About this notebook

Avoid running again the cells of this notebook.

Loading [MathJax]/extensions/Safe.js

Introduction

A running code can be monitored using carefully placed `print` statements. Yet, this solution does not allow to choose:

- Whether or not the messages must be printed during an execution. To do this, one must comment out or uncomment the `print` statements.
- Where are the messages printed: standard output (terminal) and/or files(s) on disk.

The `logging` library solves these problems.

Exemple

Loading [MathJax]/extensions/Safe.js

Setting up a logger

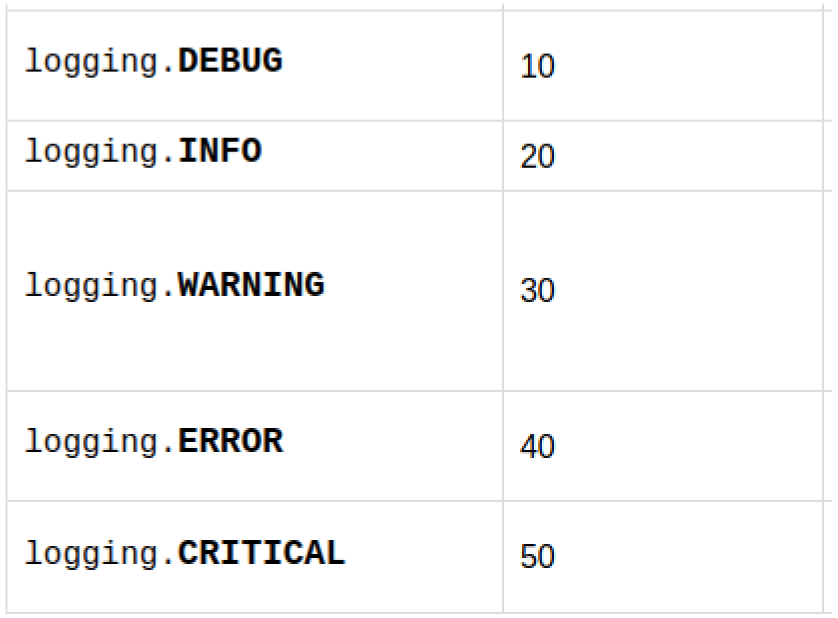
In [1]:
```python
import logging

def define_logger(base_level):
    logger = logging.getLogger()
    logger.setLevel(base_level)

    overview_handler = logging.StreamHandler()
    overview_handler.setLevel(logging.INFO)
    logger.addHandler(overview_handler)

    problem_handler = logging.StreamHandler()
    problem_handler.setLevel(logging.WARNING)
    logger.addHandler(problem_handler)

    return logger

logger = define_logger(base_level=logging.INFO)
```

The `logger` is the base object used to write messages. When it receives a message whose level is higher than `base_level`, the `logger` shares it with its `handler` instances. here, 2 handlers are defined:

- `overview_handler` will write messages whose level is at least `INFO`. `INFO` messages describe simply what is going on in the code.
- `problem_handler` will write messages whose level is at least `WARNING`. `WARNING` messages tell the user about a small problem.

Both these handlers publish their messages in the output console ("`StreamHandler`").

Then, what is the levels hierarchy? It is given in the documentation page of the logging library:



`logging.DEBUG`	10
`logging.INFO`	20
`logging.WARNING`	30
`logging.ERROR`	40
`logging.CRITICAL`	50

These levels are internally represented as integers, but one must use the syntax `logging.[...]` instead.

Using a logger

Using the previous example regarding data acquisition:

In [2]:

```python
from random import randint
from time import sleep

def sensor_reading(only_valid_data=True):
    # fake real-time data acquisition
    sleep(1)
    new_value = randint(-10, 10)
    if new_value < 0 and only_valid_data:
        raise ValueError(f"Sensor default, got negative value {new_value}.")
    return new_value

def data_acquisition(logger, replacement_value=0):
    data = []
    for k in range(5):
        try:
            new_value = sensor_reading()
            logger.info(f"Time step {k}: good value: {new_value}")

        except ValueError as e:
            new_value = replacement_value
            logger.warning(f"Time step {k}: bad value was replaced with {replacement_

        data.append(new_value)
    return data
```

Loading [MathJax]/extensions/Safe.js

```
In [3]: data_acquisition(logger, 0)
```

```
Time step 0: good value: 9
Time step 1: bad value was replaced with 0
Time step 1: bad value was replaced with 0
Time step 2: good value: 1
Time step 3: good value: 5
Time step 4: good value: 4
```

```
Out[3]: [9, 0, 1, 5, 4]
```

In the output console ('*Stream*'), we can notice two different behaviours:

- whenever acquisition succeeds (`value` >=0), a message with level `logging.INFO` is printed. Among the two defined handlers, only `overview_handler` prints this message since the level of `problem_handler` (`logging.WARNING`) is higher than `logging.INFO`.
- whenever acquisition fails, both handlers print the failure message because it is published with a level `logging.WARNING`. Thus this message is printed two times.

Loading [MathJax]/extensions/Safe.js

Advanced features

Defining two handlers having the same output is not that interesting. hereafter, the `overview_handler` is modified so that its messages are written to a file.

Moreover, some information are added to the logged messages. This is done using a `__Formatter__` object:

- The time stamp of message production
- The level of the message

The information that can be added to each message are described in the documentation. Note that the `style` argument tells that the formatting syntax is `{}` (see `fmt` argument).

Loading [MathJax]/extensions/Safe.js

In [2]:
```python
import logging


def define_logger2(base_level):
    logger = logging.getLogger()
    logger.setLevel(base_level)
    formatter = logging.Formatter(fmt='{asctime} :: {levelname} :: {message}',
                                  datefmt='%H:%M',
                                  style='{')

    overview_handler = logging.FileHandler("overview_log.txt", mode="w")
    overview_handler.setLevel(logging.INFO)
    overview_handler.setFormatter(formatter)
    logger.addHandler(overview_handler)

    problem_handler = logging.StreamHandler()
    problem_handler.setLevel(logging.WARNING)
    problem_handler.setFormatter(formatter)
    logger.addHandler(problem_handler)

    return logger

logger2 = define_logger2(base_level=logging.INFO)
```

In [3]:
```
data_acquisition(logger2, 0)
```

```
19:38 :: WARNING :: Time step 0: bad value was replaced with 0
```

Out[3]:
```
[0, 1, 2, 6, 8]
```

In [6]:
```
with open("overview_log.txt", "r") as file:
    content = file.read()
    print(content)
```

```
19:38 :: WARNING :: Time step 0: bad value was replaced with 0
19:38 :: INFO :: Time step 1: good value: 1
19:38 :: INFO :: Time step 2: good value: 2
19:38 :: INFO :: Time step 3: good value: 6
19:38 :: INFO :: Time step 4: good value: 8
```

Loading [MathJax]/extensions/Safe.js

We can notice that:

- The console output is limited to messages having a level `logging.WARNING`.
- The file *overview_log.txt* contains both levels of messages.

Loading [MathJax]/extensions/Safe.js