific dll.

During initialization this will try to import the `win32evtlogutil` and `win32evtlog` modules from the *pywin32* package. If this fails it will print a message to *stdout* and the handler that is created will not log anything.

A registry entry for the *appname* will be created. Also if *dllname* is **None** then *win32service.pyd* is used. This can cause the resulting event logs to be quite large, so you can specify a different *dllname* with the message definitions you want to use.

Parameters

- **appname** (*str*) – The name of the application which will be added to the registry.
- **dllname** (*str*) – Specify a dll to use other than *win32service.pyd*.
- **logtype** (*str*) – The log type used to register the event logs.
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

close()

Closes the handler.

6.9 BurinNullHandler

This handler doesn't do anything, but can be used to ensure a logger has a configured handler that doesn't actually output to anything (not even *sys.stderr*). This may be useful in libraries where you want to use Burin if it's available, but want to let the application configure the output handlers.

class burin.**BurinNullHandler**(*level='NOTSET'*)

A handler that doesn't do any formatting or output any log records.

This is essentially meant as a no-op handler to be used when you need a handler to be attached to a logger, but don't want any output.

create_lock()

Does not actually create a lock; this will set *self.lock* to **None**.

emit(*record*)

Does not emit anything.

Parameters

record (BurinLogRecord (page 55)) – This is not emitted to anything; it is only here so the signature matches other handlers.

handle(*record*)

Does no processing or handling of the record.

Parameters

record (BurinLogRecord (page 55)) – This is not processed in any way; it is only here so the signature matches other handlers.

6.10 BurinQueueHandler

This handler adds all logs to a queue which a *BurinQueueListener* (page 43) can then process. This can be useful in a multiprocess application to have one process handle all of the actual logging (and I/O involved) while the others just add to the queue.

class burin.**BurinQueueHandler**(*queue*, *level='NOTSET'*)

A handler that supports logging messages to a queue.

Note: This is a subclass of `logging.handlers.QueueHandler` and functions identically to it in normal use cases.

This can be used along with *BurinQueueListener* (page 43) to allow one process or thread in a program handle logging output which may consist of slow operations like file writing or sending emails. This can be useful in Web or service applications where responsiveness is important in worker processes and threads.

Logs records are added to the queue by each *BurinQueueHandler* (page 43) and then processed and output by the *BurinQueueListener* (page 43).

This will initialize the handler and set the queue to use.

Parameters

- **queue** (*queue.Queue | queue.SimpleQueue | multiprocessing.Queue*) – This must be any queue like object; it does not need to support the task tracking API.
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET")

6.11 BurinQueueListener

This can be paired with *BurinQueueHandler* (page 43) to have one process for a queue of logs which multiple handlers add to.

class burin.**BurinQueueListener**(*queue*, **handlers*, *respect_handler_level=False*)

Listens for and processes log records queued by *BurinQueueHandler* (page 43).

Note: This is a subclass of `logging.handlers.QueueListener` and is just a stub class to provide a matching listener for *BurinQueueHandler* (page 43).

This can be used along with *BurinQueueHandler* (page 43) so that log processing and output, which may consist of slow operations like file writing or sending emails, can be done outside of worker processes or threads where responsiveness may be important.

6.12 BurinRotatingFileHandler

This handler can automatically rotate a log file when it reaches a specific size.

class burin.**BurinRotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*, *errors=None*, *level='NOTSET'*)

A handler that rotates the file when it reaches a certain size.

This is derived from *BurinBaseRotatingHandler* (page 34).

Note: This is a subclass of `logging.handlers.RotatingFileHandler` and functions identically to it in normal use cases.

The file is rotated once it reaches a specific size. A limit can also be placed on how many rotated files are kept.

This will initialize the handler to write to the file.

The file will be rotated when it reaches *maxBytes* size. The number of rotated files to keep is set by *backupCount*.

When the files are rotated a number is appended to the filename in the order '.1', '.2', '.3', etc. until the *backupCount* is reached. So a *backupCount* of 5 will result in 5 files other than the active log file being kept up to '*filename*.5'. Once *backupCount* is reached the next time a rotate happens the oldest file will be removed.

The active log file set with *filename* is always the file being written to.

