`!logging.handlers` --- Logging handlers

- StreamHandler
- FileHandler
- NullHandler
- WatchedFileHandler
- BaseRotatingHandler
- RotatingFileHandler
- TimedRotatingFileHandler
- SocketHandler
- DatagramHandler
- SysLogHandler
- NTEventLogHandler
- SMTPHandler
- MemoryHandler
- HTTPHandler
- QueueHandler
- QueueListener

---

logging.handlers

Source code: `Lib/logging/handlers.py`

Important

This page contains only reference information. For tutorials, please see

- `Basic Tutorial <logging-basic-tutorial>`
- `Advanced Tutorial <logging-advanced-tutorial>`
- `Logging Cookbook <logging-cookbook>`

---

logging

The following useful handlers are provided in the package. Note that three of the handlers (`StreamHandler`, `FileHandler` and `NullHandler`) are actually defined in the `logging` module itself, but have been documented here along with the other handlers.

StreamHandler

The `StreamHandler` class, located in the core `logging` package, sends logging output to streams such as *sys.stdout*, *sys.stderr* or any file-like object (or, more precisely, any object which supports `write` and `flush` methods).

Returns a new instance of the `StreamHandler` class. If *stream* is specified, the instance will use it for logging output; otherwise, *sys.stderr* will be used.

emit(record)

If a formatter is specified, it is used to format the record. The record is then written to the stream followed by `terminator`. If exception information is present, it is formatted using `traceback.print_exception` and appended to the stream.

flush()

Flushes the stream by calling its `flush` method. Note that the `close` method is inherited from `~logging.Handler` and so does no output, so an explicit `flush` call may be needed at times.

setStream(stream)

Sets the instance's stream to the specified value, if it is different. The old stream is flushed before the new stream is set.

param stream
The stream that the handler should use.

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

3.7

terminator

String used as the terminator when writing a formatted record to a stream. Default value is `'\n'`.

If you don't want a newline termination, you can set the handler instance's `terminator` attribute to the empty string.

In earlier versions, the terminator was hardcoded as `'\n'`.

3.2

FileHandler

The `FileHandler` class, located in the core `logging` package, sends logging output to a disk file. It inherits the output functionality from `StreamHandler`.

Returns a new instance of the `FileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit`. By default, the file grows indefinitely. If *errors* is specified, it's used to determine how encoding errors are handled.

3.6 As well as string values, `~pathlib.Path` objects are also accepted for the *filename* argument.

3.9 The *errors* parameter was added.

close()

Closes the file.

emit(record)

Outputs the record to the file.

Note that if the file was closed due to logging shutdown at exit and the file mode is 'w', the record will not be emitted (see `42378`).

NullHandler

3.1

The `NullHandler` class, located in the core `logging` package, does not do any formatting or output. It is essentially a 'no-op' handler for use by library developers.

Returns a new instance of the `NullHandler` class.

emit(record)

This method does nothing.

handle(record)

This method does nothing.

createLock()

This method returns `None` for the lock, since there is no underlying I/O to which access needs to be serialized.

See `library-config` for more information on how to use `NullHandler`.

WatchedFileHandler

logging.handlers

The `WatchedFileHandler` class, located in the `!logging.handlers` module, is a `FileHandler` which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as *newsyslog* and *logrotate* which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore, *ST_INO* is not supported under Windows; `~os.stat` always returns zero for this value.

Returns a new instance of the `WatchedFileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit`. By default, the file grows indefinitely. If *errors* is provided, it determines how encoding errors are handled.

3.6 As well as string values, `~pathlib.Path` objects are also accepted for the *filename* argument.

3.9 The *errors* parameter was added.

reopenIfNeeded()

Checks to see if the file has changed. If it has, the existing stream is flushed and closed and the file opened again, typically as a precursor to outputting the record to the file.

3.6

emit(record)

Outputs the record to the file, but first calls `reopenIfNeeded` to reopen the file if it has changed.

