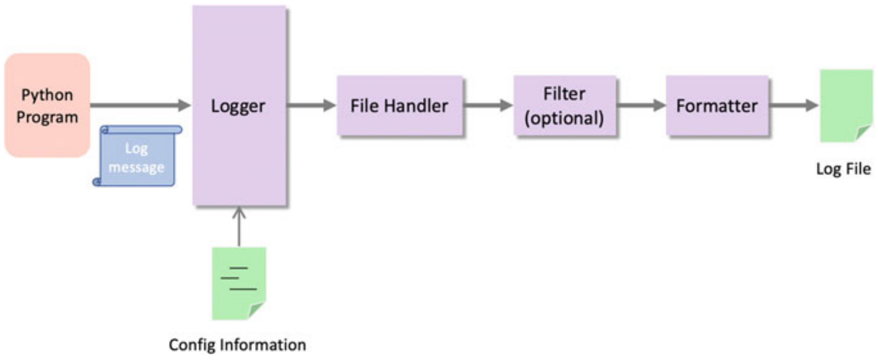
Chapter 27
Logging in Python

27.1 The Logging Module

Python has included a built-in logging module since Python 2.3. This module, the `logging` module, defines functions and classes which implement a flexible logging framework that can be used in any Python application/script or in Python libraries/modules.

Although different logging frameworks differ in the specific details of what they offer; almost all offer the same core elements (although different names are sometimes used). The Python logging module is no different and the core elements that make up the logging framework and its processing pipeline are shown below (note that a very similar diagram could be drawn for logging frameworks in Java, Scala, C++ etc.).

The following diagram illustrates a Python program that uses the built-in Python logging framework to log messages to a file.



The core elements of the logging framework (some of which are optional) are shown above and described below:

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_27

- **Log Message** The is the message to be logged from the application.
- **Logger** Provides the programmers entry point/interface to the logging system. The Logger class provides a variety of methods that can be used to log messages at different levels.
- **Handler** Handlers determine where to send a log message, default handlers include file handlers that send messages to a file and HTTP handlers that send messages to a web server.
- **Filter** This is an optional element in the logging pipeline. They can be used to further filter the information to be logged providing fine grained control of which log messages are actually output (for example to a log file).
- **Formatter** These are used to format the log message as required. This may involve adding timestamps, module and function/method information etc. to the original log message.
- **Configuration Information** The logger (and associated handlers, filters and formatters) can be configured either programmatically in Python or through configuration files. These configuration files can be written using key-value pairs or in a YAML file (which is a simple mark up language). YAML stands for Yet Another Markup Language!

It is worth noting that much of the logging framework is hidden from the developer who really only sees the logger; the remainder of the logging pipeline is either configured by default or via log configuration information typically in the form of a log configuration file.

27.2 The Logger

The Logger provides the programmers interface to the logging pipeline.

A Logger object is obtained from the `getLogger()` function defined in the `logging` module. The following code snippet illustrates acquiring the default logger and using it to log an error message. Note that the `logging` module must be imported:

```
import logging

logger = logging.getLogger()

logger.error('This should be used with something unexpected')
```

The output from this short application is logged to the console as this is the default configuration:

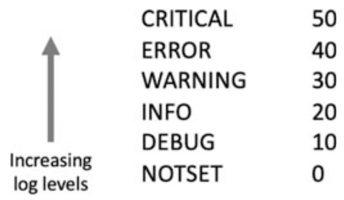
```
This should be used with something unexpected
```

27.3 Controlling the Amount of Information Logged

Log messages are actually associated with a log level. These log levels are intended to indicate the severity of the message being logged. There are six different log levels associated with the Python logging framework, these are:

- **NOTSET** At this level no logging takes place and logging is effectively turned off.
- **DEBUG** This level is intended to provide detailed information, typically of interest when a developer is diagnosing a bug or issues within an application.
- **INFO** This level is expected to provide less detail than the DEBUG log level as it is expected to provide information that can be used to confirm that the application is working as expected.
- **WARNING** This is used to provide information on an unexpected event or an indication of some likely problem that a developer or system administration might wish to investigate further.
- **ERROR** This is used to provide information on some serious issue or problem that the application has not been able to deal with and that is likely to mean that the application cannot function correctly.
- **CRITICAL** This is the highest level of issue and is reserved for critical situations such as ones in which the program can no longer continue executing.