Parameters

- **filename** (*str | pathlib.Path*) – The filename or path to write to.
- **mode** (*str*) – The mode that the file is opened with. This should be 'a' in almost all use cases. If 'w' is in the mode and *maxBytes* != 0 then it will be replaced with 'a' as otherwise the file will be truncated every time the program runs which is counter-intuitive to a rotating log file. (Default = 'a')
- **maxBytes** (*int*) – The maximum size (in bytes) the file can be before a rotation happens. The rotation happens before an emit so the file should never go above this size. If this is 0 then the file will never be rotated. (Default = 0)
- **backupCount** (*int*) – How many rotated log files to keep. If this is 0 then the file will not be rotated. (Default = 0)
- **encoding** (*str*) – The text encoding to open the file with.
- **delay** (*bool*) – Whether to delay opening the file until the first record is emitted. (Default = **False**)
- **errors** (*str*) – Specifies how encoding errors are handled. See `open()` for information on the appropriate values.
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

6.13 BurinSMTPHandler

This handler can send logs through email using a SMTP server.

class burin.**BurinSMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=5.0*, *level='NOTSET'*)

A handler that can send emails over SMTP for logging events.

Note: This is a subclass of `logging.handlers.SMTPHandler` and functions identically to it in normal use cases.

This requires an email server that you have permission to send emails through; it cannot be used standalone to send directly to a receiving server.

This will initialize the handler for sending emails.

The standard SMTP port from `smtplib.SMTP_PORT` is used by default; if you need to use a non-standard port then *mailhost* must be a tuple in the form of *(host, port)*.

You can send to multiple recipients by passing a list of addresses to *toaddrs*.

If your SMTP server requires authentication then *credentials* should be a list or tuple in the form of *(username, password)*. If you are sending credentials then *secure* should not be **None** to prevent them being sent unencrypted.

Parameters

- **mailhost** (*str | tuple(str, int)*) – The SMTP server to connect to and send mail through. By default the standard SMTP port is used; if you need to use a custom port this should be a tuple in the form of *(host, port)*.
- **fromaddr** (*str*) – The address that the email is sent from.
- **toaddrs** (*list[str] | str*) – The recipient email addresses. This can be a single address or a list of multiple addresses.
- **subject** (*str*) – The subject line of the email.
- **credentials** (*tuple(str, str)*) – If the SMTP server requires authentication you can pass a tuple here in the form *(username, password)*.
- **secure** (*tuple*) – If *credentials* is not none then can be set to a tuple to enable encryption for the connection to the SMTP server. The tuple can follow one of three forms, an empty tuple *()*, a single value tuple with the name of a keyfile *(keyfile,)*, or a 2-value tuple with the names of a keyfile and certificate file *(keyfile, certificatefile)*. This is then passed to `smtplib.SMTP.starttls()`.
- **timeout** (*float | int*) – A timeout (in seconds) for communications with the SMTP server.
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

6.14 BurinSocketHandler

This handler can send pickled log records through a socket to another Python application.

class burin.**BurinSocketHandler**(*host*, *port*, *pickleProtocol=4*, *level='NOTSET'*)

A handler that writes pickled log records to a network socket.

Note: This is a subclass of `logging.handlers.SocketHandler` but has a change that may be incompatible depending on the receiver's Python version.

The default pickle protocol version used is **4** instead of **1**; this can be configured though by the *pickleProtocol* parameter which was added.

The pickled data that is sent is just of the log records attribute dictionary (*__dict__*) so it can process the event in any way it needs and doesn't require Burin to be installed.

The *make_log_record()* (page 20) function can be used on the receiving end to recreate the log record from the pickled data if desired.

This will set the *host* and *port* for the socket to connect to.

Parameters

- **host** (*str*) – The address of the host to communicate with.
- **port** (*int*) – The port to communicate on.
- **pickleProtocol** (*int*) – The pickle protocol version to use. (Default = 4)
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

close()

Closes and handler and the socket.

handle_error(*record*)

Handles errors which may occur during an *emit()* call.

This wile close the socket if *self.closeOnError*=**True*; it then calls *BurinHandler.handle_error()* (page 39) to continue with the error handling.

Parameters

record (BurinLogRecord (page 55)) – The log record that was being processed when the error occurred.

make_pickle(*record*)

Pickles the record in a binary format.

This prepares the record for transmission across the socket.

Parameters

record (BurinLogRecord (page 55)) – The log record to pickle.

Returns

The pickled representation of the record.

Return type

bytes

6.15 BurinStreamHandler

This handler can write logs to an I/O stream.

class burin.**BurinStreamHandler**(*stream=None*, *level='NOTSET'*)

A handler that writes log records to a stream.

Note: This is a subclass of `logging.StreamHandler` and functions identically to it in normal use cases.

Note: This handler will not close the stream it is writing to as `sys.stdout` and `sys.stderr` are commonly used.

This initializes the handler and sets the *stream* to use.

If *stream* is **None** then `sys.stderr` is used by default.

