

Logging

Sidheswar Routray
Department of Computer Science & Engineering
School of Technology

What is Logging

- Logging is the process of tracking events that happen when some software runs. The logging module in Python is used for tracking events that occur during execution.
- The software's developer adds logging calls to their code to indicate that certain events have occurred.
- An event is described by a descriptive message which can optionally contain variable data.

Convenience Function

[debug\(\)](#), [info\(\)](#), [warning\(\)](#), [error\(\)](#) and [critical\(\)](#).

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	<u>print()</u>
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	<u>logging.info()</u> (or <u>logging.debug()</u> for very detailed output for diagnostic purposes)

Task you want to perform	The best tool for the task
Issue a warning regarding a particular runtime event	warnings.warn() in library code if the issue is avoidable and the client application should be modified to eliminate the warning logging.warning() if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	logging.error() , logging.exception() or logging.critical() as appropriate for the specific error and application domain

Level or severity of the events

Level	When it's used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

How logging works

- The logging library takes a modular approach and offers several categories of components: **loggers, handlers, filters, and formatters**.
- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

Example

```
import logging
```

```
logging.warning('Watch out!') # will print a message to the console
```

```
logging.info('I told you so') # will not print anything
```

Output:

WARNING:root:Watch out!

- By default, the logging system is configured to output messages with a severity level of WARNING and above (i.e., WARNING, ERROR, and CRITICAL).
- The message 'Watch out!' will be printed to the console because it is a warning.
- the default logging level is WARNING

Logging to a file

```
import logging
logging.basicConfig(filename='example.log', encoding='utf-8', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
logging.error('And non-ASCII stuff, too, like Øresund and Malmö')
```

```
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
ERROR:root:And non-ASCII stuff, too, like Øresund and Malmö
```


- **Filename='example.log'**: Specifies that the log messages should be written to a file named example.log.
- **encoding='utf-8'**: Ensures that the log file uses UTF-8 encoding, which is necessary for properly handling non-ASCII characters like Øresund and Malmö.
- **level=logging.DEBUG**: Sets the logging level to DEBUG, which means that all messages of levels DEBUG, INFO, WARNING, ERROR, and CRITICAL will be logged.

Setting Up Basic Logging

```
import logging
```

```
# Basic configuration
```

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

```
# Logging messages
```

```
logging.debug('This is a debug message')
```

```
logging.info('This is an info message')
```

```
logging.warning('This is a warning message')
```

```
logging.error('This is an error message')
```

```
logging.critical('This is a critical message')
```



```
WARNING:root:This is a warning message  
ERROR:root:This is an error message  
CRITICAL:root:This is a critical message
```

Setting Up Basic Logging

- `logging module`, which is part of Python's standard library and provides a flexible framework for emitting log messages.
- `level=logging.DEBUG`: This sets the minimum level of messages that you want to log. In this case, all messages with a level of DEBUG or higher (INFO, WARNING, ERROR, CRITICAL) will be logged.
- `format='% (asctime)s - %(levelname)s - %(message)s'`: This sets the format of the log messages. The format string includes:
 - `%(asctime)s`: The time when the log record was created.
 - `%(levelname)s`: The severity level of the log message.
 - `%(message)s`: The actual log message.



0s

```
[8] import logging
```

```
logging.basicConfig(filename='example.log', encoding='utf-8', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
logging.error('And non-ASCII stuff, too, like Øresund and Malmö')
```



```
WARNING:root:And this, too
```

```
ERROR:root:And non-ASCII stuff, too, like Øresund and Malmö
```

```
import logging
```

Remove all handlers associated with the root logger object

```
for handler in logging.root.handlers[:]:  
    logging.root.removeHandler(handler)
```