BaseRotatingHandler

The `BaseRotatingHandler` class, located in the `!logging.handlers` module, is the base class for the rotating file handlers, `RotatingFileHandler` and `TimedRotatingFileHandler`. You should not need to instantiate this class, but it has attributes and methods you may need to override.

The parameters are as for `FileHandler`. The attributes are:

namer

If this attribute is set to a callable, the `rotation_filename` method delegates to this callable. The parameters passed to the callable are those passed to `rotation_filename`.

> [!NOTE] The namer function is called quite a few times during rollover, so it should be as simple and as fast as possible. It should also return the same output every time for a given input, otherwise the rollover behaviour may not work as expected.
>
> It's also worth noting that care should be taken when using a namer to preserve certain attributes in the filename which are used during rotation. For example, `RotatingFileHandler` expects to have a set of log files whose names contain successive integers, so that rotation works as expected, and `TimedRotatingFileHandler` deletes old log files (based on the `backupCount` parameter passed to the handler's initializer) by determining the oldest files to delete. For this to happen, the filenames should be sortable using the date/time portion of the filename, and a namer needs to respect this. (If a namer is wanted that doesn't respect this scheme, it will need to be used in a subclass of

> `TimedRotatingFileHandler` which overrides the `~TimedRotatingFileHandler.getFilesToDelete` method to fit in with the custom naming scheme.)

3.3

BaseRotatingHandler.rotator

If this attribute is set to a callable, the `rotate` method delegates to this callable. The parameters passed to the callable are those passed to `rotate`.

3.3

BaseRotatingHandler.rotation_filename(default_name)

Modify the filename of a log file when rotating.

This is provided so that a custom filename can be provided.

The default implementation calls the 'namer' attribute of the handler, if it's callable, passing the default name to it. If the attribute isn't callable (the default is `None`), the name is returned unchanged.

param default_name
The default name for the log file.

3.3

BaseRotatingHandler.rotate(source, dest)

When rotating, rotate the current log.

The default implementation calls the 'rotator' attribute of the handler, if it's callable, passing the source and dest arguments to it. If the attribute isn't callable (the default is `None`), the source is simply renamed to the destination.

param source
The source filename. This is normally the base filename, e.g. 'test.log'.

param dest
The destination filename. This is normally what the source is rotated to, e.g. 'test.log.1'.

3.3

The reason the attributes exist is to save you having to subclass - you can use the same callables for instances of `RotatingFileHandler` and `TimedRotatingFileHandler`. If either the namer or rotator callable raises an exception, this will be handled in the same way as any other exception during an `emit` call, i.e. via the `handleError` method of the handler.

If you need to make more significant changes to rotation processing, you can override the methods.

For an example, see `cookbook-rotator-namer`.

RotatingFileHandler

The `RotatingFileHandler` class, located in the `!logging.handlers` module, supports rotation of disk log files.

Returns a new instance of the `RotatingFileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit`. By default, the file grows indefinitely. If *errors* is provided, it determines how encoding errors are handled.

You can use the *maxBytes* and *backupCount* values to allow the file to `rollover` at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly *maxBytes* in length; but if either of *maxBytes* or *backupCount* is zero, rollover never occurs, so you generally want to set *backupCount* to at least 1, and have a non-zero *maxBytes*. When *backupCount* is non-zero, the system will save old log files by appending the extensions '.1', '.2' etc., to the filename. For example, with a *backupCount* of 5 and a base file name of `app.log`, you would get `app.log`, `app.log.1`, `app.log.2`, up to `app.log.5`. The file being written to is always `app.log`. When this file is filled, it is closed and renamed to `app.log.1`, and if files `app.log.1`, `app.log.2`, etc. exist, then they are renamed to `app.log.2`, `app.log.3` etc. respectively.