Parameters

- **stream** (*io.TextIOBase*) – The stream to log to. If this is **None** then `sys.stderr` is used.
- **level** (*int / str*) – The logging level of the handler. (Default = 'NOTSET')

set_stream(*stream*)

Sets the StreamHandler's stream to the specified value, if it is different.

Returns the old stream, if the stream was changed, or None if it wasn't.

6.16 BurinSyslogHandler

This handler can write logs out using Syslog.

class burin.**BurinSyslogHandler**(*address=('localhost', 514)*, *facility=1*, *socktype=None*, *level='NOTSET'*)

A handler that supports sending log records to a local or remote syslog.

Note: This is a subclass of `logging.handlers.SysLogHandler` and functions identically to it in normal use cases.

Unlike the standard library handler the 'l' in 'Syslog' of the class name is not capitalized so this class better matches the actual 'Syslog' name.

This initializes the handler and sets it for sending to syslog.

By default the handler will try to use a local syslog through UDP port 514; to change this *address* must be set as a tuple in the form *(host, port)*.

By default a UDP connection is created; if TCP is needed ensure *socktype* is set to `socket.SOCK_STREAM`.

Parameters

- **address** (*tuple(str, int)*) – The address to connect to syslog at. This should be a tuple in the form of *(host, port)*. (Default = ('localhost', 514))

- **facility** (*int*) – The syslog facility to use. These are available as class attributes on the handler to simplify usage. (Default = 1 (LOG_USER))
- **socktype** (*int*) – The socket type to use for the connection to syslog. By default a `socket.SOCK_DGRAM` socket is used if this is **None**; for TCP connections specify `socket.SOCK_STREAM`.
- **level** (*int / str*) – The logging level of the handler. (Default = 'NOTSET')

close()

Closes the handler and the syslog socket.

6.17 BurinTimedRotatingFileHandler

This handler can rotate log files based on a timing pattern.

class burin.**BurinTimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*, *errors=None*, *level='NOTSET'*)

A handler that rotates the file at specific intervals.

This is derived from *BurinBaseRotatingHandler* (page 34).

Note: This is a subclass of `logging.handlers.TimedRotatingFileHandler` and functions identically to it in normal use cases.

The file is rotated once at the specified interval. A limit can also be placed on how many rotated files are kept.

This will initialize the handler to write to the file.

The file will be rotated based on the *when*, *interval*, and *atTime* values. The number of rotated files to keep is set by *backupCount*.

when	Interval type	*atTime* usage
'S'	Seconds	Ignored
'M'	Minutes	Ignored
'H'	Hours	Ignored
'D'	Days	Ignored
'W0'-'W6'	*interval* ignored; Weekday (0 = Monday)	Time of the day to rotate
'MIDNIGHT'	*interval* ignored; Midnight or *atTime*	Time of the date to rotate

When the files are rotated a time and/or date is appended to the filename until the *backupCount* is reached. The `time.strftime()` format `%Y-%m-%d_%H-%M-%S` is used with later parts stripped off when not relevant for the rotation interval selected. Once *backupCount* is reached the next time a rotate happens the oldest file will be removed.

The rotation interval is calculated (during initialization) based on the last modification time of the log file, or the current time if the file doesn't exist, to determine when the next rotation will occur.

The active log file set with *filename* is always the file being written to.

Parameters

- **filename** (*str | pathlib.Path*) – The filename or path to write to.
- **when** (*str*) – The type of interval to use when calculating the rotation. Use the table above to see the available options and how they impact the rotation interval. (Default = 'h')
- **interval** (*int*) – The interval to use for the file rotation. Use the table above to see how this is used in determining the rotation interval. (Default = 1)
- **backupCount** (*int*) – How many rotated log files to keep. If this is 0 then the file will not be rotated. (Default = 0)
- **encoding** (*str*) – The text encoding to open the file with.
- **delay** (*bool*) – Whether to delay opening the file until the first record is emitted. (Default = **False**)
- **utc** (*bool*) – Whether to use UTC time or local time. (Default = **False**)
- **atTime** (*datetime.time*) – The time to use for weekday or 'midnight' (daily at set time) rotations. Use the table above to see how this is used in determining the rotation interval.
- **errors** (*str*) – Specifies how encoding errors are handled. See `open()` for information on the appropriate values.
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

should_rollover(*record*)

Determines if a rollover should occur.

Note: The *record* parameter is not used, it is included to keep the method signatures the same for all subclasses of *BurinBaseRotatingHandler* (page 34)

Note: In Python 3.11 `logging.handlers.TimedRotatingFileHandler.shouldRollover()` was changed to ensure that if the target is not currently a regular file the check is skipped and the next one is scheduled. Previously checks simply ran and failed repeatedly. This change is incorporated here for all versions of Python compatible with Burin (including versions below 3.11).