The log levels are relative to one another and defined in a hierarchy. Each log level has a numeric value associated with it as shown below (although you should never need to use the numbers). Thus `INFO` is a higher log level than `DEBUG`, in turn `ERROR` is a higher log level than `WARNING`, `INFO`, `DEBUG` etc.


Increasing log levels

CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

Associated with the log level that a message is logged with, a logger also has a log level associated with it. The logger will process all messages that are at the loggers log level or above that level. Thus if a logger has a log level of `WARNING` then it will log all messages logged using the `warning`, `error` and `critical` log levels.

Generally speaking, an application will not use the `DEBUG` level in a production system. This is usually considered inappropriate as it is only intended for debug scenarios. The `INFO` level may be considered appropriate for a production system although it is likely to produce large amounts of information as it typically traces the execution of functions and methods. If an application has been well tested and verified then it is only really warnings and errors which should occur/be of concern. It is therefore not uncommon to default to the `WARNING` level for production systems (indeed this is why the default log level is set to WARNING within the Python logging system).

If we now look at the following code that obtains the default logger object and then uses several different logger methods, we can see the effect of the log levels on the output:

```
import logging

logger = logging.getLogger()

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')
```

The default log level is set to *warning*, and thus only messages logged at the warning level or above will be printed out:

```
This is a warning!
This should be used with something unexpected
Something serious
```

As can be seen from this, the messages logged at the debug and info level have been ignored.

However, the Logger object allows us to change the log level programmatically using the setLevel() method, for example logger.setLevel(logging. DEBUG) or via the logging.basicConfig(level = logging.DEBUG) function; both of these will set the logging level to DEBUG. Note that the log level must be set before the logger is obtained.

If we add one of the above approaches to setting the log level to the previous program we will change the amount of log information generated:

```
import logging

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger()

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

This will now output all the log messages as `debug` is the lowest logging level. We can of course turn off logging by setting the log level to `NOTSET`

```
logger.setLevel(logging.NOTSET)
```

Alternatively you can set the Loggers `disabled` attribute to `True`:

```
logging.Logger.disabled = True
```

27.4 Logger Methods

The Logger class provides a number of methods that can be used to control what is logged including:

- `setLevel(level)` Sets this loggers log level.
- `getEffectiveLevel()` Returns this loggers log level.
- `isEnabledFor(level)` Checks to see if this logger is enabled for the log level specified.
- `debug(message)` logs messages at the debug level.
- `info(message)` logs messages at the info level.
- `warning(message)` logs messages at the warning level.
- `error(message)` logs messages at the error level.
- `critical(message)` logs messages at the critical level.
- `exception(message)` This method logs a message at the error level. However, it can only be used within an exception handler and includes a stack trace of any associated exception, for example:

```
import logging

logger = logging.getLogger()

try:
    print('starting')
    x = 1 / 0
    print(x)
except:
    logger.exception('an exception message')

print('Done')
```

- `log(level, message)` logs messages at the log level specified as the first parameter.

In addition there are several methods that are used to manage handlers and filters:

- `addFilter(filter)` This method adds the specified filter filter to this logger.
- `removeFilter(filter)` The specified filter is removed from this logger object.
- `addHandler(handler)` The specified handler is added to this logger.
- `removeHandler(handler)` Removes the specified handler from this logger.

27.5 Default Logger

A default (or *root*) logger is always available from the logging framework.

This logger can be accessed via the functions defined in the `logging` module. These functions allow messages to be logged at different levels using methods such as `info()`, `error()`, `warning()` but without the need to obtain a reference to a logger object first. For example:

```
import logging

# Set the root logger level
logging.basicConfig(level=logging.DEBUG)

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

This example sets the logging level for the root or default logger to `DEBUG` (the default is `WARNING`). It then uses the default logger to generate a range of log messages at different levels (from `DEBUG` up to `CRITICAL`). The output from this program is given below:

```
DEBUG:root:This is to help with debugging
INFO:root:This is just for information
WARNING:root:This is a warning!
ERROR:root:This should be used with something unexpected
CRITICAL:root:Something serious
```

Note that the format used by default with the root logger prints the log level, the name of the logger generating the output and the message. From this you can see that it is the root longer that is generating the output.

27.6 Module Level Loggers

Most modules will not use the root logger to log information, instead they will use a *named* or *module level* logger. Such a logger can be configured independently of the root logger. This allows developers to turn on logging just for a module rather than for a whole application. This can be useful if a developer wishes to investigate an issue that is located within a single module.