3.6 As well as string values, `~pathlib.Path` objects are also accepted for the *filename* argument.

3.9 The *errors* parameter was added.

doRollover()

Does a rollover, as described above.

emit(record)

Outputs the record to the file, catering for rollover as described previously.

shouldRollover(record)

See if the supplied record would cause the file to exceed the configured size limit.

TimedRotatingFileHandler

The `TimedRotatingFileHandler` class, located in the `!logging.handlers` module, supports rotation of disk log files at certain timed intervals.

Returns a new instance of the `TimedRotatingFileHandler` class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of *when* and *interval*.

You can use the *when* to specify the type of *interval*. The list of possible values is below. Note that they are not case sensitive.

Value	Type of interval	If/how *atTime* is used
`'S'`	Seconds	Ignored
`'M'`	Minutes	Ignored
`'H'`	Hours	Ignored
`'D'`	Days	Ignored
`'W0'-'W6'`	Weekday (0=Monday)	Used to compute initial rollover time
`'midnight'`	Roll over at midnight, if *atTime* not specified, else at time *atTime*	Used to compute initial rollover time

When using weekday-based rotation, specify 'W0' for Monday, 'W1' for Tuesday, and so on up to 'W6' for Sunday. In this case, the value passed for *interval* isn't used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format `%Y-%m-%d_%H-%M-%S` or a leading portion thereof, depending on the rollover interval.

When computing the next rollover time for the first time (when the handler is created), the creation time (if supported by the OS and file system) or the last modification of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the *utc* argument is true, times in UTC will be used; otherwise local time is used.

If *backupCount* is nonzero, at most *backupCount* files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If *delay* is true, then file opening is deferred until the first call to `emit`.

If *atTime* is not `None`, it must be a `datetime.time` instance which specifies the time of day when rollover occurs, for the cases where rollover is set to happen "at midnight" or "on a particular weekday". Note that in these cases, the *atTime* value is effectively used to compute the *initial* rollover, and subsequent rollovers would be calculated via the normal interval calculation.

If *errors* is specified, it's used to determine how encoding errors are handled.

> [!NOTE] Calculation of the initial rollover time is done when the handler is initialised. Calculation of subsequent rollover times is done only when rollover occurs, and rollover occurs only when emitting output. If this is not kept in mind, it might lead to some confusion. For example, if an interval of "every minute" is set, that does not mean you will always see log files with times (in the filename) separated by a minute; if, during application execution, logging output is generated more frequently than once a minute, *then* you can expect to see log files with times separated by a minute. If, on the other hand, logging messages are only output once every five minutes (say), then there will be gaps in the file times corresponding to the minutes where no output (and hence no rollover) occurred.

3.4 *atTime* parameter was added.

3.6 As well as string values, `~pathlib.Path` objects are also accepted for the *filename* argument.

3.9 The *errors* parameter was added.

3.16 Use the creation time instead of the last modification time, if supported by the OS and file system.

doRollover()

Does a rollover, as described above.

emit(record)

Outputs the record to the file, catering for rollover as described above.

getFilesToDelete()

Returns a list of filenames which should be deleted as part of rollover. These are the absolute paths of the oldest backup log files written by the handler.

shouldRollover(record)

See if enough time has passed for a rollover to occur and if it has, compute the next rollover time.

SocketHandler

The `SocketHandler` class, located in the `!logging.handlers` module, sends logging output to a network socket. The base class uses a TCP socket.

Returns a new instance of the `SocketHandler` class intended to communicate with a remote machine whose address is given by *host* and *port*.

3.4 If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a TCP socket is created.

close()

Closes the socket.

emit()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a `~logging.LogRecord`, use the `~logging.makeLogRecord` function.

handleError()

Handles an error which has occurred during `emit`. The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()

This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (`socket.SOCK_STREAM`).

makePickle(record)

Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket. The details of this operation are equivalent to:

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

Note that pickles aren't completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

send(packet)

Send a pickled byte-string *packet* to the socket. The format of the sent byte-string is as described in the documentation for `~SocketHandler.makePickle`.