Parameters

record (BurinLogRecord (page 55)) – The log record. (Not used)

Returns

Whether a rollover is scheduled to occur.

Return type

bool

6.18 BurinWatchedFileHandler

This handler watches the file it is writing to and will close and reopen it automatically if it detects any changes.

class burin.**BurinWatchedFileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*, *level='NOTSET'*)

A handler that watches for changes to the file.

Note: This is a subclass of `logging.handlers.WatchedFileHandler` and functions identically to it in normal use cases.

If the file this is logging to changes it will close and then reopen the file.

This is intended for use on Unix/Linux systems and checks for device or inode changes. Such changes would occur if a program like *logrotate* was to rotate the file.

This should not be used on Windows and is not needed as log files are opened with exclusive locks and cannot be moved or renamed when in use.

This will setup the handler and stat the file.

Parameters

- **filename** (*str | pathlib.Path*) – The filename or path to write to.
- **mode** (*str*) – The mode that the file is opened with. (Default = 'a')
- **encoding** (*str*) – The text encoding to open the file with.
- **delay** (*bool*) – Whether to delay opening the file until the first record is emitted. (Default = **False**)
- **errors** (*str*) – Specifies how encoding errors are handled. See `open()` for information on the appropriate values.
- **level** (*int | str*) – The logging level of the handler. (Default = 'NOTSET')

emit(*record*)

Emits the record to the file.

This will check if the file needs to be reopened before writing to it.

Parameters

record (`BurinLogRecord` (page 55)) – The log record to emit.

CHAPTER

SEVEN

FORMATTERS

The purpose of a formatter is to convert a *BurinLogRecord* (page 55) into (typically) a textual representation of the logging event according to a specified format.

A *BurinFormatter* (page 51) should be set on every handler, but if a handler doesn't have one then a very simple formatter will be used instead.

If multiple logs need to be formatted in a batch for a custom handler then a *BurinBufferingFormatter* (page 54) can be used or a subclass of it can be created to meet those needs.

7.1 BurinFormatter

The *BurinFormatter* (page 51) is derived from `logging.Formatter` and should function identically in almost all use cases.

Note: All methods of the *BurinFormatter* (page 51) with an *underscore_separated* name also have a *camelCase* alias name which matches the names used in the standard `logging` library.

---

class burin.**BurinFormatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*, *, *defaults=None*)

Formatter for converting a log record for output.

Note: This is a subclass of `logging.Formatter` but has some minor changes (such as raising *FormatError* (page 63) instead of `ValueError`).

These changes shouldn't impact normal usage when compared with the standard `logging` library.

Formatters are responsible for converting a log record into (usually) a string which can then be output by a handler.

Below is the attributes of a log record that could be useful to log. Any of these can be added to the format string in whatever formatting style that is selected.

asctime
Time the log record was created in a human readable format.

created
Time the log record was created as returned by `time.time()`

filename
Filename from where the logging call was issued.

Note: This is only the filename part; for the whole path see *pathname*

funcName
Name of the function where the logging call was issued.

levelname
Text name for the logging level of the record.

levelno
Numeric value for logging level of the record.

lineno
Line number where the logging call was issued.

message
Log message as processed by `BurinLogRecord.get_message()` (page 56) method.

This is set on the record when `BurinFormatter.format()` (page 53) is called.

module
Module name where the logging call was issued.

msecs
Millisecond portion of the time when the log record was created.

name
Name of the logger that was called.

pathname
Full pathname of the source file where the logging call was issued.

process
Process Id

processName
Process name

relativeCreated
Time in milliseconds from when the log record was created since the Burin package was loaded.

taskName
Asyncio task name.

Note: In Python 3.12 this was added to the standard library; it is supported here for all versions of Python compatible with Burin (including versions below 3.12).

However; this will always be **None** in Python 3.7 as Task names were added in Python 3.8.

thread
Thread Id

threadName
Thread name

Note: Some of the attributes may not have values depending on the Python implementation used or the values of `logMultiprocessing`, `logProcesses`, and `logThreads`.

Note: There are other attributes of log records which are part of its operation and should not need to be formatted. It is recommended to stick to the list above.

The formatter will use the format string and specified *style*.

You can use datefmt to change how the time and date are formatted, otherwise the default is an ISO8601-like format.

If no format string is provided a simple style-dependent default is used which just includes the message from the log record.

