4.1 Logging in Python

Logging in Python

The Python Standard Library provides a useful module called `logging` to log events occurring in the application. Logs are most often used to find the cause of an error. By default, Python and its modules provide many logs informing you of the causes of errors. However, it's good practice to create your own logs that may be useful to you or other programmers.

An example of using your own logs can be any Internet system. When users visit your site, you can log information about the browsers they use. If something goes wrong, you'll be able to determine in which browsers the problem is occurring.

In Python, you can store logs in different places. Most often it's in the form of a file, but it can also be an output stream, or even an external service. To start logging, we need to import the appropriate module:

```
import logging
```

In this part of the course, you'll learn how to create logs using the logging module. See what this module offers and start using it to become a better programmer.

The Logger object

One application may have several loggers created both by us and by programmers of the modules. If your application is simple, as in the example below, you can use the root logger. To do this, call the `getLogger` function without providing a name. The root logger is at the highest point in the hierarchy. Its place in the hierarchy is assigned based on the names passed to the `getLogger` function.

```
import logging

logger = logging.getLogger()
hello_logger = logging.getLogger('hello')
hello_world_logger = logging.getLogger('hello.world')
recommended_logger = logging.getLogger(__name__)
```

Logger names are similar to the names of the Python modules in which the dot separator is used. Their format is as follows:

hello – creates a logger which is a child of the root logger;

hello.world – creates a logger which is a child of the hello logger.

If you want to make another nesting, just use the dot separator.

The `getLogger` function returns a `Logger` object. Let's look at the example code in the editor. We'll find there the ways to get the Logger object, both with and without a name.

We recommend calling the `getLogger` function with the `name` argument, which is replaced by the current module name. This allows you to easily specify the source of the logged message.

NOTE: Several calls to the `getLogger` function with the same name will always return the same object.

Logging levels

The `Logger` object allows you to create logs with different levels of logging that help you to distinguish between less important logs and those reporting a serious error. By default, the following logging levels are defined:

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

Each level has a name and a numeric value. You can also define your own level, but those offered by the `logging` module are quite sufficient. The `Logger` object has methods that set the logging level for you. Take a look at the example in the editor.

```
import logging

logging.basicConfig()

logger = logging.getLogger()

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

Result:

```
CRITICAL:root:Your CRITICAL message
ERROR:root:Your ERROR message
WARNING:root:Your WARNING message
```

All of the above methods require you to provide a message that will be visible in the logs. The default log format includes the level, the logger name and the message you've defined. Note that all these values are separated by a colon. Later in this course, you'll learn how to change the default formatting.

You're probably wondering why messages with `INFO` and `DEBUG` levels are not displayed. This is due to the default configuration, which we'll talk about in a moment.

NOTE: The `basicConfig` method will be discussed later in the course. For now, remember that it's responsible for the basic logging configuration.

The setLevel method

The root logger has the logging level set to `WARNING`. This means that messages at the `INFO` or `DEBUG` levels aren't processed.

Sometimes you may want to change this behavior, especially if you create your own logger. To do this, you need to pass a logging level to the `setLevel` method. See how we do this in the editor.

```
import logging
```

```
logging.basicConfig()

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

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

Result:

```
CRITICAL:root:Your CRITICAL message
ERROR:root:Your ERROR message
WARNING:root:Your WARNING message
INFO:root:Your INFO message
DEBUG:root:Your DEBUG message
```

Setting the DEBUG level causes messages with this or a higher level to be logged. It's worth mentioning that loggers created using the name argument have the NOTSET level set by default. In this case, their logging level is set based on the parent levels, starting from the closest parent to the root logger.

If the closest parent has a level set to NOTSET, the logger level is set based on the levels of subsequent parents in the hierarchy. Level setting ends if a parent has a level other than NOTSET. If none of the visited parents has a level other than NOTSET, then all messages will be processed regardless of their level.

Basic configuration (part 1)

As we mentioned before, the basic logging configuration is done using the `basicConfig` method. Calling the `basicConfig` method (without specifying any arguments) creates a `StreamHandler` object that processes the logs and then displays them in the console.

The `StreamHandler` object is created by the default `Formatter` object responsible for the log format. As a reminder, the default format consists of the level name, logger name, and defined message. Finally the newly created handler is added to the root logger. Later you'll learn how to create your own handler and formatter.

