1.Setting up Black Python code formatter on VS Code

EH

Owned by Elijah Hoole, created with a template

Last updated: Sept 22, 20232 min read5 people viewed

This how-to guide provides instructions for setting up the Black formatter for Python. Everyone working on General-OCR-Framework **must** use this formatter for code uniformity across the code base.

Install Black on your Python environment

To install Black formatter:

```
pip install black
```

To format an existing Python script or a directory containing multiple python scripts:

```
black {source_file_or_directory}...
```

You can also use black as a Python package. To format an existing Python script or directory containing multiple python scripts:

```
python -m black {source_file_or_directory}...
```

Replace `{source_file_or_directory}` with the actual name of your source file or directory.

Set up black as the default Python formatter on VS Code

1. Install the Black VS Code extension
2. Install Black Formatter Extension
3. Go to your VS Code settings, by clicking Ctrl + , (Ctrl + Comma) OR clicking the Manage button on the left bottom corner and going to > Settings.
4. Search for ‘formatter’
5. Set Default Formatter to Black Formatter
6. Set your Default Formatter to be Black Formatter as shown above. Make sure you are editing ‘User’ level defaults (that you are working on ‘User’ tab in the line immediately below the search box).
7. Scroll down and enable ‘Format On Save’ as shown below
8. Enable Format On Save
9. Open a new python script

Type:

```
string_to_format = 'hello word'
```

Press:

```
Ctrl + S
```

Check if code has changed to:

```
string_to_format = "hello word"
```

1.Using type annotation or type hinting

EH

Owned by Elijah Hoole

Last updated: Sept 20, 20232 min read5 people viewed

What is type annotation?

Unlike statically typed languages like C++ or Java, Python does not force us to declare the type of the values we manipulate using our programme. Type is inferred at run time instead. This leads to the familiar `TypeError` we encounter regularly.

From Python 3.5 onward, the language supports type annotation or hinting. With type hints, other developers get to know what type each of the arguments to a method are just by reading the function signature.

For example, consider the interface of this function that I wrote for UOB OCR project:

```
def create_df_from_cell_boxes(table, row_boxes, col_boxes, cell_boxes,
expected_cols):

    . . .

    return df
```

The first argument is `table.` Although I know that `table` refers to the cropped `PIL.Image.Image` of the table region from a page, another developer will have no idea. Further, because of common usage most of us know that `df` is a pandas.core.frame.DataFrame. But, if I had used a generic variable name like output_1, it would be hard for other developers to know what the method returns. With type hinting, we can remove this burden and make the code much more readable and easier to understand for fellow developers.

With type hints, the same function signature would look like this:

```
from PIL import Image
import pandas as pd
from typing import List, Tuple


def create_df_from_cell_boxes(
   table: Image.Image,
   row_boxes: List[List[float]],
   col_boxes: List[List[float]],
   cell_boxes: List[List[float]],
   expected_cols: int,
) -> pd.DataFrame:
   ...

   return df
```

Now we know that table is a `PIL.Image.Image` and that the method returns a `pandas.core.frame.DataFrame`. We also know that that `row_boxes`, `col_boxes`, and `cell_boxes` are list of lists where each sub list contains float numbers. The interface itself now provides so much more information and makes the code so much easier to understand. Further, when our code is type annotated, we can use a type checker like `mypy` to catch potential type errors before run time.

If you want to include default values and return more than one value:

```
def create_df_from_cell_boxes(

   table: Image.Image,

   row_boxes: List[List[float]],

   col_boxes: List[List[float]],

   cell_boxes: List[List[float]],

   expected_cols: int = 5,

) -> Tuple[pd.DataFrame, pd.DataFrame]:

   ...

   return df, df_score
```

💻

2.Practical ways to keep your Python code simple and readable

EH

Owned by Elijah Hoole

Last updated: Sept 20, 20232 min read4 people viewed

This article suggests some practical ways to keep your Python code simple. Code is written once but read and edited multiple times, often by developers other than the one who originally wrote the code. So keeping code simple and readable is of utmost importance.

This guide borrows heavily from Nuwan Senaratna’s blog.