Reconfigure logging

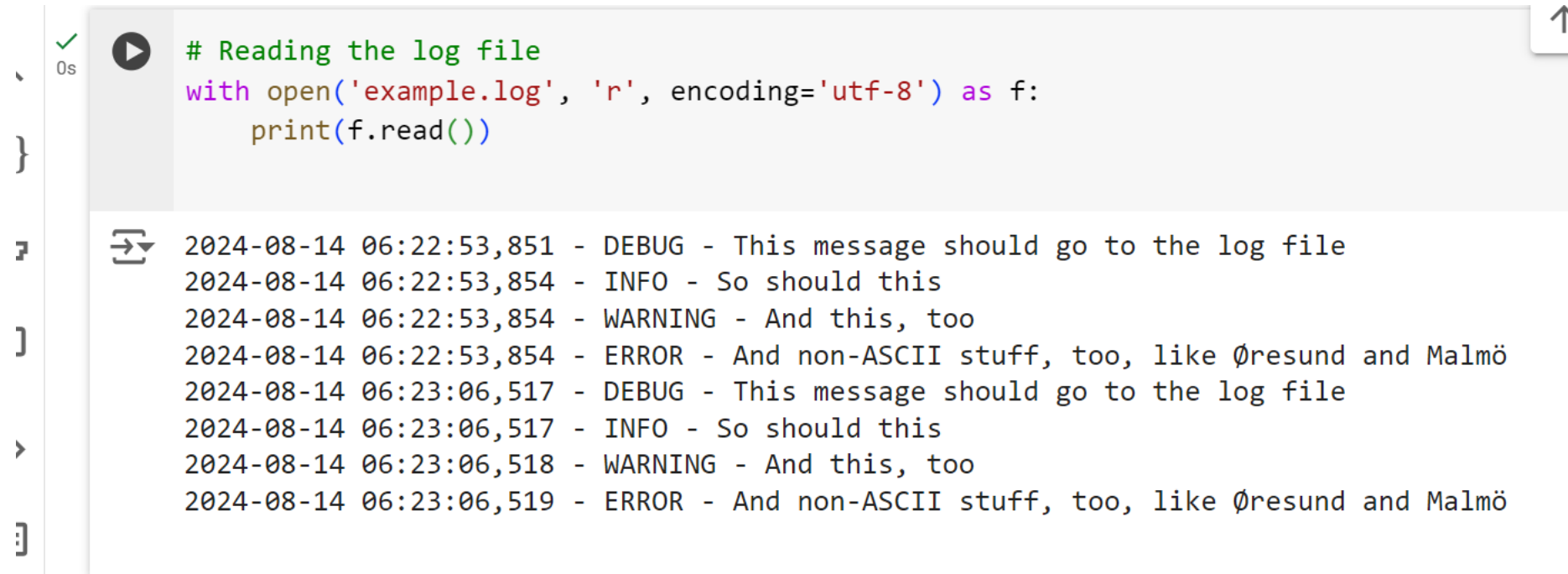
```
logging.basicConfig(filename='example.log', encoding='utf-8', level=logging.DEBUG,  
                    format='%(asctime)s - %(levelname)s - %(message)s')
```

Logging messages

```
logging.debug('This message should go to the log file')  
logging.info('So should this')  
logging.warning('And this, too')  
logging.error('And non-ASCII stuff, too, like Øresund and Malmö')
```

Reading the log file

```
with open('example.log', 'r', encoding='utf-8') as f:  
    print(f.read())
```



The screenshot shows a code editor with a light gray background. On the left, there is a vertical sidebar with a green checkmark and '0s' at the top, and a series of brackets (}',], >,]) below it. The main area contains a code block with a play button icon on the left. The code is as follows:

```
# Reading the log file  
with open('example.log', 'r', encoding='utf-8') as f:  
    print(f.read())
```

Below the code block, there is a terminal output area with a double arrow icon on the left. The output consists of eight lines of log messages:

```
2024-08-14 06:22:53,851 - DEBUG - This message should go to the log file  
2024-08-14 06:22:53,854 - INFO - So should this  
2024-08-14 06:22:53,854 - WARNING - And this, too  
2024-08-14 06:22:53,854 - ERROR - And non-ASCII stuff, too, like Øresund and Malmö  
2024-08-14 06:23:06,517 - DEBUG - This message should go to the log file  
2024-08-14 06:23:06,517 - INFO - So should this  
2024-08-14 06:23:06,518 - WARNING - And this, too  
2024-08-14 06:23:06,519 - ERROR - And non-ASCII stuff, too, like Øresund and Malmö
```

Example

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

if __name__ == '__main__':
    main()
```

If you run *myapp.py*, you should see this in *myapp.log*:

```
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

```
INFO:root:Started
INFO:root:Doing something
INFO:root:Finished
```

- The logging module is imported to enable logging.
- The mylib module is imported to demonstrate logging across modules.

logging.basicConfig(filename='myapp.log', level=logging.INFO):Configures logging to output messages to a file named myapp.log.