This function allows for partial sends, which can happen when the network is busy.

createSocket()

Tries to create a socket; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can't be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes:

- `retryStart` (initial delay, defaulting to 1.0 seconds).
- `retryFactor` (multiplier, defaulting to 2.0).
- `retryMax` (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up *after* the handler has been used, you could lose messages (since the handler won't even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).

DatagramHandler

The `DatagramHandler` class, located in the `!logging.handlers` module, inherits from `SocketHandler` to support sending logging messages over UDP sockets.

Returns a new instance of the `DatagramHandler` class intended to communicate with a remote machine whose address is given by *host* and *port*.

> [!NOTE] As UDP is not a streaming protocol, there is no persistent connection between an instance of this handler and *host*. For this reason, when using a network socket, a DNS lookup might have to be made each time an event is logged, which can introduce some latency into the system. If this affects you, you can do a lookup yourself and initialize this handler using the looked-up IP address rather than the hostname.

3.4 If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a UDP socket is created.

emit()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a `~logging.LogRecord`, use the `~logging.makeLogRecord` function.

makeSocket()

The factory method of `SocketHandler` is here overridden to create a UDP socket (`socket.SOCK_DGRAM`).

send(s)

Send a pickled byte-string to a socket. The format of the sent byte-string is as described in the documentation for `SocketHandler.makePickle`.

SysLogHandler

The `SysLogHandler` class, located in the `!logging.handlers` module, supports sending logging messages to a remote or local Unix syslog.

Returns a new instance of the `SysLogHandler` class intended to communicate with a remote Unix machine whose address is given by *address* in the form of a `(host, port)` tuple. If *address* is not specified, `('localhost', 514)` is used. The address is used to open a socket. An alternative to providing a `(host, port)` tuple is providing an address as a string or a `bytes` object, for example '/dev/log'. In this case, a Unix domain socket is used to send the message to the syslog. If *facility* is

not specified, `LOG_USER` is used. The type of socket opened depends on the *socktype* argument, which defaults to `socket.SOCK_DGRAM` and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of `socket.SOCK_STREAM`. If *timeout* is specified, it sets a timeout (in seconds) for the socket operations. This can help prevent the program from hanging indefinitely if the syslog server is unreachable. By default, *timeout* is `None`, meaning no timeout is applied.

Note that if your server is not listening on UDP port 514, `SysLogHandler` may appear not to work. In that case, check what address you should be using for a domain socket - it's system dependent. For example, on Linux it's usually '/dev/log' but on OS/X it's '/var/run/syslog'. You'll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.

> [!NOTE] On macOS 12.x (Monterey), Apple has changed the behaviour of their syslog daemon - it no longer listens on a domain socket. Therefore, you cannot expect `SysLogHandler` to work on this system.
>
> See `91070` for more information.

3.2 *socktype* was added.

3.14 *timeout* was added.

3.16 *address* can now be a `bytes` object.

close()

Closes the socket to the remote host.

createSocket()

Tries to create a socket and, if it's not a datagram socket, connect it to the other end. This method is called during handler initialization, but it's not regarded as an error if the other end isn't listening at this point - the method will be called again when emitting an event, if there is no socket at that point.

3.11

emit(record)

The record is formatted, and then sent to the syslog server. If exception information is present, it is *not* sent to the server.

3.2.1 (See: `12168`.) In earlier versions, the message sent to the syslog daemons was always terminated with a NUL byte, because early versions of these daemons expected a NUL terminated message - even though it's not in the relevant specification (`5424`). More recent versions of these daemons don't expect the NUL byte but strip it off if it's there, and even more recent daemons (which adhere more closely to RFC 5424) pass the NUL byte on as part of the message.

To enable easier handling of syslog messages in the face of all these differing daemon behaviours, the appending of the NUL byte has been made configurable, through the use of a class-level attribute, `append_nul`. This defaults to `True` (preserving the existing behaviour) but can be set to `False` on a `SysLogHandler` instance in order for that instance to *not* append the NUL terminator.