To be able to test your code for the following rules (of thumb), first install `flake8` in your Python environment.

```
pip install flake8-functions
```

1. Maximum function length = 30

If your function is more than 30 lines long, you need to break up your function and wrap some steps into smaller functions.

You can check Python function lengths by:

```
python -m flake8 --max-function-length 30 your_source_file.py
```

2. Maximum number of parameters in a function = 3

No function should have more than three parameters. If you have more than three parameters in a function, either the function is too complex or your parameters need to be consolidated into a class.

You can check if your functions have more than three parameters by:

```
python -m flake8 --max-parameters-amount 3 your_source_file.py
```

3. Maximum function cyclomatic complexity = 6

Cyclomatic complexity roughly estimates the number of unique paths a piece of code could execute in. For instance, each `if else` block represents a unique path. Try to restrict each function's cyclomatic complexity to 6.

```
python -m flake8 --max-complexity 6 your_source_file.py
```

4. Maximum number of return values of a function = 3

If a function is returning more than 3 values, the function is probably doing too many things and could benefit from refactoring. If you are sure that you need to return more than three values and all of them are linked to a single purpose, you can encapsulate the return values in a more complex data structure such as a dictionary. This is not something we can automatically check, so this is something to keep in mind. (I’ll see if I can write a flake8 extension)

5. Number of things a function does = 1

A function should never do more than one thing. If it feels like it does more than one thing, then it should be split into multiple functions. It is not something we can test automatically but this is something you need to keep in mind when you code. But if you follow Rules 1, 2, 3, and 4 you will mostly get Rule 5 for free.

💻

3.Python naming conventions

EH

Owned by Elijah Hoole, created with a template

Last updated: Nov 21, 20232 min read3 people viewed

Guidelines for naming things in your Python code.

📘

Common sense principles for naming things well

Choosing meaningful and expressive names for variables, functions, classes, and other elements in programming is crucial for writing clear, maintainable, and understandable code. Here are some general principles for choosing names:

1. **Descriptive and intuitive**: Names should accurately reflect the purpose or nature of the entity they represent. For instance, naming a variable `json_file` when it is actually a dictionary loaded from a JSON file can be misleading. A more descriptive name would be `data_dict` or `loaded_json`, indicating that the content has already been processed into a dictionary.
2. **Avoid misleading names**: Names should not give a false impression about the type, purpose, or behavior of the entity. Misleading names can lead to confusion and errors. For example, naming a string variable `list_of_names` is misleading if it's not actually a list data type.

3. **Specificity over generality**: Specific names are usually better than generic ones. For example, `user_age` is more descriptive than just `age`, and `calculate_net_income` is more informative than `calculate`.
4. **Avoid abbreviations and acronyms**: Unless the abbreviation is more widely known than the full term (like `URL`), it's generally better to avoid them in names. Abbreviations can make the code less accessible, especially to those who are not familiar with them.
5. **Length of names**: Names should be as long as necessary to be descriptive, but as short as possible to maintain readability. For example, `num` is preferable to `n`, but `number_of_students` is excessively long and could be `student_count`.
6. **Use pronounceable and readable names**: Names should be easily pronounceable and readable. This makes communication about the code easier and enhances the readability of the code.
7. **Context matters**: The context in which a name is used can often allow for shorter names. For example, in a function that calculates an area, `width` and `height` are preferable to `rectangle_width` and `rectangle_weight`.
8. **Consistency**: Be consistent with naming conventions throughout your code. If you start naming similar entities in a certain way, stick to that pattern. Consistency helps in understanding the structure and logic of the code.
9. **Avoid 'magic numbers'**: Instead of using hard-coded numbers, use named constants which can give context. For example, use `MAX_RETRY_LIMIT` instead of just `5` in your code.
10. **Avoid redundancy**: Don’t include unnecessary context. For example, if you have a class named `Car`, you don’t need to name a method `get_car_speed`; `get_speed` would be sufficient and clearer.

These principles promote code readability and maintainability. They make it easier for others (and yourself) to understand and work with your code, especially in a collaborative or long-term project.

📘

Common naming styles

PEP-8 distinguishes the following naming styles.

