Chapter 28
Advanced Logging

28.1 Introduction

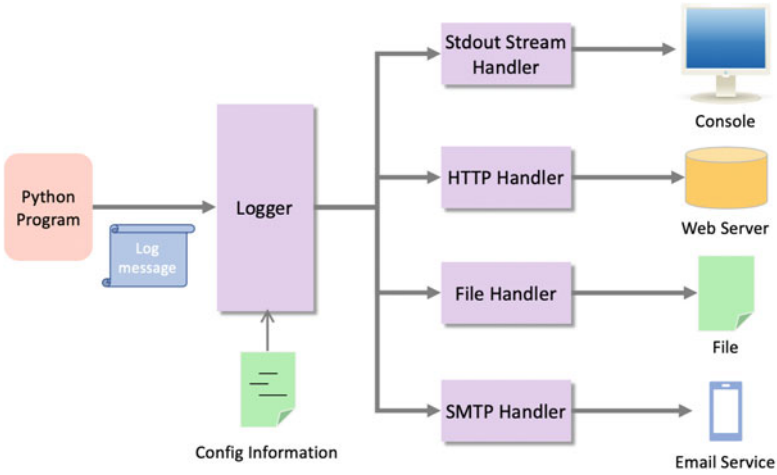
In this chapter we go further into the configuration and modification of the Python logging module. In particular we will look at Handlers (used to determine the destination fo log messages), Filters which can be used by Handlers to provide finer grained control of log output and logger configuration files. We conclude the chapter by considering performance issues associated with logging.

28.2 Handlers

Within the logging pipeline, it ia handlers that send the log message to their final destination.

By default the handler is set up to direct output to the console/terminal associated with the running program. However, this can be changed to send the log messages to a file, to an email service, to a web server etc. Or indeed to any combination of these as there can be multiple handlers configured for a logger. This is shown in the diagram below:

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_28



In the above diagram the logger has been configured to send all log messages to four different handlers which allow a log message to be written to the console, to a web server to a file and to an email service. Such a behaviour may be required because:

- The web server will allow developers access to a web interface that allows them to see the log files even if they do not have permission to access a production server.
- The log file ensures that all the log data is permanently stored in a file within the file store.
- An email message may be sent to a notification system so that someone will be notified that there is an issue to be investigated.
- The console may still be available to the system administrators who may wish to look at the log messages generated.

The Python logging framework comes with several different handlers as suggested above and listed below:

- `logging.StreamHandler` sends messages to outputs such as `stdout`, `stderr` etc.
- `logging.FileHandler` sends log messages to files. There are several varieties of File Handler in addition to the basic `FileHandler`, these include the `logging.handlers.RotatingFileHandler` (which will rotate log files based on a maximum file size) and `logging.handlers.TimeRotatingFileHandler` (which rotates the log file at specified time intervals e.g. daily).
- `logging.handlers.SocketHandler` which sends messages to a TCP/IP socket where it can be received by a TCP Server.
- `logging.handlers.SMTPHandler` that sends messages by the SMTP (Simple Mail Transfer Protocol) to a email server.
- `logging.handlers.SysLogHandler` that sends log messages to a Unix syslog program.

- `logging.handlers.NTEventLogHandler` that sends message to a Windows event log.
- `logging.handlers.HTTPHandler` which sends messages to a HTTP server.
- `logging.NullHandler` that does nothing with error messages. This is often used by library developers who want to include logging in their applications but expect developers to set up an appropriate handler when they use the library.

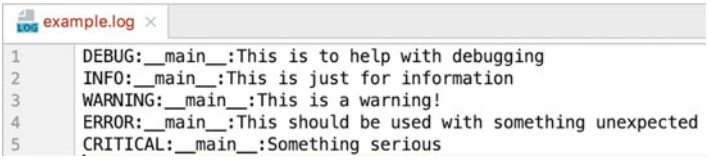
All of these handlers can be configured programmatically or via a configuration file.

