

PYTHON LOGGING

Logging the Exceptions:

Python Logging

Logging is the process of storing information about the execution flow and exceptions of a program into a file. It is highly recommended to log this information to help with **debugging, monitoring, and analyzing application behavior**.

Advantages of Logging

1. Logs help **debug programs** by providing detailed information about application execution.
2. Logs can be used to **analyze statistics**, such as the number of requests or errors per day.

logging levels:

Logging Levels

Python's logging module categorizes messages into six severity levels:

Level	Numeric Value	Description
CRITICAL	50	Very serious problems that require immediate attention
ERROR	40	Serious errors that affect program execution
WARNING	30	Warning messages indicating potential issues
INFO	20	General information about program execution
DEBUG	10	Detailed debugging information
NOTSET	0	Level not set; messages inherit level from parent logger

By default, Python displays messages with WARNING and higher levels.

Purpose of Logging

1. Unlike simple print statements, logging provides a structured and configurable way to record messages.
2. Logs can be stored in files or external systems, making it suitable for production environments.
3. Proper logging helps in tracking errors, monitoring application health, and maintaining audit trails.

How to implement logging:

To perform logging, first we required to create a file to store messages and we have to specify which level messages we have to store.

We can do this by using `basicConfig()` function of logging module.

```
logging.basicConfig(filename='log.txt',level=logging.WARNING)
```

The above line will create a file `log.txt` and we can store either `WARNING` level or higher level messages to that file.

After creating log file, we can write messages to that file by using the following methods.

```
logging.debug(message)
```

```
logging.info(message)
```

```
logging.warning(message)
```

```
logging.error(message)
```

```
logging.critical(message)
```

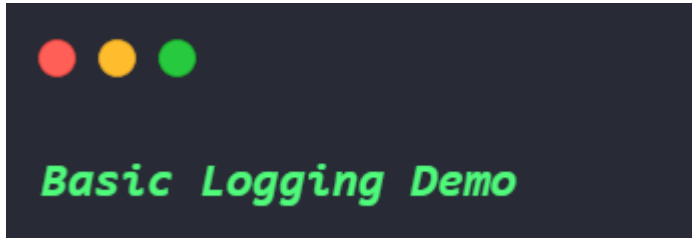
Basic Logging (WARNING and above)

```
import logging

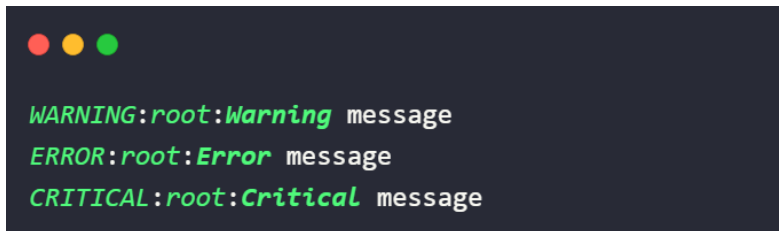
logging.basicConfig(filename='log1.txt', level=logging.WARNING)

print("Basic Logging Demo")
logging.debug("Debug message")
logging.info("Info message")
logging.warning("Warning message")
logging.error("Error message")
logging.critical("Critical message")
```

Out put::



log1.txt Contents:



Note:

In the above program only WARNING and higher level messages will be written to log file. If we set level as DEBUG then all messages will be written to log file.

Logging INFO and above with timestamps::

```
import logging

logging.basicConfig(
    filename='log2.txt',
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

logging.info("Information message")
logging.warning("Warning message")
logging.error("Error message")
```

Console Output: *(none, only log file)*

log2.txt Contents:

```
2025-09-29 15:30:12,345 - INFO - Information message
2025-09-29 15:30:12,346 - WARNING - Warning message
2025-09-29 15:30:12,347 - ERROR - Error message
```


Logging to Console instead of File

```
import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug("Debug message")
logging.info("Info message")
logging.warning("Warning message")
logging.error("Error message")
logging.critical("Critical message")
```

Console Output:



```
DEBUG:root:Debug message
INFO:root:Info message
WARNING:root:Warning message
ERROR:root:Error message
CRITICAL:root:Critical message
```

Conclusion ::

1. Python's logging module provides a flexible way to record program execution, errors, and important events.
2. It is more powerful and configurable than print statements, allowing messages to be stored in files, displayed on the console, or both.
3. Logging helps in debugging, monitoring, and analyzing applications, especially in production environments.
4. By using different logging levels (DEBUG, INFO, WARNING, ERROR, CRITICAL), developers can filter messages according to their importance.
5. Proper logging is a best practice for building robust, maintainable, and professional Python applications.