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords, or CamelCase)
- `mixedCase` (differs from CapitalizedWords by initial lowercase character!)
- `Capitalized_Words_With_Underscores` (ugly!)

When writing Python code, avoid `mixedCase` and `Capitalized_Words_With_Underscores`.

📘

Which naming style to use where

The Thing You Are Naming	What Style to Use	Example
Single character variables	b (single lowercase letter)	`x`
Constants	B (single uppercase letter)	`N`
Variables	lowercase	`name`
Functions, method names, and instance variables	lower_case_with_underscores	`calculate_age`, `current_balance`

Global/Class Constants	UPPERCASE	`PI`
Global/Class Constants	UPPER_CASE_WITH_UNDERSCORES	`DEFAULT_COLOR`
Classes and Exceptions	CapitalizedWords (or CamelCase)	`BankAccount` or `TypeError`
Sub-directories under a package	lowercase or lower_case_with_underscores	`data` or `data_directory`

💻

5. Guard patterns

Purpose:

- To validate function inputs and handle edge cases early, ensuring code clarity and preventing errors.
- To exit a function quickly if preconditions aren't met, avoiding unnecessary conditional blocks.

Structure:

- Placed at the beginning of the function, typically within an `if` statement.
- Check for conditions that would make the function's main logic invalid.
- If a condition fails, return a value or raise an exception to halt execution.

Example:

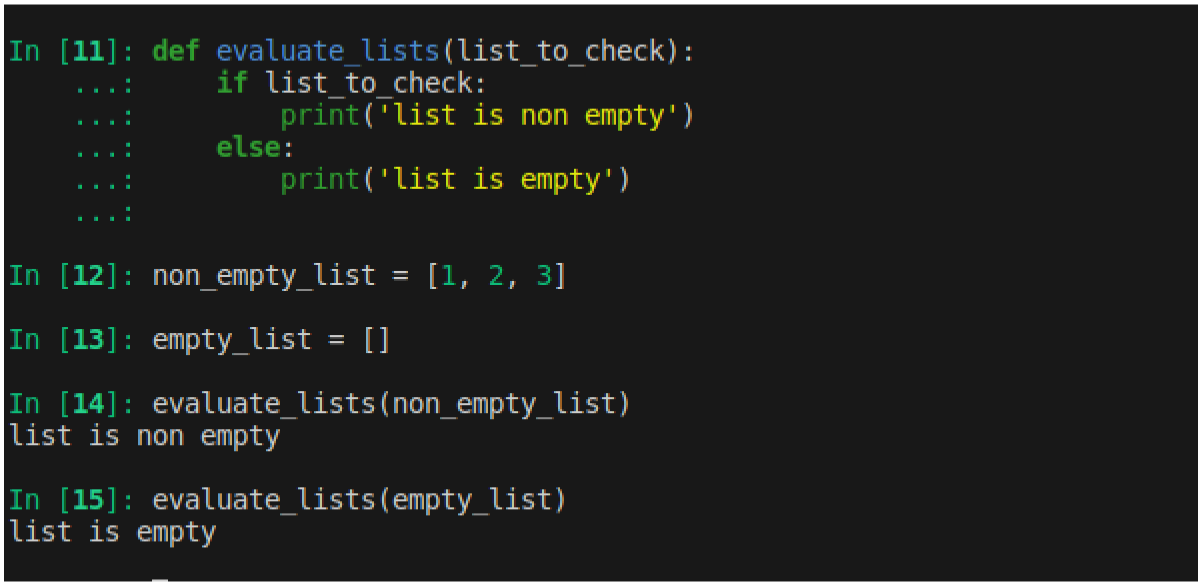
```
def calculate_average(numbers):
    if not numbers:  # Guard clause to check for empty list
        Return None
    total = sum(numbers)
    return total / len(numbers)
```

Benefits:

- Enhanced Readability: Flattens conditional structure and highlights exceptional cases.
- Early Error Handling: Catches potential issues before they propagate deeper into the code.
- Improved Maintainability: Clearer logic makes code easier to understand and modify.