Sets the logging level to INFO, so all messages of level INFO and higher will be logged.

logging.info('Started'):Logs an informational message indicating that the program has started.

mylib.do_something():Calls a function from the mylib module that logs a message.

logging.info('Finished'):Logs another informational message indicating that the program has finished.

- Ensures that main() is only called if myapp.py is run as the main program.
- logging.info('Doing something'):**Logs an informational message when the do_something function is called.

Changing the format of displayed messages

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

```
DEBUG:This message should appear on the console
INFO:So should this
WARNING:And this, too
```

Date and time

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s')
logging.warning('is when this event was logged.')
```

which should print something like this:

```
2010-12-12 11:41:42,612 is when this event was logged.
```

Date and time specific format

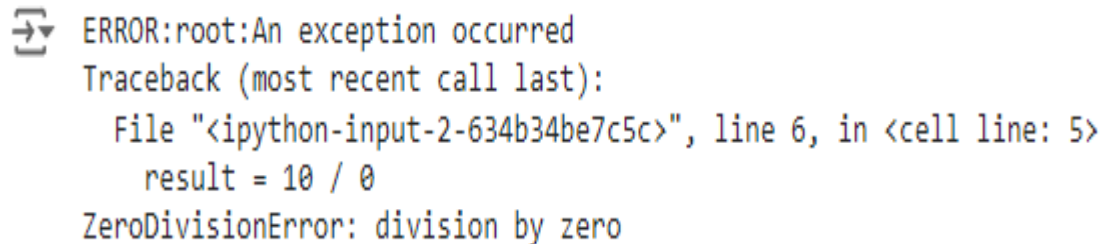
```
import logging
logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')
logging.warning('is when this event was logged.')
```

which would display something like this:

```
12/12/2010 11:46:36 AM is when this event was logged.
```

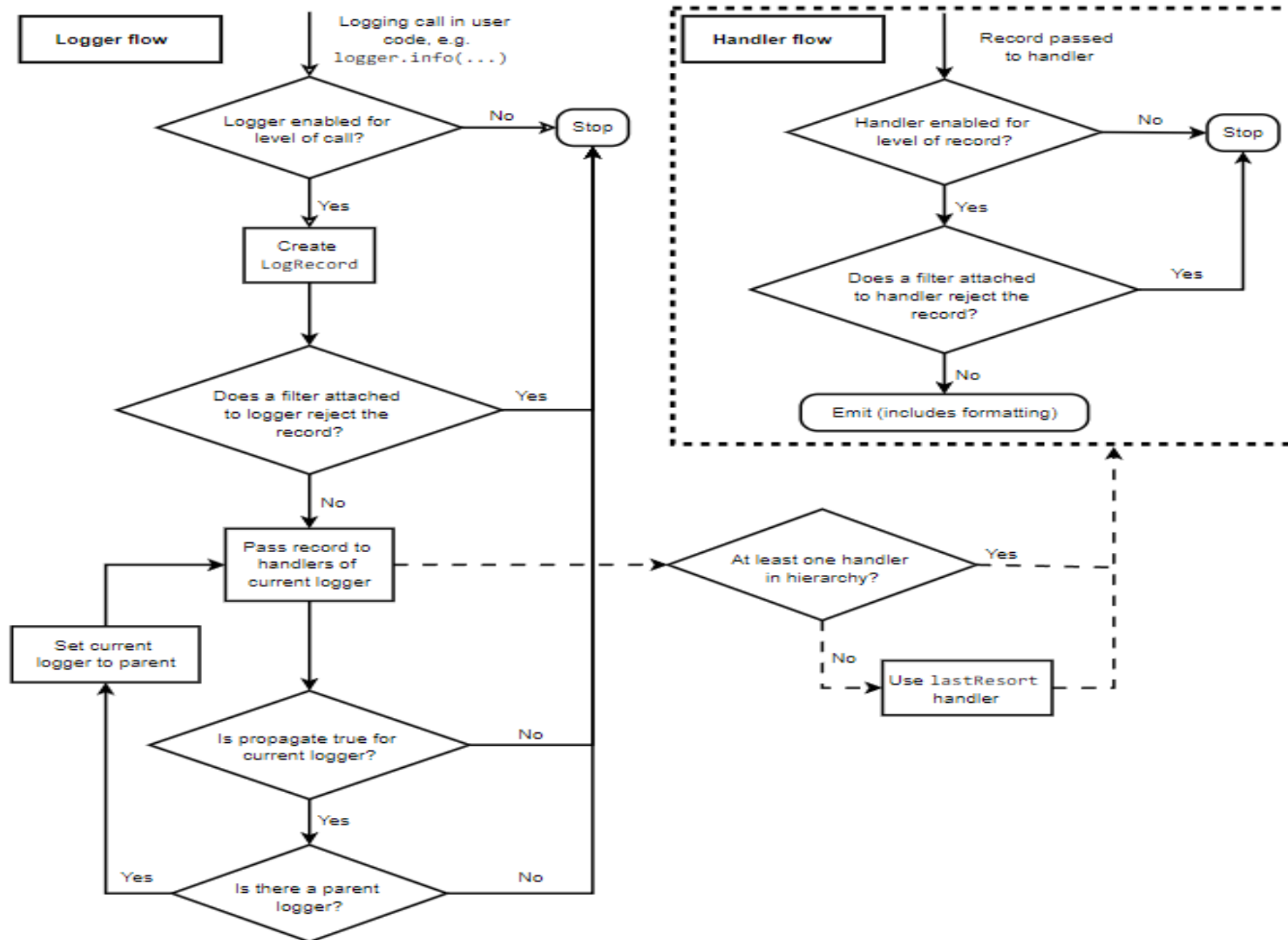
Write a script that contains a try-except block to catch an exception and log it using the exception() method.