28.2.1 Setting the Root Output Handler

The following example, uses the `logging.basicConfig()` function to set up the root logger to use a `FileHandler` that will write the log messages to a file called 'example.log':

```
import logging

# Sets a file handler on the root logger to
# save log messages to the example.log file
logging.basicConfig(filename='example.log',level=logging.DEBUG)

# If no handler is explicitly set on the name logger
# it will delegate the messages to the parent logger to handle
logger = logging.getLogger(__name__)

logger.debug('This is to help with debugging' )
logger.info('This is just for information' )
logger.warning('This is a warning!' )
logger.error('This should be used with something unexpected'  )
logger.critical('Something serious' )
```

Note that if no handler is specified for a named logger then it delegates output to the parent (in this case the root) logger. The file generated for the above program is shown below:



As can be seen from this the default formatter is now configured for a `FileHandler`. This FileHandler adds the log message level before the log message itself.

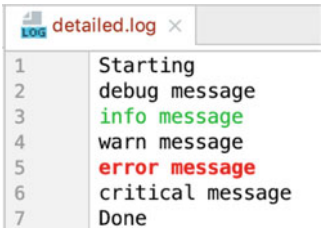
28.2.2 Programmatically Setting the Handler

It is also possible to programmatically create a handler and set it for the logger. This is done by instantiating one of the existing handler classes (or by subclassing an existing handler such as the root `Handler` class or the `FileHander` etc.). The instantiated handler can then be added as a handler to the logger (remember the logger can have multiple handlers this is why the method is called `addHandler()` rather than something such as setHandler).

An example of explicitly setting the FileHandler for a logger is given below:

```
import logging

# Empty basic config turns off default console handler
logging.basicConfig()

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# create file handler which logs to the specified file
file_handler = logging.FileHandler('detailed.log')

# Add the handler to the Logger
logger.addHandler(file_handler)

# 'application' code
def do_something():
    logger.debug('debug message')
    logger.info('info message')
    logger.warning('warn message')
    logger.error('error message')
    logger.critical('critical message')

logger.info('Starting')
do_something()
logger.info('Done')
```

The result of running this code is that a log file is created with the logged messages:



Given that this is a lot more code than using the `basicConfig()` function; the question here might be 'Why bother?'. The answer is two fold:

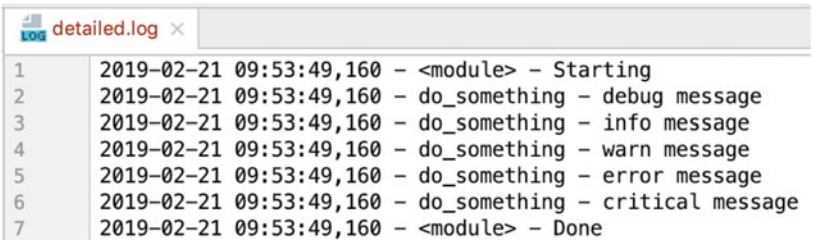
- You can have different handlers for different loggers rather than setting the handler to be used centrally.
- Each handler can have its own format set so that logging to a file has a different format to logging to the console.

We can set the format for the handler by instantiating the `logging.Formatter` class with an appropriate format string. The formatter object can then be applied to a handler using the `setFormatter()` method on the handler object.

For example, we can modify the above code to include a formatter that is then set on the file handler as shown below.

```
# create file handler which logs to the specified file
file_handler = logging.FileHandler('detailed.log' )

# Create formatter for the file_handler
formatter = logging.Formatter('%(asctime)s - %(funcName)s -
%(message)s' )
file_handler.setFormatter(formatter)

logger.addHandler(file_handler)
```

The log file now generated is modified such that each message includes a time stamp, the function name (or module if at the module level) as well as the log message itself.



28.2.3 Multiple Handlers

As suggested in the previous section we can create multiple handlers to send log messages to different locations; for example from the console, to files and even email servers. The following program illustrates setting up both a *file handler* and a *console handler* for a module level logger.

To do this we create two handlers the `file_handler` and the `console_handler`. As a side effect we can also give them different log levels and different formatters. In this case the `file_handler` inherits the log level of the logger itself (which is `DEBUG`) while the `console_handler` has its log level set explicitly at `WARNING`. This means different amounts of information will be logged to the log file than the console output.