Also when checking for empty lists always write:

```
if some_list:  # Checks for truthiness of the list
    # List is not empty, proceed with logic
```



```
In [11]: def evaluate_lists(list_to_check):
    ...:     if list_to_check:
    ...:         print('list is non empty')
    ...:     else:
    ...:         print('list is empty')
    ...:

In [12]: non_empty_list = [1, 2, 3]

In [13]: empty_list = []

In [14]: evaluate_lists(non_empty_list)
list is non empty

In [15]: evaluate_lists(empty_list)
list is empty
```

Avoid:

```
if len(some_list) == 0:  # Less concise and less Pythonic
```

Remember:

- Use guard clauses consistently for preconditions and edge cases.
- Prioritize truthiness checks for empty collections for clarity and efficiency.
- Embrace Python's language features to write clean and concise code.

💻

6. Using f-strings for enhanced readability and conciseness

Instead of using the traditional `print` method with string concatenation, always employ f-strings for a more concise and readable approach.

Rationale:

- Improved Readability: F-strings allow embedding variables directly within the string, reducing the need for concatenation and enhancing clarity.
- Reduced Errors: No need to manage separate string and variable lists, mitigating the risk of typographical errors and mismatching indexes.
- Concise Syntax: Shorter and cleaner compared to traditional methods, especially for complex formatting.

Examples:

Traditional approach:

```
page_num = 10
table_num = 5

print('page number is', page_num, 'table is', table_num)
```

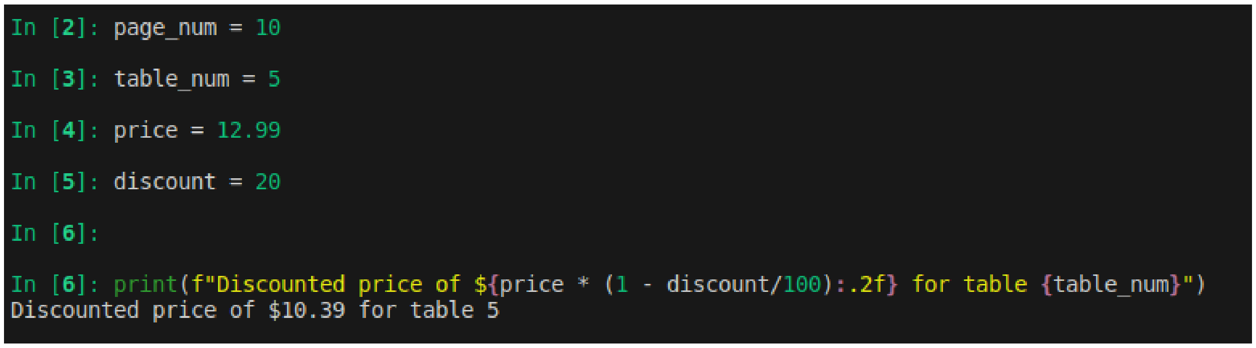
Using f-strings:

```
print(f"Currently processing page {page_num} with table
{table_num}.")
```

You can do formatting inside f-strings:

Python

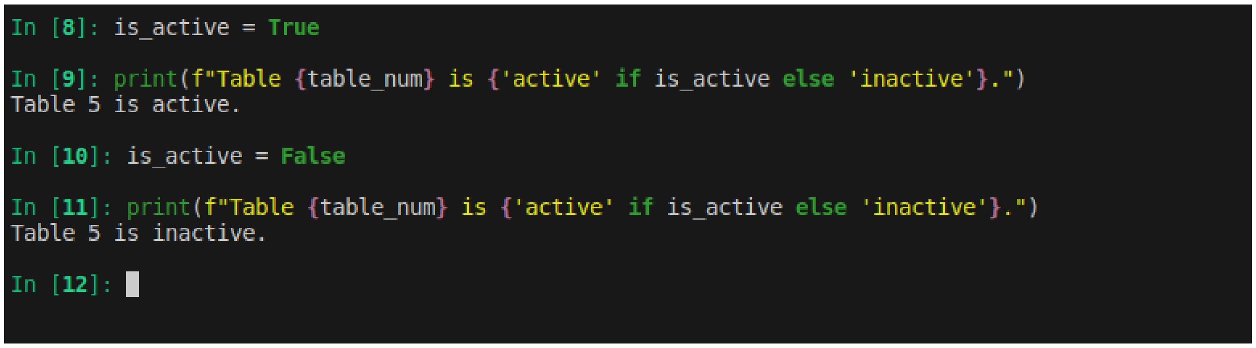
```
price = 12.99
discount = 20

print(f"Discounted price of ${price * (1 - discount / 100):.2f}
for table {table_num}.")
```



```
In [2]: page_num = 10

In [3]: table_num = 5

In [4]: price = 12.99

In [5]: discount = 20

In [6]:

In [6]: print(f"Discounted price of ${price * (1 - discount/100):.2f} for table {table_num}")
Discounted price of $10.39 for table 5
```