Note: In Python 3.8 *validate* was added to the standard `logging.Formatter`; it is supported here for all versions of Python compatible with Burin (including versions below 3.8).

In Python 3.10 *defaults* was added to the standard `logging.Formatter`; it is supported here for all versions of Python compatible with Burin (including versions below 3.10).

Parameters

- **fmt** (*str*) – The format string to use when formatting a log record. If this is **None** then a default style-specific format string will be used that has just the log message.
- **datefmt** (*str*) – The date/time format to use (as accepted by `time.strftime()`). If this is **None** then a default format similar to the ISO8601 standard is used.
- **style** (*str*) – The type of formatting to use for the format string. Possible values are '%' for %-formatting, '{' for `str.format()` formatting, and '$' for `string.Template` formatting. (Default = '%')
- **validate** (*bool*) – Whether the format should be validated against the style to protect again misconfiguration. (Default = **True**)
- **defaults** (*dict{str: Any}*) – A dictionary that provides default values for custom fields. This is a keyword only argument and cannot be passed as a positional argument.

Raises

FormatError (page 63) – If there are errors with the *format* or *style*, or if *validate* is **True** and validation fails.

format(*record*)

Format the record as text.

The record's attribute dictionary is used for the string formatting operation.

Before the formatting occurs some other steps are taken such as calling *BurinLogRecord.get_message()* (page 56) to get the complete log message, any time formatting that may be needed, and exception formatting if necessary.

Parameters

record (BurinLogRecord (page 55)) – The log record to format.

Returns

The log record formatted to text.

Return type

str

format_time(*record*, *datefmt=None*)

Gets the creation time of the specified log record as formatted text.

This should be called by the formatter itself within *BurinFormatter.format()* (page 53); it is separated here to simplify overriding how the time is formatted.

Note: In Python 3.9 this method was changed on the standard `logging.Formatter` so that the class attribute *default_msec_format* is optional. This is supported here for all versions of Python compatible with Burin (including versions below 3.9).

Parameters

- **record** (BurinLogRecord (page 55)) – The log record to get the time from.
- **datefmt** (*str*) – The date/time format to use (as accepted by `time.strftime()`). If **None** then the *datefmt* passed in during initialization of the *BurinFormatter* (page 51) instance is used.

Returns

The formatted date and time of the log record.

Return type

str

7.2 BurinBufferingFormatter

This cannot be used by any built-in handlers but provides a class that can be used for formatting a batch of log records at once if needed by a custom handler.

class burin.**BurinBufferingFormatter**(*linefmt=None*)

A formatter that can be used for formatting multiple records in a batch.

Note: This is a subclass of *logging.BufferingFormatter* and functions identically to it in normal use cases.

This will set a formatter to use for every record.

If no formatter is set then a default formatter is used.

Parameters

linefmt (BurinFormatter (page 51)) – The formatter to use. If this is **None** then a default formatter will be used. (Default = **None**)

CHAPTER

EIGHT

LOG RECORDS

A log record represents a logging event and all of the values associated with that event.

When a logger is processing a logging event it will create a new log record, the class used for creating the record is referred to as a log record factory.

Unlike the standard `logging` package Burin allows for multiple log record factories to be set at once. The factory that is used can be set on a per logger basis using the *`BurinLogger.msgStyle`* (page 25) property; this should be the `factoryKey` of the record.

The built-in log record factories for Burin are focused on allowing different styles of deferred formatting which is demonstrated in the *Deferred Formatting Styles* (page 8) section.

Custom log record factories can be added though and offer a lot flexibility in how a log message is processed. An example of this is demonstrated in the *Customisable Log Records* (page 8) section.

Note: Only methods defined within each Burin log record class are documented here. All log records inherit from the *`BurinLogRecord`* (page 55) class.

All methods of the log record classes with an *underscore_separated* name also have a *camelCase* alias name which matches the names used in the standard `logging` library.

8.1 BurinLogRecord

This is the base log record; and is not used as a log record factory. It is meant to be subclassed by other log records. No formatting is done to the message.

All other Burin log record classes are derived from this class.

class burin.**BurinLogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*, ***kwargs*)

Represents all of the values of a logging event.

Note: Unlike the builtin `logging.LogRecord` this does not perform any formatting of the log message. It is instead intended to just be a base class to be inherited from.

The *`BurinPercentLogRecord`* (page 57) instead provides the same printf (% style) formatting of the Python builtin LogRecord.

Note: In Python 3.12 the *taskName* attribute was added to the standard `logging.LogRecord` class; it is supported here for all versions of Python compatible with Burin (including versions below 3.12).