3.3 (See: `12419`.) In earlier versions, there was no facility for an "ident" or "tag" prefix to identify the source of the message. This can now be specified using a class-level attribute, defaulting to `""` to preserve existing behaviour, but which can be overridden on a `SysLogHandler` instance in order for that instance to prepend the ident to every message handled. Note that the provided ident must be text, not bytes, and is prepended to the message exactly as is.

encodePriority(facility, priority)

Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic `LOG_` values are defined in `SysLogHandler` and mirror the values defined in the `sys/syslog.h` header file.

Priorities

Name (string)	Symbolic value
`alert`	LOG_ALERT
`crit` or `critical`	LOG_CRIT
`debug`	LOG_DEBUG
`emerg` or `panic`	LOG_EMERG
`err` or `error`	LOG_ERR
`info`	LOG_INFO
`notice`	LOG_NOTICE
`warn` or `warning`	LOG_WARNING

Facilities

Name (string)	Symbolic value
`auth`	LOG_AUTH
`authpriv`	LOG_AUTHPRIV
`cron`	LOG_CRON
`daemon`	LOG_DAEMON
`ftp`	LOG_FTP
`kern`	LOG_KERN
`lpr`	LOG_LPR
`mail`	LOG_MAIL
`news`	LOG_NEWS
`syslog`	LOG_SYSLOG
`user`	LOG_USER
`uucp`	LOG_UUCP
`local0`	LOG_LOCAL0
`local1`	LOG_LOCAL1
`local2`	LOG_LOCAL2
`local3`	LOG_LOCAL3
`local4`	LOG_LOCAL4
`local5`	LOG_LOCAL5
`local6`	LOG_LOCAL6
`local7`	LOG_LOCAL7

mapPriority(levelname)

Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL` to the equivalent syslog names, and all other level names to 'warning'.

NTEventLogHandler

The `NTEventLogHandler` class, located in the `!logging.handlers` module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond's Win32 extensions for Python installed.

Returns a new instance of the `NTEventLogHandler` class. The *appname* is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The *dllname* should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, `'win32service.pyd'` is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The *logtype* is one of `'Application'`, `'System'` or `'Security'`, and defaults to `'Application'`.

close()

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

emit(record)

Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory(record)

Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

getEventType(record)

Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler's typemap attribute, which is set up in `__init__` to a dictionary which contains mappings for `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL`. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler's *typemap* attribute.

getMessageID(record)

Returns the message ID for the record. If you are using your own messages, you could do this by having the *msg* passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base

message ID in `win32service.pyd`.

SMTPHandler

The `SMTPHandler` class, located in the `!logging.handlers` module, supports sending logging messages to an email address via SMTP.

Returns a new instance of the `SMTPHandler` class. The instance is initialized with the from and to addresses and subject line of the email. The *toaddrs* should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the *mailhost* argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the *credentials* argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the *secure* argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the `smtplib.SMTP.starttls` method.)

A timeout can be specified for communication with the SMTP server using the *timeout* argument.

3.3 Added the *timeout* parameter.

emit(record)

Formats the record and sends it to the specified addressees.

getSubject(record)

If you want to specify a subject line which is record-dependent, override this method.

MemoryHandler

The `MemoryHandler` class, located in the `!logging.handlers` module, supports buffering of logging records in memory, periodically flushing them to a `target` handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

`MemoryHandler` is a subclass of the more general `BufferingHandler`, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling `shouldFlush` to see if the buffer should be flushed. If it should, then `flush` is expected to do the flushing.