You can evaluate conditionals or small expressions inside f-strings

```
is_active = True

print(f"Table {table_num} is {'active' if is_active else
'inactive'}.")
```



```
In [8]: is_active = True

In [9]: print(f"Table {table_num} is {'active' if is_active else 'inactive'}.")
Table 5 is active.

In [10]: is_active = False

In [11]: print(f"Table {table_num} is {'active' if is_active else 'inactive'}.")
Table 5 is inactive.

In [12]:
```

Remember:

- Place variables within curly braces `{}` within the f-string.
- You can perform basic formatting and expressions within the braces.
- F-strings enhance code readability and maintainability, fostering cleaner and more concise scripts.

💻

7. Using logging vs. print

During development, you may quickly want to `print` something to see the output. This is completely fine so long as you delete after you are satisfied with what your code is producing.

But, often, there is stuff you want to log at runtime. For example, if your flow has ten different processes that run one after the other, you may want to log that Process A has started or that Process B has finished. For this sort of display, you should use `logging` and avoid `print`.

Why use logging?

Logging Levels:

- `logging` provides different log levels (e.g., `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`), allowing you to control the verbosity of your logs.
- This enables you to have more control over what information is displayed based on the severity of the message, making it easier to filter and analyze logs.

Configurability:

- `logging` allows you to configure different log handlers (e.g., console, file, network) and formatters to control the output format.
- This configurability makes it easier to direct logs to different destinations and adjust the logging behavior without modifying the code.

Timestamps and Context Information:

- The `logging` module automatically includes timestamps in log messages, providing valuable information about when events occurred.
- Additionally, it allows you to include contextual information (e.g., module name, function name) in the log messages, aiding in debugging and analysis.