However; names were added to `asyncio.Task` objects in Python 3.8, so in Python 3.7 the *taskName* attribute on a log record will always be **None**.

Custom log record factories that are created should inherit from this and typically only override the *BurinLogRecord.get_message()* (page 56) method.

This initializes the log record and stores all relevant values.

Unlike the standard library `logging.LogRecord` this also stores all extra *kwargs* that were not used in the logging call. These can then be used later when formatting the log message.

Parameters

- **name** (*str*) – The name of the logger that was called.
- **level** (*int*) – The level for the log message.
- **pathname** (*str*) – The full pathname of the file where the logging call was made.
- **lineno** (*int*) – The line number of where the logging call was made.
- **msg** (*str*) – The logging message.
- **args** (*tuple(Any) | None*) – Additional positional arguments passed with the logging call.
- **exc_info** (*tuple(type, Exception, traceback)*) – Exception information related to the logging call.
- **func** (*str*) – The name of the function where the logging call was made.
- **sinfo** (*str*) – Text of the stack information from where the logging call was made.

get_message()

This returns the log message.

This should be overridden in subclasses to provide additional formatting or other modifications to the log message.

Returns
The log message.

Return type
str

8.2 BurinBraceLogRecord

This log record can be used for `str.format()` style formatting.

class burin.**BurinBraceLogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*, ***kwargs*)

A log record that will be formatted in `str.format()` ({ style).

This allows for deferred formatting using positional and/or keyword arguments that are passed in during log record creation.

This is derived from *BurinLogRecord* (page 55).

factoryKey = '{'

This is the key used for the class as a log record factory. This is updated automatically when the class is set using *set_log_record_factory()* (page 20).

get_message()

This formats the log message.

All additional *args* and *kwargs* that were part of the log record creation are used for the formatting of the log message.

Returns
The formatted log message.

Return type
str

8.3 BurinDollarLogRecord

This log record can be used for `string.Template` style formatting.

class burin.**BurinDollarLogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*, ***kwargs*)

A log record that will be formatted in `string.Template` ($ style).

This allows for deferred formatting using keyword arguments that are passed in during log record creation.

This is derived from *BurinLogRecord* (page 55).

factoryKey = '$'

This is the key used for the class as a log record factory. This is updated automatically when the class is set using *set_log_record_factory()* (page 20).

get_message()

This formats the log message.

All additional *kwargs* that were part of the log record creation are used for the formatting of the log message.

`string.Template.safe_substitute()` so no exceptions are raised if keys and format placeholders don't all match.

Returns
The formatted log message.

Return type
str

8.4 BurinPercentLogRecord

This is the default log record factory and uses printf style formatting.

This should behave identically to `logging.LogRecord`.

class burin.**BurinPercentLogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*, ***kwargs*)

A log record that will be formatted like printf (% style).

This allows for deferred formatting using positional arguments that are passed in during log record creation.

This should behave identically to the Python builtin `logging.LogRecord` in normal use cases.

This is derived from *BurinLogRecord* (page 55).

factoryKey = '%'

This is the key used for the class as a log record factory. This is updated automatically when the class is set using *set_log_record_factory()* (page 20).

get_message()

This formats the log message.

All additional *args* that were part of the log record creation are used for the formatting of the log message.

Returns
The formatted log message.

Return type
str

CHAPTER

NINE

FILTERS AND FILTERERS

Filters can be used to determine if specific log records should be logged or not, and also provide the ability to modify records during processing.

Both *BurinLogger* (page 24) and *BurinHandler* (page 38) are subclasses of *BurinFilterer* (page 60). When processing a logging event all filter checks will be done on the record to determine if it should be logged.

Custom filters can be created by subclassing *BurinFilter* (page 59) and overriding the *BurinFilter.filter()* (page 59) method to perform custom checks or modify log records in place during processing.

9.1 BurinFilter

The default *BurinFilter* (page 59) is based on `logging.Filter` and should function identically to it; it is not a subclass of it though.

class burin.**BurinFilter**(*name=''*)

A filter can be used to apply filtering or modification of log records.

Note: This functions identically to the standard library `logging.Filter` class.

The base filter by default will allow all events which are lower in the logger hierarchy.

This creates the filter for the specified name.

The name of logger is used to allow only events from the specified logger and all loggers lower in the hierarchy. If this is an empty string the all events are allowed.

Parameters

name (*str*) – The name of the logger to allow events from along with all other loggers lower in the hierarchy. All events are allowed if this is an empty string. (Default = '')

filter(*record*)

Determines if the record should be logged.

