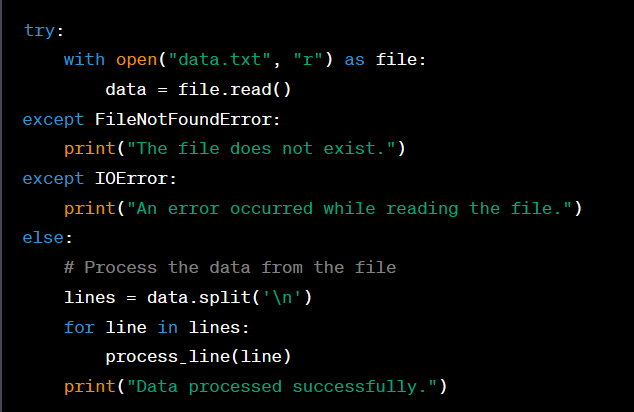
1. **What is the role of the 'else' block in a try-except statement? Provide an example scenario where it would be useful.**

**Ans:**

* *The `else` block in a `try-except` statement is used to specify a block of code that should be executed when no exceptions are raised within the preceding `try` block. In other words, the `else` block is executed when the code in the `try` block runs without any errors or exceptions.*
* *The primary role of the `else` block is to separate the error-handling logic (defined in the `except` block) from the code that should execute when no exceptions occur. This can improve code readability and make it more structured.*
* **Example :**
* *Suppose you have a program that reads data from a file and processes it. You want to handle the case where the file might not exist or is empty while also processing the data when everything is correct. Here's how you can use the `else` block***:**

****

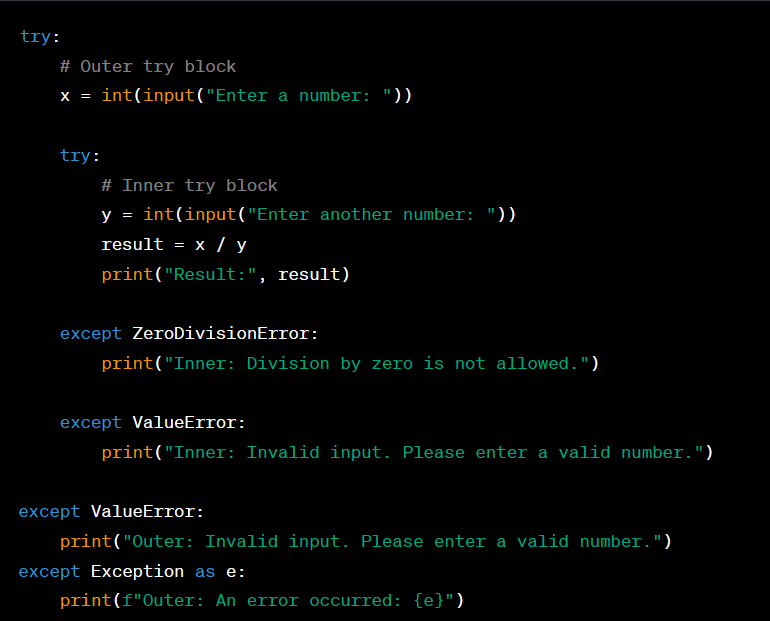
*In this example:*

* *The `try` block attempts to open and read the "data.txt" file.*
* *The `except` blocks handle specific exceptions related to the file operation, such as `FileNotFoundError` or `IOError`.*
* *If no exceptions are raised (i.e., the file exists and can be read), the code in the `else` block processes the data and provides a success message.*
* *The `else` block allows you to keep the code for handling exceptions separate from the code that executes when everything goes as expected, which can improve code organization and clarity.*

1. **Can a try-except block be nested inside another try-except block? Explain with an example.**

**Ans:**

*Yes, you can nest a `try-except` block inside another `try-except` block. This allows you to handle exceptions at different levels of granularity. The outer `try-except` block can catch broader exceptions, while the inner `try-except` block can handle more specific exceptions within the outer block. Here's an example to illustrate this:*