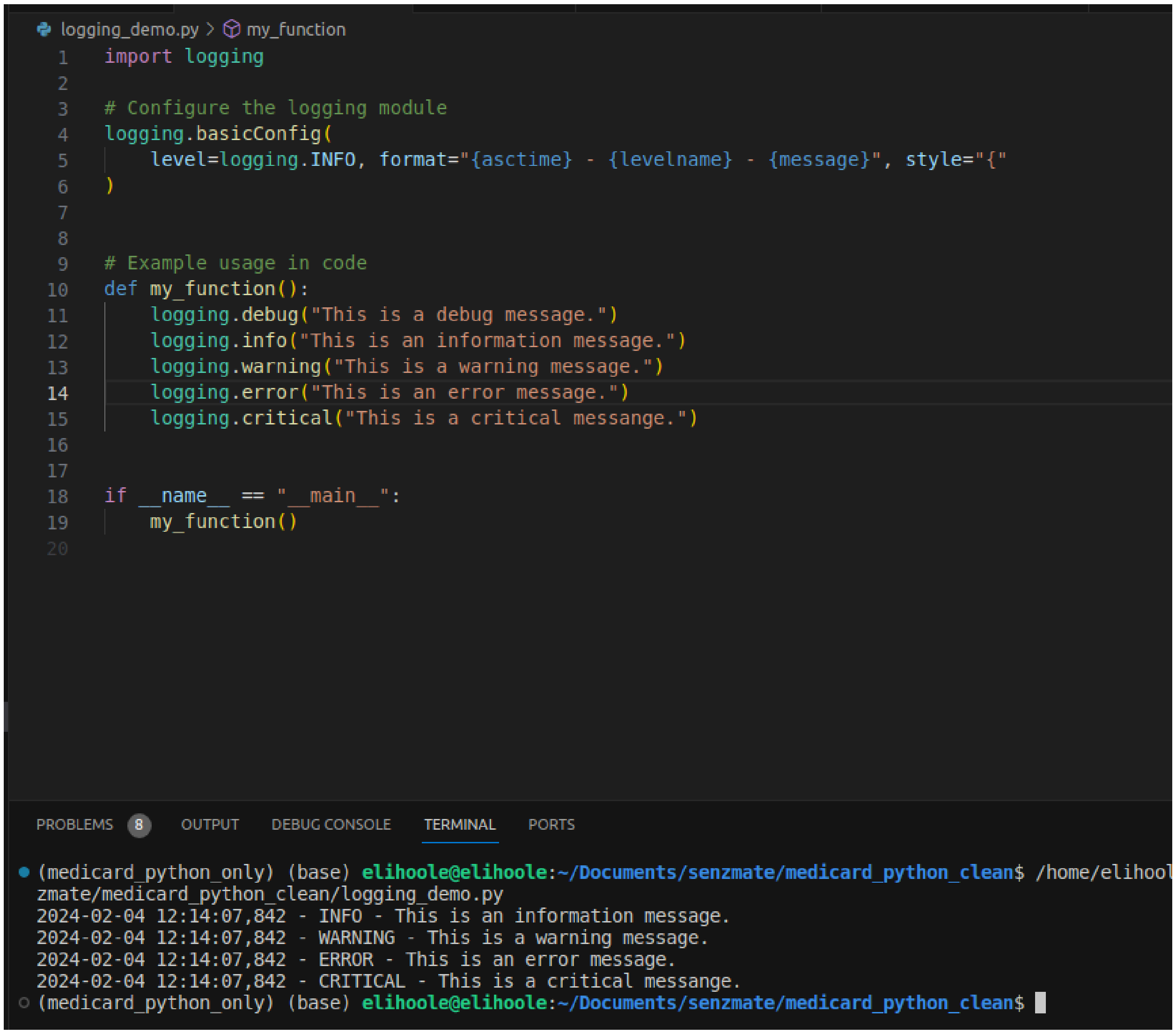
Redirection and Filtering:

- With `logging`, you can easily redirect logs to different outputs (e.g., file, database) without changing the code.
- Filtering based on log levels or other criteria can be efficiently implemented with `logging`, helping you focus on relevant information.

Consistency in Code Style:

- Adhering to a consistent style, like using `logging` for logs and `print` for short term debugging, makes the codebase more readable and maintainable.

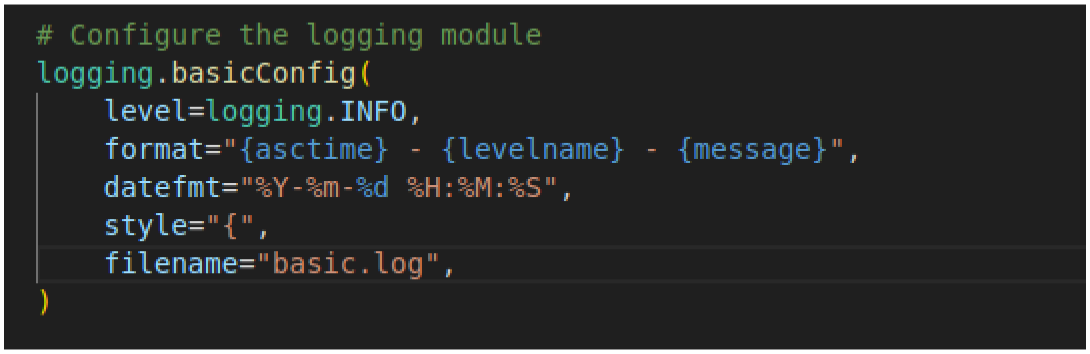
How to use logging



```python
import logging

# Configure the logging module
logging.basicConfig(
    level=logging.INFO, format="{asctime} - {levelname} - {message}", style="{"
)


# Example usage in code
def my_function():
    logging.debug("This is a debug message.")
    logging.info("This is an information message.")
    logging.warning("This is a warning message.")
    logging.error("This is an error message.")
    logging.critical("This is a critical message.")


if __name__ == "__main__":
    my_function()
```