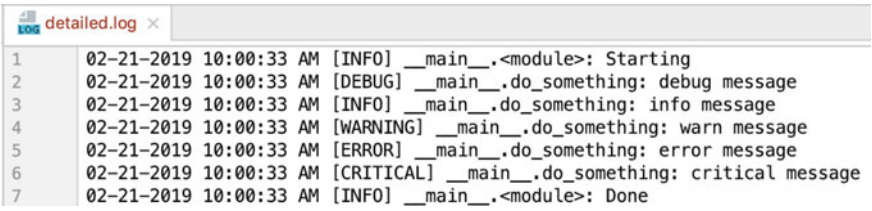
We have also set different formatters on each handler; in this case the log file handler's formatter provides more information than the console handlers formatter.

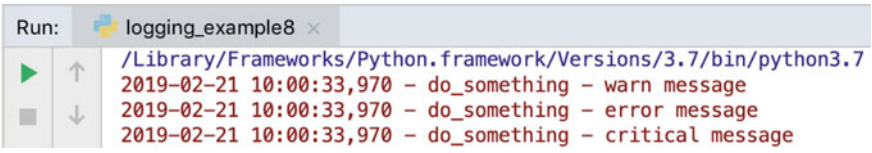
Both handlers are then added to the logger before it is used.

```
# Multiple Handlers and formatters
import logging

# Set up the default root logger to do nothing
logging.basicConfig(handlers=[logging.NullHandler()])

# Obtain the module level logger and set level to DEBUG
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Create file handler
file_handler = logging.FileHandler('detailed.log')

# Create console handler with a higher log level
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.WARNING)

# Create formatter for the file handler
fh_formatter = logging.Formatter('%(asctime)s [%(levelname)s]
%(name)s.%(funcName)s: %(message)s',
                                 datefmt='%m-%d-%Y %I:%M:%S
%p')
file_handler.setFormatter(fh_formatter)
```

```
# Create formatter for the console handler
console_formatter = logging.Formatter('%(asctime)s -
%(funcName)s - %(message)s')
console_handler.setFormatter(console_formatter)

# Add the handlers to logger
logger.addHandler(console_handler)
logger.addHandler(file_handler)

# 'application' code
def do_something():
    logger.debug('debug message')
    logger.info('info message')
    logger.warning('warn message')
    logger.error('error message')
    logger.critical('critical message')

logger.info('Starting')
do_something()
logger.info('Done')
```

The output from this program is now split between the log file and the console out, as shown below:





28.3 Filters

Filters can be used by Handlers to provide finer grained control of the log output. A filter can be added to a logger using the `logger.addFilter()` method. A Filter can be created by extending the `logging.Filter` class and

implementing the `filter()` method. This method takes a log record. This log record can be validated to determine if the record should be output or not. If it should be output then `True` is returned, if the record should be ignored `False` should be returned.

In the following example, a filter called `MyFilter` is defined that will filter out all log messages containing the string 'John'. It is added as a filter to the logger and then two log messages are generated.

```
import logging

class MyFilter(logging.Filter):

    def filter(self, record):
        if 'John' in record.msg:
            return False
        else:
            return True

logging.basicConfig(format='%(asctime)s %(message)s',
level=logging.DEBUG)

logger = logging.getLogger()
logger.addFilter(MyFilter())

logger.debug('This is to help with debugging')
logger.info('This is information on John')
```

The output shows that only the log message that does not contain the string 'John' is output:

```
2019-02-20 17:23:22,650 This is to help with debugging
```

28.4 Logger Configuration

All the examples so far in this chapter have used programmatic configuration of the logging framework. This is certainly feasible as the examples show, but it does require a code change if you wish to alter the logging level for any particular logger, or to change where a particular handler is routing the log messages.

For most production systems a better solution is to use an external configuration file which is loaded when the application is run and is used to dynamically configure the logging framework. This allows system administrators and others to change the log level, the log destination, the log format etc. without needing to change the code.

The logging configuration file can be written using several standard formats from JSON (the Java Script Object Notation), to YAML (Yet Another Markup Language) format, or as a set of key-value pairs in a `.conf` file. For further information on the different options available see the Python `logging` module documentation.