Note: If you are subclassing *BurinFilter* (page 59) and intend to modify the log record then the modified record should also be returned. The *BurinFilterer* (page 60) will then use the modified record for all further processing and return it to the original caller.

Parameters

record (BurinLogRecord (page 55)) – The record to check.

Returns
Whether the record should be logged or not.

Return type
bool

9.2 BurinFilterer

This is a base class that is subclassed by both *BurinLogger* (page 24) and *BurinHandler* (page 38) so that filtering functionality can be re-used in both.

While this is based on the standard library `logging.Filterer` it is not a subclass of it.

Note: All methods of the *BurinFilterer* (page 60) with an *underscore_separated* name also have a *camelCase* alias name which matches the names used in the standard `logging` library.

---

class burin.**BurinFilterer**

A base class for loggers and handlers to allow common code for filtering.

Note: This works identically to the `logging.Filterer` in Python 3.12. The class is recreated in Burin to simplify allowing the *BurinFilterer.filter()* (page 60) method to return a log record. This was added to the standard library in 3.12 and is supported here for all versions of Python compatible with Burin (including versions below 3.12).

Initializes the filterer with an empty list of filters.

add_filter(*filter*)

Adds the specified filter to the the list of filters.

Parameters
filter (BurinFilter (page 59)) – The filter to add to this filterer instance.

filter(*record*)

Determine if a record is loggable according to all filters.

All filters are checked in the order that they were added using the *BurinFilterer.add_filter()* (page 60) method. If any filter returns **False** the record will not be logged.

If a filter returns a log record instance then that instance will be used for all further processing.

If none of the filters return **False** then a log record will be returned. If any filters returned an instance of a log record then the returned record will be the last instance that was returned by a filter.

However if any filter does return a **False** value then this method will also return a false value.

Note: In Python 3.12 the ability for a filterer to return a record was added to the standard library; it is supported here for all versions of Python compatible with Burin (including versions below 3.12).

Parameters
record (BurinLogRecord (page 55)) – The log record instance to check.

Returns
An instance of the record if it should be logged or **False** if it shouldn't. If any filters modified the record or returned an different instance of a record then that is what will be returned here. It should be used for all further processing and handling of the log record event.

Return type
BurinLogRecord (page 55) | bool

remove_filter (*filter*)

Removes the specified filter from the list of filters

Parameters
filter (BurinFilter (page 59)) – The filter to remove from this filterer instance.

CHAPTER

TEN

EXCEPTIONS

Burin uses a few custom exceptions. Where any of these may be raised is documented in class, method, or function descriptions.

exception burin.**ConfigError**

General exception for configuration errors.

exception burin.**FactoryError**

General exception for errors setting or using log record factories.

exception burin.**FormatError**

General exception for formatting errors.

CHAPTER

ELEVEN

PROJECT INFORMATION

11.1 Package Installation

Burin is available on PyPI and can be installed using any Python package manager such as `pip`.

Burin does not have any dependencies and is purely Python, so it should be usable in almost any CPython environment from version 3.7 - 3.12. It may also work with other Python implementations, but has not been tested.

11.2 Git Repository

Burin is open-source and the main repository is hosted on Github.

If you have questions or suggestions they can be discussed through the project's discussion board. Though if you encounter any issues please create an issue on the project's issue tracker.

Any pull requests should adhere to the same general style of the existing code base and pass all current linting rules and tests configured on the project.

11.3 Documentation

Burin's documentation is hosted on Read the Docs.

The documentation source is avaialble in the repository and can be built using Sphinx.

11.4 Build and Test

Burin uses Hatch to manage environments, task running, and building for development.

All tests use PyTest and can be run using the 'test' environment defined in the Hatch configuration within the pyproject.toml file.

Ruff is used for linting and also configured through the pyproject.toml file.

CHAPTER

TWELVE

RELEASE HISTORY

12.1 0.2.0 - February 10, 2024

12.1.1 Removals and Deprecations

- Python 3.6 support removed
- Python 3.7 support is deprecated and will be removed in a future release