In the previous examples, we called the `basicConfig` method without any arguments. Using the `basicConfig`, method you can change the logging level (in the same way as using the `setLevel` method) and even the location of the logs. Take a look at the example in the editor.

```
import logging

logging.basicConfig(level=logging.CRITICAL, filename='prod.log', filemode='a')

logger = logging.getLogger()

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

Result in `prod.log` file:

```
CRITICAL:root:Your CRITICAL message
```

In the example, the `basicConfig` method takes three arguments. The first one is the logging level equal to `CRITICAL`, which means that only messages with this level will be processed.

Passing a filename to the second argument creates a `FileHandler` object (instead of a `StreamHandler` object). As you've probably noticed, the logs no longer appear in the console. After setting the `filename` argument, all logs will be directed to the specified file.

In addition, passing the last `filemode` argument with the value 'a' (this is the default mode) means that new logs will be appended to this file. If you'd like to change this mode, you can use other modes that are analogous to those used in the built-in ''open ''function.

These aren't all the arguments that the `basicConfig` method can take. Are you ready for another dose of knowledge? Let's move on!

NOTE: The `basicConfig` method changes the configuration of the root logger and its children who don't have their own handler defined.

Basic configuration (part 2)

The `basicConfig` method presented earlier can also be used to change the default log formatting. This is done using the `format` argument, which can be defined using any characters or attributes of the `LogRecord` object. Let's explain it with the example in the editor.

```
import logging

FORMAT = '%(name)s:%(levelname)s:%(asctime)s:%(message)s'

logging.basicConfig(level=logging.CRITICAL, filename='prod.log', filemode='a',
format=FORMAT)

logger = logging.getLogger()

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

Result in the `prod.log` file:

```
root:CRITICAL:2019-10-10 17:16:46,293:Your CRITICAL message
```

The format we define is created by combining the attributes of the `LogRecord` object separated by a colon. The `LogRecord` object is automatically created by the logger during logging. It contains many attributes, such as the name of the logger, the logging level, or even the line number in which the logging method is called. A full list of all available attributes can be found here logrecord-attributes].

In our example, we use the following attributes:ç

- **%(name)s** – this pattern will be replaced by the name of the logger that calls the logging method. In our case, it's the root logger;

- **%(levelname)s** – this pattern will be replaced with the set login level. In our case, this is the CRITICAL level;
- **%(asctime)s** – this pattern will be replaced with a human-readable date format that indicates when the LogRecord object was created. The decimal value is expressed in milliseconds;
- **%(message)s** – this pattern will be replaced by the defined message. In our case, it's 'Your CRITICAL message'.

In general, the scheme for using the `LogRecord` object argument in the format argument looks like this:

```
(LOG_RECORD_ATTRIBUTE_NAME)s
```

Your first handler

Each logger can save logs in different locations as well as in different formats. To do this, you must define your own handler and formatter.

In most cases, you'll want to save your logs to a file. The logging module has the `FileHandler` class, which facilitates this task. When creating a `FileHandler` object, you must pass a filename where the logs will be saved.

Additionally, you can pass a file `mode` with the mode argument, e.g., mode='a'. In the next step, you should set the logging level that will be processed by the handler. By default, the newly created handler is set to the `NOTSET` level. You can change this using the `setLevel` method. In the example in the editor, we've set the CRITICAL level.

Finally, you need to add the created handler to your logger using the `addHandler` method.

```
import logging

logger = logging.getLogger(__name__)

handler = logging.FileHandler('prod.log', mode='w')
handler.setLevel(logging.CRITICAL)

logger.addHandler(handler)

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

Result in the `prod.log` file:

```
Your CRITICAL message
```

If you check the `prod.log` file, you'll see that only the message is saved there. Do you know what we forgot? Your handler hasn't created a formatter. You'll learn how to do this in a moment.

NOTE: Each logger can have several handlers added. One handler can save logs to a file, while another can send them to an external service. In order to process messages with a level lower than `WARNING` by added handlers, it's necessary to set this level threshold in the root logger.

Your first formatter

Congratulations! You've just created your first handler. Only the formatter is missing, but don't worry. It's just two steps. Take a look at the example in the editor.

```
import logging

FORMAT = '%(name)s:%(levelname)s:%(asctime)s:%(message)s'

logger = logging.getLogger(__name__)

handler = logging.FileHandler('prod.log', mode='w')
handler.setLevel(logging.CRITICAL)

formatter = logging.Formatter(FORMAT)
handler.setFormatter(formatter)

logger.addHandler(handler)

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

Result in the prod.log file:

```
__main__:CRITICAL:2019-10-10 20:40:05,119:Your CRITICAL message
```

In the first step you need to create a `Formatter` object by passing the format you've defined to its constructor. In the example, we use the format defined in one of the previous examples.

The next step is to set the formatter in the handler object. This is done using the `setFormatter` method. After doing this, you can analyze your logs in the `prod.log` file.