Initializes the handler with a buffer of the specified capacity. Here, *capacity* means the number of logging records buffered.

emit(record)

Append the record to the buffer. If `shouldFlush` returns true, call `flush` to process the buffer.

flush()

For a `BufferingHandler` instance, flushing means that it sets the buffer to an empty list. This method can be overwritten to implement more useful flushing behavior.

shouldFlush(record)

Return `True` if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

Returns a new instance of the `MemoryHandler` class. The instance is initialized with a buffer size of *capacity* (number of records buffered). If *flushLevel* is not specified, `ERROR` is used. If no *target* is specified, the target will need to be set using `setTarget` before this handler does anything useful. If *flushOnClose* is specified as `False`, then the buffer is *not* flushed when the handler is closed. If not specified or specified as `True`, the previous behaviour of flushing the buffer will occur when the handler is closed.

3.6 The *flushOnClose* parameter was added.

close()

Calls `flush`, sets the target to `None` and clears the buffer.

flush()

For a `MemoryHandler` instance, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when buffered records are sent to the target. Override if you want different behavior.

setTarget(target)

Sets the target handler for this handler.

shouldFlush(record)

Checks for buffer full or a record at the *flushLevel* or higher.

HTTPHandler

The `HTTPHandler` class, located in the `!logging.handlers` module, supports sending logging messages to a web server, using either `GET` or `POST` semantics.

Returns a new instance of the `HTTPHandler` class. The *host* can be of the form `host:port`, should you need to use a specific port number. If no *method* is specified, `GET` is used. If *secure* is true, a HTTPS connection will be used. The *context* parameter may be set to a `ssl.SSLContext` instance to configure the SSL settings used for the HTTPS connection. If *credentials* is specified, it should be a 2-tuple consisting of userid and password, which will be placed in a HTTP 'Authorization' header using Basic authentication. If you specify credentials, you should also specify secure=True so that your userid and password are not passed in cleartext across the wire.

3.5 The *context* parameter was added.

mapLogRecord(record)

Provides a dictionary, based on `record`, which is to be URL-encoded and sent to the web server. The default implementation just returns `record.__dict__`. This method can be overridden if e.g. only a subset of `~logging.LogRecord` is to be sent to the web server, or if more specific customization of what's sent to the server is required.

emit(record)

Sends the record to the web server as a URL-encoded dictionary. The `mapLogRecord` method is used to convert the record to the dictionary to be sent.

> [!NOTE] Since preparing a record for sending it to a web server is not the same as a generic formatting operation, using `~logging.Handler.setFormatter` to specify a `~logging.Formatter` for a `HTTPHandler` has no effect. Instead of calling `~logging.Handler.format`, this handler calls `mapLogRecord` and then `urllib.parse.urlencode` to encode the dictionary in a form suitable for sending to a web server.

QueueHandler

3.2

The `QueueHandler` class, located in the `!logging.handlers` module, supports sending logging messages to a queue, such as those implemented in the `queue` or `multiprocessing` modules.

Along with the `QueueListener` class, `QueueHandler` can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via `SMTPHandler`) are done on a separate thread.

Returns a new instance of the `QueueHandler` class. The instance is initialized with the queue to send messages to. The *queue* can be any queue-like object; it's used as-is by the `enqueue` method, which needs to know how to send messages to it. The queue is not *required* to have the task tracking API, which means that you can use `~queue.SimpleQueue` instances for *queue*.

> [!NOTE] If you are using `multiprocessing`, you should avoid using `~queue.SimpleQueue` and instead use `multiprocessing.Queue`.

> [!WARNING] The `multiprocessing` module uses an internal logger created and accessed via `~multiprocessing.get_logger`. `multiprocessing.Queue` will log `DEBUG` level messages upon items being queued. If those log messages are processed by a `QueueHandler` using the same `multiprocessing.Queue` instance, it will cause a deadlock or infinite recursion.

emit(record)

Enqueues the result of preparing the LogRecord. Should an exception occur (e.g. because a bounded queue has filled up), the `~logging.Handler.handleError` method is called to handle the error. This can result in the record silently being dropped (if `logging.raiseExceptions` is `False`) or a message printed to `sys.stderr` (if `logging.raiseExceptions` is `True`).

prepare(record)

Prepares a record for queuing. The object returned by this method is enqueued.