12.1.2 Features and Additions

- Added support and feature compatiblity for Python 3.12
  - Added `burin.config.logAsyncioTasks` option and *taskName* property to `BurinLogRecord` (*taskName* is always **None** on Python 3.7 as task names weren't added to asyncio until 3.8)
  - Added `burin.get_handler_by_name` function
  - Added `burin.get_handler_names` functions
  - Added `BurinLogger.get_children` method
  - Added `BurinFilter` and `BurinFilterer` classes so that filterer checks can return a `BurinLogRecord` instance
  - Added checking of *flushOnClose* for handlers during shutdown
- Added support and feature compatiblity for Python 3.11
  - Added `burin.get_level_names_mapping` function
  - Added `BurinSyslogHandler.create_socket` method
  - Improved `BurinTimedRotatingFileHandler.should_rollover` so if target is not a normal file the check doesn't run repeatedly and will get rescheduled
  - Improved efficiency of finding first non-internal frame during logging event, especially if current frame is unavailable
  - Added access denied exception handling on initialization of `BurinNTEventLogHandler`
- Added optional *level* parameter to all burin handlers so a separate `BurinHandler.set_level` call isn't needed after creating a handler
- Enabled a single dictionary argument to be used with '{' and '$' style log records, just as they could be used for '%' style records
- Added *filedelay* parameter to `burin.basic_config`

- If running on Python 3.11 or greater then '$' style formatters will use `string.Template.is_valid()` for more efficient validation checking
- Added `BurinPercentLogRecord` to process records with '%' style formatting
- `BurinLogRecord` is now a base class that doesn't do any formatting itself

12.1.3 Fixes

- Fixed potential referencing issues by moving attributes `logMultiprocessing`, `logProcesses`, `logThreads`, and `raiseExceptions` to new `burin.config` object
- Fixed issue where '$' style formatters would return **None** after formatting
- Fixed extra arguments not getting passed through from `burin.exception` and `BurinLogger.exception`
- Fixed `NOTSET` log level missing from main burin module
- Fixed `burin.get_level_name` return value for unknown level names
- Fixed `BurinBufferingFormatter` not assigning default formatter properly
- Fixed issue where `BurinLogRecord.msecs` could round to 1000 (based on Python 3.11 fix)
- Fixed $ style formatter to use correct time search pattern (based on Python 3.11 fix)

12.1.4 Internal

- Created internal package _log_records
- Renamed internal package _logging to _loggers
- `BurinHandler` no longer inherits from `logging.Handler`, aside from Burin specific changes though functionality should remain identical
- In fallback `current_frame` function the exception object itself is used instead of going through `sys`
- More methods or other functions from the standard `logging` library have been re-created or overridden in Burin classes

12.1.5 Dependencies

- Replaced Flake8 with Ruff as dev dependency for linting
- Updated Sphinx doc dependency to 7.2.6
- Updated sphinx-rtd-theme doc dependency to 2.0.0
- Added Pytest and Coverage dependencies for testing
- Removed Flit dependency for building

12.1.6 Build and Environment

- Pipenv is no longer used in the project, so all related files (Pipfile, Pipfile.lock) have been removed
- Hatch is now used for both environment management, task running, and building

12.2 0.1.0 - June 2, 2022

- First formal release.

CHAPTER

THIRTEEN

LICENSE

Burin is licensed with the BSD 3-Clause license:

```
BSD 3-Clause License - SPDX:BSD-3-Clause

Copyright (c) 2022-2024 William Foster

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
   contributors may be used to endorse or promote products derived from
   this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

Some portions of this library are based on the Python 3.12.2 standard logging library to replicate functionality and improve compatibility. Files/modules with these portions are indicated with additional copyright notices in the module docstring at the top of the file. The Python standard library logging package is covered by the following licenses.

PSF LICENSE AGREEMENT FOR PYTHON 3.12.2:

```
1. This LICENSE AGREEMENT is between the Python Software Foundation
   ("PSF"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 3.12.2 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF
```

(continues on next page)

(continued from previous page)

```
   hereby grants Licensee a nonexclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare
   derivative works, distribute, and otherwise use Python 3.12.2 alone or
   in any derivative version, provided, however, that PSF's License
   Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2023
   Python Software Foundation; All Rights Reserved" are retained in Python
   3.12.2 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.12.2 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee
   hereby agrees to include in any such work a brief summary of the changes
   made to Python 3.12.2.

4. PSF is making Python 3.12.2 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY
   OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY
   REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY
   PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.12.2 WILL NOT INFRINGE
   ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.12.2
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A
   RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.12.2, OR
   ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material
   breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any
   relationship of agency, partnership, or joint venture between PSF and
   Licensee.  This License Agreement does not grant permission to use PSF
   trademarks or trade name in a trademark sense to endorse or promote
   products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 3.12.2, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

Vinay Sajip's license for logging package:

```
Copyright 2001-2022 by Vinay Sajip. All Rights Reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Vinay Sajip
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
VINAY SAJIP DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL
VINAY SAJIP BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR
ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

CHAPTER

FOURTEEN

INDEX

PYTHON MODULE INDEX

b

INDEX

G

H

I

L

M

N

P

R

S

W