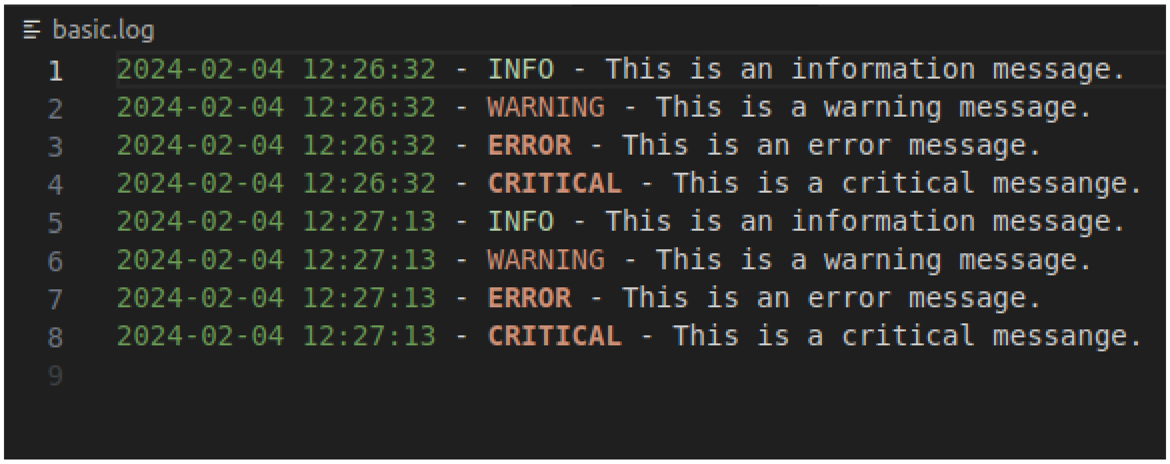
```
(medicard_python_only) (base) elihoole@elihoole:~/Documents/senzmate/medicard_python_clean$ /home/elihool
zmate/medicard_python_clean/logging_demo.py
2024-02-04 12:14:07,842 - INFO - This is an information message.
2024-02-04 12:14:07,842 - WARNING - This is a warning message.
2024-02-04 12:14:07,842 - ERROR - This is an error message.
2024-02-04 12:14:07,842 - CRITICAL - This is a critical messange.
(medicard_python_only) (base) elihoole@elihoole:~/Documents/senzmate/medicard_python_clean$
```

If you notice in the screenshot above, the `debug` message did not get displayed. This is because we have set the level to `INFO` at the time of initialization. `logging` displays messages at or above the level of severity we set during initialisation. This is great because when we are developing or in our stage environment, we may want to view a lot of `debug` messages but once we push to production, we may only be interested in higher severity levels. When you use `logging` instead of print, you can just set the severity level differently at just one place in the code base to hide or display different levels of logs instead of adding or removing ‘#’ at a dozen different lines.



```python
# Configure the logging module
logging.basicConfig(
    level=logging.INFO,
    format="{asctime} - {levelname} - {message}",
    datefmt="%Y-%m-%d %H:%M:%S",
    style="{",
    filename="basic.log",
)
```

You can customize your `logging` further. Above, I have added a specific date format and instead of displaying the messages in stdout, I am redirecting them to a log file called `basic.log`. I ran the script twice and as you can see below, there logs have been repeated twice. This is great as it enables us to go back and see what happened in past runs. There are also straightforward ways to push `logging` to databases should you need such a facility.



basic.log

```
2024-02-04 12:26:32 - INFO - This is an information message.
2024-02-04 12:26:32 - WARNING - This is a warning message.
2024-02-04 12:26:32 - ERROR - This is an error message.
2024-02-04 12:26:32 - CRITICAL - This is a critical messange.
2024-02-04 12:27:13 - INFO - This is an information message.
2024-02-04 12:27:13 - WARNING - This is a warning message.
2024-02-04 12:27:13 - ERROR - This is an error message.
2024-02-04 12:27:13 - CRITICAL - This is a critical message.
```

Keep in mind: assume logs are unsecure

Never log sensitive information such as passwords or email addresses. Always assume that logs are unsecure.