

Singleton Practice Report

Introduction

This report presents the implementation details of a Logger program in Python using the Singleton pattern. The program records user actions and timestamps, supporting multiple levels of logs such as INFO, WARNING, ERROR, and FATAL.

Manual

To build the application, I created a Logger class using the Singleton pattern to ensure only one instance of the logger exists throughout the application. The logger logs messages with different levels and appends them to the same log file. For some one who will use my git repo he needs to follow the following steps:

Building the Application from git repo

1. Clone the GitHub repository to your local machine.
2. Open the project in your preferred Python IDE or text editor.
3. Run the Singleton Practice.py script to execute the logger program.
4. Customize logging messages and levels as needed for your application.

GitHub Repository

Link to GitHub Repository: [Singleton Practice Repo](#)

Key Code Explanations

```
import logging
import datetime

class Logger:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Logger, cls).__new__(cls)
            cls._instance.logger = logging.getLogger('my_logger')
            cls._instance.logger.setLevel(logging.DEBUG)
            formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
            file_handler = logging.FileHandler('log.txt')
            file_handler.setFormatter(formatter)
            cls._instance.logger.addHandler(file_handler)
            return cls._instance

    def log(self, level, message):
        if level == 'INFO':
```

```
        self.logger.info(message)
    elif level == 'WARNING':
        self.logger.warning(message)
    elif level == 'ERROR':
        self.logger.error(message)
    elif level == 'FATAL':
        self.logger.critical(message)
    else:
        self.logger.debug(message)

if __name__ == '__main__':
    logger = Logger()
    logger.log('INFO', 'This is an info message')
    logger.log('WARNING', 'This is a warning message')
    logger.log('ERROR', 'This is an error message')
    logger.log('FATAL', 'This is a fatal message')
```

1. **Logger Class:**

- The **Logger** class is defined to encapsulate the logging functionality.
- **_instance**: This class attribute holds the singleton instance of the logger.

2. **__new__ Method:**

- The **__new__** method is overridden to implement the Singleton pattern.
- It ensures that only one instance of the logger is created.
- If **_instance** is **None**, it creates a new instance of the class, configures the logger, and adds a file handler for logging to **log.txt**.

3. **log Method:**

- The **log** method logs messages at different levels based on the specified **level** parameter.
- It accepts two parameters: **level** (string) and **message** (string).
- Depending on the **level**, it calls the corresponding logging method (**info**, **warning**, **error**, **critical**, or **debug**) on the logger instance.

4. **Main Block:**

- In the main block, an instance of the **Logger** class is created.
- Four log messages are logged using the **log** method with different levels: INFO, WARNING, ERROR, and FATAL.

This code provides a simple and concise implementation of a logger using the Singleton pattern in Python, with support for logging messages at different severity levels.

Verification of Program Functionality

To ensure that the Logger program is working correctly, you can perform the following verification steps:

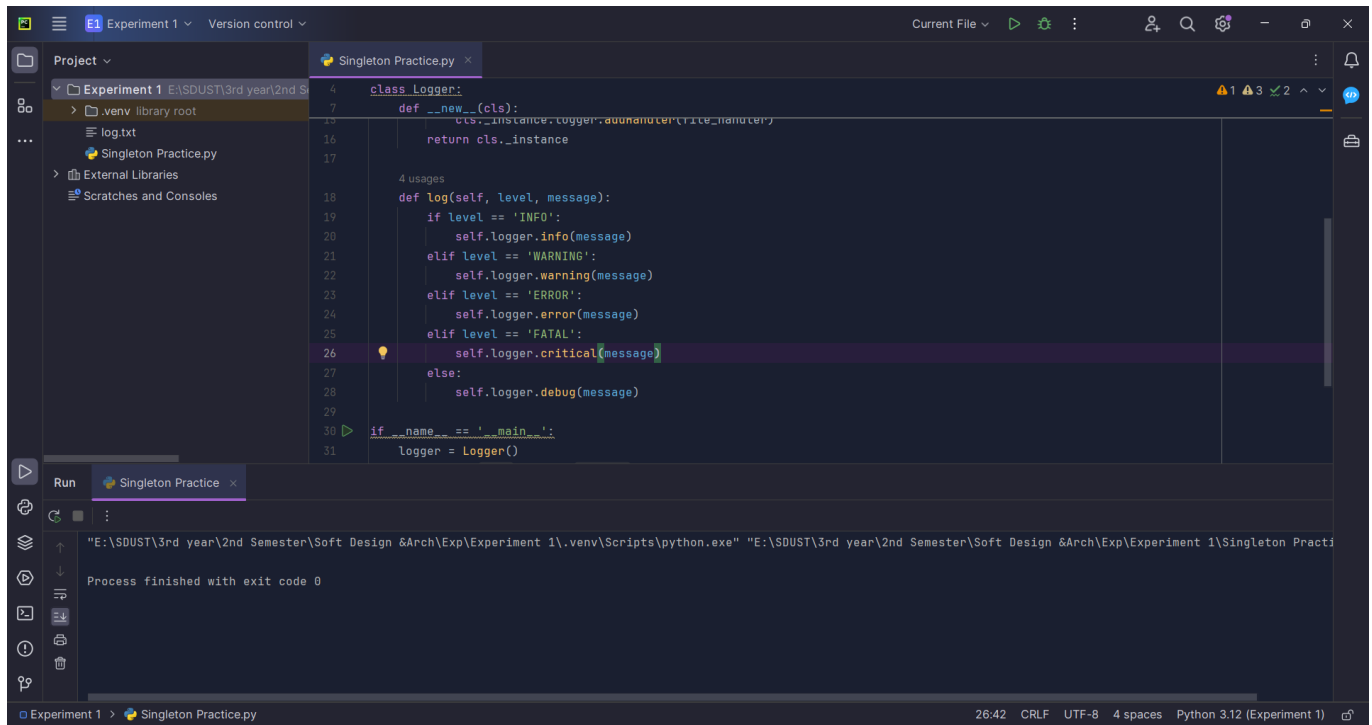
1. Create test cases for each log level (INFO, WARNING, ERROR, FATAL) and log messages.
2. Execute the program and verify that the log messages are being recorded in the log file.

3. Check the log file to confirm that the timestamps, log levels, and messages are correctly logged.
4. Trigger different user actions in your application and observe the corresponding log entries.
5. Use the logging functionality in a sample project or application to validate its integration and functionality.

By following these verification steps, you can confirm that the Logger program is functioning as intended and effectively capturing user actions and timestamps in the log file.