Previous code examples in this chapter have used the `getLogger()` function with no parameters to obtain a logger object, for example:

```
logger = logging.getLogger()
```

This is really just another way of obtaining a reference to the root logger which is used by the stand alone logging functions such as `logging.info()`, `logging.debug()` function, thus:

```
logging.warning('my warning')
```

and

```
logger = logging.getLogger()
logger.warning('my warning')
```

Have exactly the same effect; the only difference is that the first version involves less code.

However, it is also possible to create a *named* logger. This is a separate logger object that has its own name and can potentially have its own log level, handlers and formatters etc. To obtain a named logger pass a *name* string into the `getLogger()` method:

```
logger1 = logging.getLogger('my logger')
```

This returns a logger object with the name ‘my logger’. Note that this may be a brand new logger object, however if any other code within the current system has previously requested a logger called ‘my logger’ then that logger object will be returned to the current code. Thus multiple calls to `getLogger()` with the same name will always return a reference to the same Logger object.

It is common practice to use the name of the module as the name of the logger; as only one module with a specific name should exist within any specific system. The name of the module does not need to be hard coded as it can be obtained using the `__name__` module attribute, it is thus common to see:

```
logger2 = logging.getLogger(__name__)
```

We can see the effect of each of these statements by printing out each logger:

```
logger = logging.getLogger()
print('Root logger:', logger)

logger1 = logging.getLogger('my logger')
print('Named logger:', logger1)

logger2 = logging.getLogger(__name__)
print('Module logger:', logger2)
```

When the above code is run the output is:

```
Root logger: <RootLogger root (WARNING)>
Named logger: <Logger my logger (WARNING)>
Module logger: <Logger __main__ (WARNING)>
```

This shows that each logger has their own name (the code was run in the main module and thus the module name was `__main__`) and all three loggers have an effective log level of `WARNING` (which is the default).

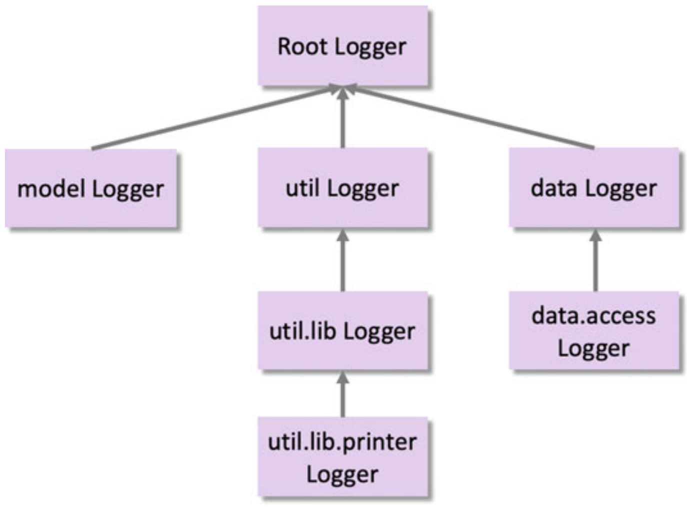
27.7 Logger Hierarchy

There is in fact a hierarchy of loggers with the root logger at the top of this hierarchy.

All *named* loggers are below the root logger.

The name of a logger can actually be a period-separated hierarchical value such as `util`, `util.lib` and `util.lib.printer`. Loggers that are further down the hierarchy are children of loggers further up the logger hierarchy.

For example given a logger called `lib`, then it will be below the `root` logger but above the logger with the name `util.lib`. This logger will in turn be above the logger called `util.lib.printer`. This is illustrated in the following diagram:



The logger name hierarchy is analogous to the Python package hierarchy, and identical to it if you organise your loggers on a per-module basis using the recommended construction `logging.getLogger(__name__)`.

This hierarchy is important when considering the log level. If a log level has not been set for the current logger then it will look to its parent to see if that logger has a log level set. If it does that will be the log level used. This search back up the logger hierarchy will continue until either an explicit log level is found or the root logger is encountered which has a default log level of WARNING.

This is useful as it is not necessary to explicitly set the log level for every logger object used in an application. Instead it is only necessary to set the root log level (or for a module hierarchy an appropriate point in the module hierarchy). This can then be overridden where specifically required.