In this book we will briefly explore the YAML file format used to configure loggers.

```
version: 1
formatters:
  myformatter:
    format: '%(asctime)s [%(levelname)s] %(name)s.%(funcName)s:
%(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: myformatter
    stream: ext://sys.stdout
loggers:
  myLogger:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
  level: ERROR
  handlers: [console]
```

The above YAML code is stored in a file called `logging.conf.yaml`; however you can call this file anything that is meaningful.

The YAML file always starts with a version number. This is an integer value representing the YAML schema version (currently this can only be the value 1). All other keys in the file are optional, they include:

- *formatters*—this lists one or more formatters; each formatter has a name which acts as a key and then a format value which is a string defining the format of a log message.
- *filters*—this is a lit of filter names and a set of filter definitions.
- *handlers*—this is a list of named handlers. Each handler definition is made up of a set of key value pairs where the keys define the class used for the filter (mandatory), the log level of the filter (optional), the formatter to use with the handler (optional) and a list of filters to apply (optional).
- *loggers*—provides one or more named loggers. Each logger can indicate the log level (optional) and a list of handlers (optional). The `propagate` option can be used to stop messages propagating to a parent logger (by setting it to `False`).
- *root*—this is the configuration for the root logger.

This file can be loaded into a Python application using the `PyYAML` module. This provides a YAML parser that can load a YAML file as a dictionary structure that can be passed to the `logging.config.dictConfig()` function. As this is a file it must be opened and closed to ensure that the resource is handled appropriately; it is therefore best managed using the `with-as` statement as shown below:

```
with open('logging.config.yaml' , 'r') as f:
    config = yaml.safe_load(f.read())
    logging.config.dictConfig(config)
```

This will open the YAML file in read-only mode and close it when the two statements have been executed. This snippet is used in the following application that loads the logger configuration from the YAML file:

```
import logging
import logging.config
import yaml

with open('logging.config.yaml', 'r') as f:
    config = yaml.safe_load(f.read())
    logging.config.dictConfig(config)

logger = logging.getLogger('myLogger')

# 'application' code
def do_something():
    logger.debug('debug message')
    logger.info('info message')
    logger.warning('warn message')
    logger.error('error message')
    logger.critical('critical message')

logger.info('Starting')
do_something()
logger.info('Done')
```

The output from this using the earlier YAML file is:

```
2019-02-21 16:20:46,466 [INFO] myLogger.<module>: Starting
2019-02-21 16:20:46,466 [DEBUG] myLogger.do_something: debug
message
2019-02-21 16:20:46,466 [INFO] myLogger.do_something: info
message
2019-02-21 16:20:46,466 [WARNING] myLogger.do_something: warn
message
2019-02-21 16:20:46,466 [ERROR] myLogger.do_something: error
message
2019-02-21 16:20:46,466 [CRITICAL] myLogger.do_something:
critical message
2019-02-21 16:20:46,466 [INFO] myLogger.<module>: Done
```

28.5 Performance Considerations

Performance when logging should always be a consideration. In general you should aim to avoid performing any unnecessary work when logging is disabled (or disabled for the level being used). This may seem obvious but it can occur in several unexpected ways.

One example is string concatenation. If a message to be logged involves string concatenation; then that string concatenation will always be performed when a log method is being invoked. For example:

```
logger.debug('Count: ' + count + ', total: ' + total)
```

This will always result in the string being generated for `count` and `total` before the call is made to the debug function; even if the debug level is not turned on. However using a format string will avoid this. The formatting involved will only be performed if the string is to be used in a log message. You should therefore always use string formatting to populate log messages. For erxmaple:

```
logger.debug('Count: %d, total: %d ', count, 42)
```

Another potential optimisation is to use the `logger.isEnabledFor(level)` method as a guard against running the log statement. This can be useful in situations where an associated operation must be performed to support the logging operation and this operation is expensive. For example:

```
if logger.isEnabledFor(logging.DEBUG):
    logger.debug('Message with %s, %s', expensive_func1(),
                                        expensive_func2())
```

Now the two expensive functions will only be executed if the DEBUG log level is set.

28.6 Exercises

Using the logging you dded to the Account class int he last chapter, you should load the log configuration information from a YAML file similar to that used in this chapter.

This should be loaded into the application program used to drive the account classes.