```
import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s -
%(message)s')
try:
    result = 10 / 0
except ZeroDivisionError:
    logging.exception('An exception occurred')
```



↗ ERROR:root:An exception occurred
Traceback (most recent call last):
 File "<ipython-input-2-634b34be7c5c>", line 6, in <cell line: 5>
 result = 10 / 0
ZeroDivisionError: division by zero

Logging Flow



Example of Logging Flow

```
import logging
```

1. Create a logger

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

```
logger.setLevel(logging.DEBUG)
```

2. Create handlers

```
console_handler = logging.StreamHandler()
```

```
file_handler = logging.FileHandler('example.log')
```

3. Set levels for handlers

```
console_handler.setLevel(logging.DEBUG)
```

```
file_handler.setLevel(logging.ERROR)
```

4. Create formatters and set them for handlers

```
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
```

```
console_handler.setFormatter(formatter)
```

```
file_handler.setFormatter(formatter)
```

5. Add handlers to the logger

```
logger.addHandler(console_handler)
```

```
logger.addHandler(file_handler)
```

6. Logging messages

```
logger.debug('This is a debug message')
```

```
logger.error('This is an error message')
```



```
2024-08-21 06:02:17,658 - example_logger - DEBUG - This is a debug message
DEBUG:example_logger:This is a debug message
2024-08-21 06:02:17,663 - example_logger - ERROR - This is an error message
ERROR:example_logger:This is an error message
```

Example of Logging Flow Contd..

1.Logger Setup:

- A logger named `example_logger` is created with the `DEBUG` level.

2.Handlers:

- Two handlers are created: one for the console and one for the file.
- The console handler will process messages with level `DEBUG` or higher, while the file handler will process messages with level `ERROR` or higher.

3.Formatters:

- Both handlers use the same formatter that includes timestamp, logger name, level, and message.

4.Message Processing:

- When `logger.debug()` is called, the message is processed by the console handler because its level is `DEBUG`.
- The message is not processed by the file handler because its level is `ERROR`.

ConsoleHandler:

Purpose: Used to output log messages to the console or standard output (typically the terminal or command prompt).

Output Destination: Directly displays log messages in the terminal or command prompt where the script is executed.

```
import logging  
console_handler = logging.StreamHandler()
```

FileHandler:

Purpose: Used to output log messages to a file.

Output Destination: Writes log messages to a file on the filesystem.