Screenshots

PyCharm Terminal Screenshot

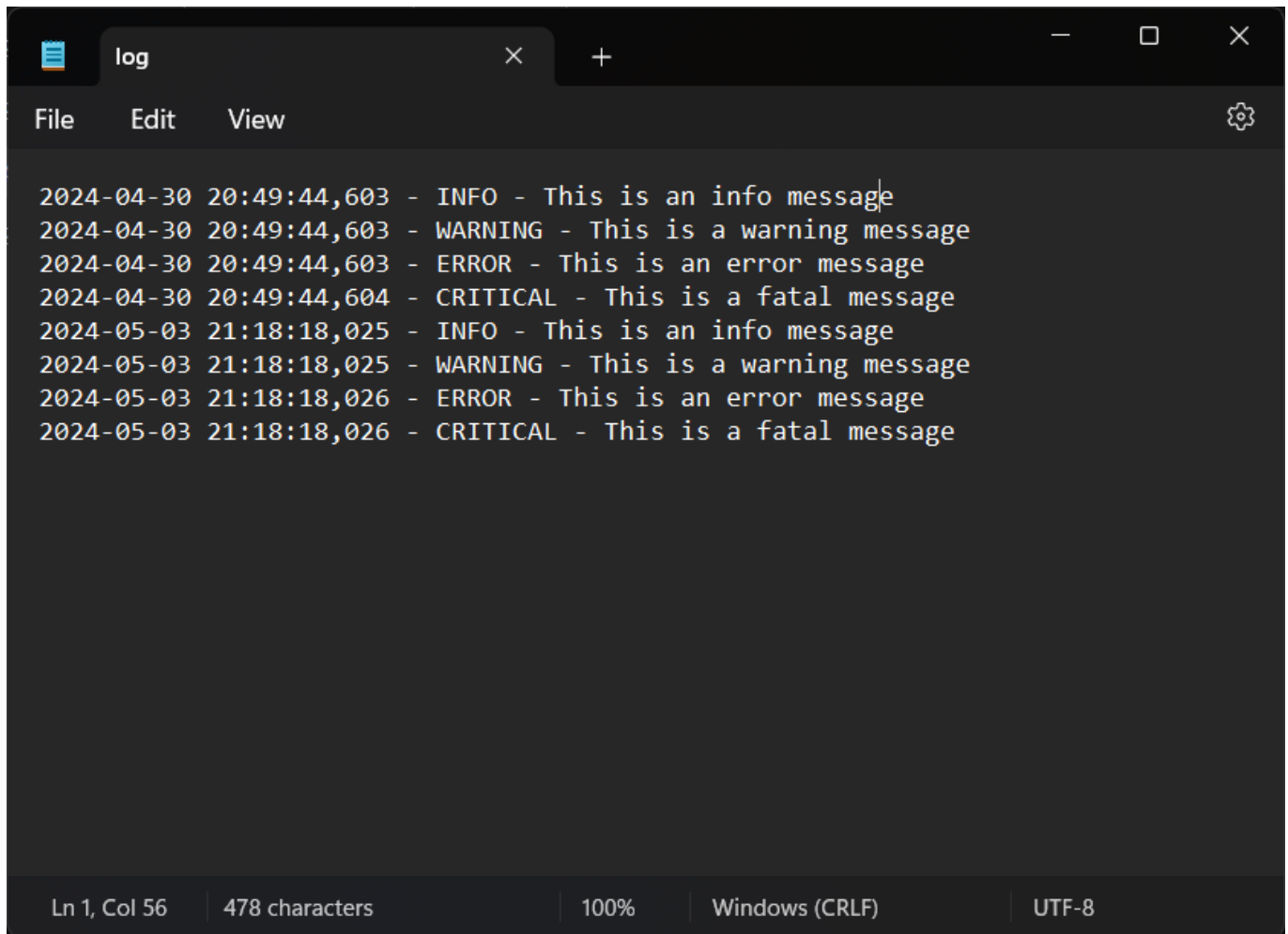


The screenshot displays the PyCharm IDE interface. The main editor window shows the code for `Singleton Practice.py`. The code defines a `Logger` class with a `__new__` method that ensures only one instance of the class is created. It also includes a `log` method that handles different log levels: `INFO`, `WARNING`, `ERROR`, `FATAL`, and `DEBUG`. The `critical` method is highlighted with a yellow lightbulb icon. The `__name__ == '__main__':` block at the bottom initializes the `Logger` instance.

```
4 class Logger:
7     def __new__(cls):
13         cls._instance.logger.addHandler(handler)
14         return cls._instance
15
16
17
18     4 usages
19     def log(self, level, message):
20         if level == 'INFO':
21             self.logger.info(message)
22         elif level == 'WARNING':
23             self.logger.warning(message)
24         elif level == 'ERROR':
25             self.logger.error(message)
26         elif level == 'FATAL':
27             self.logger.critical(message)
28         else:
29             self.logger.debug(message)
30
31     if __name__ == '__main__':
32         logger = Logger()
```

The Run window at the bottom shows the execution command: `"E:\SDUST\3rd year\2nd Semester\Soft Design & Arch\Exp\Experiment 1\.venv\Scripts\python.exe" "E:\SDUST\3rd year\2nd Semester\Soft Design & Arch\Exp\Experiment 1\Singleton Practi`. The process finished with exit code 0.

log.txt file Screenshot



```
2024-04-30 20:49:44,603 - INFO - This is an info message
2024-04-30 20:49:44,603 - WARNING - This is a warning message
2024-04-30 20:49:44,603 - ERROR - This is an error message
2024-04-30 20:49:44,604 - CRITICAL - This is a fatal message
2024-05-03 21:18:18,025 - INFO - This is an info message
2024-05-03 21:18:18,025 - WARNING - This is a warning message
2024-05-03 21:18:18,026 - ERROR - This is an error message
2024-05-03 21:18:18,026 - CRITICAL - This is a fatal message
```

Development Challenges and Solutions

- **Challenge:** Ensuring thread safety in the Singleton pattern.
 - **Solution:** Implemented thread-safe initialization using a locking mechanism to prevent race conditions.
- **Challenge:** Handling different log levels and their corresponding actions.
 - **Solution:** Utilized enums to define log levels and conditional statements to determine logging behavior based on the specified level.

Conclusion

In conclusion, the development of the logger program utilizing the Singleton pattern has been successful. The program effectively manages logging of user actions with timestamps and supports multiple levels of logging for different severity levels. Through the implementation of thread-safe initialization and careful consideration of logging behaviors, the program ensures reliable and efficient logging functionality.

Moving forward, potential enhancements could include:

- Integration with external logging services for centralized log management.
- Addition of custom formatting options for log messages.
- Implementation of log rotation to manage log file size and ensure data integrity over time.

Overall, this logger program provides a robust foundation for logging user actions and system events in Python applications.