27.8 Formatters

The are two levels at which you can format the messages logged, these are within the log message passed to a logging method (such as `info()` or `warn()`) and via the top level configuration that indicates what additional information may be added to the individual log message.

27.8.1 Formatting Log Messages

The log message can have control characters that indicate what values should be placed within the message, for example:

```
logger.warning('%s is set to %d', 'count', 42)
```

This indicates that the format string expects to be given a string and a number. The parameters to be substituted into the format string follow the format string as a comma separated list of values.

27.8.2 Formatting Log Output

The logging pipeline can be configured to incorporate standard information with each log message. This can be done globally for all handlers. It is also possible to programmatically set a specific formatter on a individual handler; this is discussed in the next section.

To globally set the output format for log messages use the `logging.basicConfig()` function using the named parameter `format`.

The `format` parameter takes a string that can contain LogRecord attributes organised as you see fit. There is a comprehensive list of LogRecord attributes which can be referenced at https://docs.python.org/3/library/logging.html#logrecord-attributes. The key ones are:

- `args` a tuple listing the arguments used to call the associated function or method.
- `asctime` indicates the time that the log message was created.
- `filename` the name of the file containing the log statement.
- `module` the module name (the name portion of the filename).
- `funcName` the name of the function or method containing the log statement.
- `levelname` the log level of the log statement.
- `message` the log message itself as provided to the log method.

The effect of some of these are illustrated below.

```
import logging

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

logger = logging.getLogger(__name__)

def do_something():
    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')

do_something()
```

The above program generates the following log statements:

```
2019-02-20 16:50:34,084 This is to help with debugging
2019-02-20 16:50:34,084 This is just for information
2019-02-20 16:50:34,085 This is a warning!
2019-02-20 16:50:34,085 This should be used with something
unexpected
2019-02-20 16:50:34,085 Something serious
```

However, it might be useful to know the log level associated with the log statements, as well as the function that the log statements were called from. It is possible to obtain this information by changing the format string passed to the `logging.basicConfig()` function:

```
logging.basicConfig(format='%(asctime)s[%(levelname)s]
%(funcName)s: %(message)s', level=logging.DEBUG)
```

Which will now generate the output within log level information and the function involved:

```
2019-02-20 16:54:16,250[DEBUG] do_something: This is to help
with debugging
2019-02-20 16:54:16,250[INFO] do_something: This is just for
information
2019-02-20 16:54:16,250[WARNING] do_something: This is a
warning!
2019-02-20 16:54:16,250[ERROR] do_something: This should be
used with something unexpected
2019-02-20 16:54:16,250[CRITICAL] do_something: Something
serious
```

We can even control the format of the date time information associated with the log statement using the `datafmt` parameter of the `logging.basicConfig()` function:

```
logging.basicConfig(format='%(asctime)s %(message)s',
datefmt='%m/%d/%Y %I:%M:%S %p', level=logging.DEBUG)
```

This format string uses the formatting options used by the `datetime.strptime()` function (see https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior) for information on the control characters, in this case

- %m—Month as a zero-padded decimal number e.g. 01, 11, 12.
- %d—Day of the month as a zero-padded decimal number e.g. 01, 12 etc.
- %Y—Year with century as a decimal number e.g. 2020.
- %I—Hour (12-h clock) as a zero-padded decimal number e.g. 01, 10 etc.
- %M—Minute as a zero-padded decimal number e.g. 0, 01, 59 etc.
- %S—Second as a zero-padded decimal number e.g. 00, 01, 59 etc.
- %p—Either AM or PM.

Thus the output generated using the above `datefmt` string is:

```
02/20/2019 05:05:18 PM This is to help with debugging
02/20/2019 05:05:18 PM This is just for information
02/20/2019 05:05:18 PM This is a warning!
02/20/2019 05:05:18 PM This should be used with something
unexpected
02/20/2019 05:05:18 PM Something serious
```

To set a formatter on an individual handler see the next section.

27.9 Online Resources

For further information on the Python logging framework see the following:

- https://docs.python.org/3/library/logging.html The standard library documentation on the logging facilities in Python.
- https://docs.python.org/3/howto/logging.html A how to guide on logging from the Python standard library documentation.
- https://pymotw.com/3/logging/index.html Python Module of the Week logging page.

27.10 Exercises

This exercise will involve adding logging to the `Account` class you have been working on in this book.

You should add log methods to each of the methods in the class using either the `debug` or `info` methods. You should also obtain a module logger for the account classes.