```
import logging  
file_handler = logging.FileHandler('app.log')
```


Loggers

- **Logger** objects have a threefold job.
- First, they expose several methods to application code so that applications can log messages at runtime.
- Second, logger objects determine which log messages to act upon based upon severity or filter objects.
- Third, logger objects pass along relevant log messages to all interested log handlers.

logger objects fall into two categories

- configuration
- message sending.

Configuration

- [Logger.setLevel\(\)](#) specifies the lowest-severity log message a logger will handle.
- Where debug is the lowest built-in severity level and critical is the highest built-in severity.

Configuration

- [Logger.addHandler\(\)](#) and [Logger.removeHandler\(\)](#) add and remove handler objects from the logger object.
- [Logger.addFilter\(\)](#) and [Logger.removeFilter\(\)](#) add and remove filter objects from the logger object.

Methods create log messages

- [Logger.debug\(\)](#),
- [Logger.info\(\)](#),
- [Logger.warning\(\)](#),
- [Logger.error\(\)](#),
- [Logger.critical\(\)](#) .

Log Message

- [Logger.exception\(\)](#) creates a log message similar to [Logger.error\(\)](#).
- The difference is that [Logger.exception\(\)](#) dumps a stack trace along with it. Call this method only from an exception handler.

Creating custom log level

- [Logger.log\(\)](#) takes a log level as an explicit argument.

Logging Levels

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

These are primarily of interest if you want to define your own levels and need them to have specific values relative to the predefined levels.

If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

it is possibly *a very bad idea to define custom levels if you are developing a library*. That's because if multiple library authors all define their own custom levels, there is a chance that the logging output from such multiple libraries used together will be difficult for the using developer to control and/or interpret, because a given numeric value might mean different things for different libraries.

Handlers

- **Handler** objects are responsible for dispatching the appropriate log messages (based on the log messages' severity) to the handler's specified destination.
- As an example scenario, an application may want to send all log messages to a **log file**, all log messages of error or higher to **stdout**, and all messages of critical to an **email address**. This scenario requires **three individual handlers** where each handler is responsible for sending messages of a specific severity to a specific location.

Task to do

- You are developing a command-line task management system for a small team of users. The system should allow users to create tasks, assign them, update statuses, and track deadlines. Instead of using a database, you decide to store task data in text files.
- User Management:
- Implement a user registration system where users can sign up and log in. Store user data in a file, including usernames and hashed passwords.


```
import hashlib
import logging

# Configure logging
logging.basicConfig(filename='app.log', level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(message)s')
```

```
def register_user(username, password):  
    try:  
        hashed_password = hashlib.sha256(password.encode()).hexdigest()  
        with open('users.txt', 'a') as users_file:  
            users_file.write(f"{username}:{hashed_password}\n")  
        logging.info("User registered: %s", username)  
    except Exception as e:  
        logging.error("Error while registering user: %s", e)
```

```
def authenticate_user(username, password):
    try:
        hashed_password = hashlib.sha256(password.encode()).hexdigest()
        with open('users.txt', 'r') as users_file:
            for line in users_file:
                stored_username, stored_password = line.strip().split(':')
                if username == stored_username and hashed_password == stored_password:
                    return True
            return False
    except Exception as e:
        logging.error("Error while authenticating user: %s", e)
        return False
```

```
if __name__ == "__main__":  
    try:  
        while True:  
            print("1. Register\n2. Login\n3. Quit")  
            choice = input("Enter your choice: ")  
  
            if choice == '1':  
                username = input("Enter username: ")  
                password = input("Enter password: ")  
                register_user(username, password)  
                print("User registered successfully!")  
  
            elif choice == '2':  
                username = input("Enter username: ")  
                password = input("Enter password: ")  
                if authenticate_user(username, password):  
                    print("Login successful!")
```

```
elif choice == '2':  
    username = input("Enter username: ")  
    password = input("Enter password: ")  
    if authenticate_user(username, password):  
        print("Login successful!")  
  
        # Other task management code ...  
  
    else:  
        print("Login failed. Invalid credentials.")  
  
elif choice == '3':  
    print("Exiting.")  
    break  
  
else:  
    print("Invalid choice. Please choose again.")  
  
except Exception as e:  
    logging.error("Unhandled error: %s", e)
```