The base implementation formats the record to merge the message, arguments, exception and stack information, if present. It also removes unpickleable items from the record in-place. Specifically, it overwrites the record's `msg` and `message` attributes with the merged message (obtained by calling the handler's `format` method), and sets the `args`, `exc_info` and `exc_text` attributes to `None`.

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

> [!NOTE] The base implementation formats the message with arguments, sets the `message` and `msg` attributes to the formatted message and sets the `args` and `exc_text` attributes to `None` to allow pickling and to prevent further attempts at formatting. This means that a handler on the `QueueListener` side won't have the information to do custom formatting, e.g. of exceptions. You may wish to subclass `QueueHandler` and override this method to e.g. avoid setting `exc_text` to `None`. Note that the `message` / `msg` / `args` changes are related to ensuring the record is pickleable, and you might or might not be able to avoid doing that depending on whether your `args` are pickleable. (Note that you may have to consider not only your own code but also code in any libraries that you use.)

enqueue(record)

Enqueues the record on the queue using `put_nowait()`; you may want to override this if you want to use blocking behaviour, or a timeout, or a customized queue implementation.

listener

When created via configuration using `~logging.config.dictConfig`, this attribute will contain a `QueueListener` instance for use with this handler. Otherwise, it will be `None`.

3.12

QueueListener

3.2

The `QueueListener` class, located in the `!logging.handlers` module, supports receiving logging messages from a queue, such as those implemented in the `queue` or `multiprocessing` modules. The messages are received from a queue in an internal thread and passed, on the same thread, to one or more handlers for processing. While `QueueListener` is not itself a handler, it is documented here because it works hand-in-hand with `QueueHandler`.

Along with the `QueueHandler` class, `QueueListener` can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via `SMTPHandler`) are done on a separate thread.

Returns a new instance of the `QueueListener` class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it's passed as-is to the `dequeue` method, which needs to know how to get messages from it. The queue is not *required* to have the task tracking API (though it's used if available), which means that you can use `~queue.SimpleQueue` instances for *queue*.

> [!NOTE] If you are using `multiprocessing`, you should avoid using `~queue.SimpleQueue` and instead use `multiprocessing.Queue`.

If `respect_handler_level` is `True`, a handler's level is respected (compared with the level for the message) when deciding whether to pass messages to that handler; otherwise, the behaviour is as in previous Python versions - to always pass each message to each handler.

3.5 The `respect_handler_level` argument was added.

3.14 `QueueListener` can now be used as a context manager via `with`. When entering the context, the listener is started. When exiting the context, the listener is stopped. `~contextmanager.__enter__` returns the `QueueListener` object.

dequeue(block)

Dequeues a record and return it, optionally blocking.

The base implementation uses `get()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

prepare(record)

Prepare a record for handling.

This implementation just returns the passed-in record. You may want to override this method if you need to do any custom marshalling or manipulation of the record before passing it to the handlers.

handle(record)

Handle a record.

This just loops through the handlers offering them the record to handle. The actual object passed to the handlers is that which is returned from `prepare`.

start()

Starts the listener.

This starts up a background thread to monitor the queue for LogRecords to process.

3.14 Raises `RuntimeError` if called and the listener is already running.

stop()

Stops the listener.

This asks the thread to terminate, and then waits for it to do so. Note that if you don't call this before your application exits, there may be some records still left on the queue, which won't be processed.

enqueue_sentinel()

Writes a sentinel to the queue to tell the listener to quit. This implementation uses `put_nowait()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

3.3

Module `logging`
API reference for the logging module.

Module `logging.config`
Configuration API for the logging module.

□□□ #2
□ □□□ □□□ 2026-10-31 17:14:08 UTC
□ □□□ □□□ 2026-10-31 17:14:12 UTC