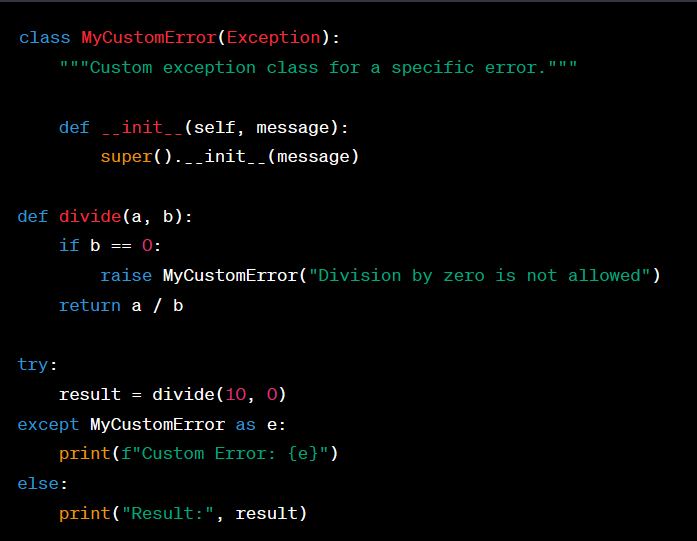
*****In this example:*

* + *The outer `try` block takes user input for the first number `x`.*
  + *Inside the outer `try` block, there is an inner `try` block that takes user input for the second number `y` and calculates the result `x / y`.*
  + *The inner `try` block has its own `except` blocks for handling exceptions specific to division by zero (`ZeroDivisionError`) and invalid input (`ValueError`).*
  + *The outer `try` block also has its `except` blocks for handling exceptions not caught by the inner `try` block. A more general `except Exception as e` block is used to catch unexpected exceptions.*

*The nested structure allows you to handle exceptions at different levels of the program. In this example, you can see how errors can be caught at both the inner and outer levels, and different error messages can be provided based on the context of the error.*

1. **How can you create a custom exception class in Python? Provide an example that demonstrates its usage.**

**Ans***: You can create a custom exception class in Python by defining a new class that inherits from the built-in `Exception` class or one of its subclasses. This allows you to create your own exception types to handle specific error conditions in your code. Here's an example that demonstrates how to create and use a custom exception class:*

****

**In this example:**

*1. We define a custom exception class named `MyCustomError` that inherits from the built-in `Exception` class. We also provide an `\_\_init\_\_` method to customize the error message.*

*2. The `divide` function checks if the divisor (`b`) is zero. If it is, it raises a `MyCustomError` with a specific error message.*

*3. In the `try` block, we call the `divide` function with arguments that would result in division by zero. This raises the `MyCustomError`.*

*4. In the `except MyCustomError as e` block, we catch the custom exception, and we can access the error message using the `e` variable. We print the custom error message.*

*5. If no exception is raised, the `else` block is executed, and we print the result.*

**4. What are some common exceptions that are built-in to Python?**

**Ans:** *Python includes a wide range of built-in exceptions to handle various error conditions that can occur during program execution. Here are some common built-in exceptions in Python:*

*1. SyntaxError: Raised when there is a syntax error in the code.*

*2. IndentationError: Raised when there is an issue with the code's indentation.*

*3. NameError: Raised when a local or global name is not found.*

*4. TypeError: Raised when an operation or function is applied to an object of inappropriate type.*

*5. ValueError: Raised when a function receives an argument of the correct type but with an inappropriate or invalid value.*

*6. KeyError: Raised when a dictionary is accessed with a key that doesn't exist.*

*7. IndexError: Raised when an index is out of range for a sequence (e.g., list, tuple).*

*8. ZeroDivisionError: Raised when division or modulo operation is performed with a divisor of zero.*

*9. FileNotFoundError: Raised when an attempt to open a file for reading or writing fails because the file does not exist.*

*11. AttributeError: Raised when an attribute reference or assignment fails because the attribute doesn't exist.*

*12. ImportError: Raised when an import statement fails to find a module or cannot import a module.*

*13. KeyError: Raised when a key is not found in a dictionary.*

*14. ValueError: Raised when an operation receives an argument of the correct type but with an inappropriate or invalid value.*

**5. What is logging in Python, and why is it important in software development?**

**Ans***: Logging in Python is the process of recording and storing information, events, and messages generated during the execution of a program. It is an essential practice in software development for the following reasons:*

1. *Debugging and Troubleshooting: When developing and maintaining software, it's crucial to have a reliable record of what the program is doing. Logging allows developers to trace the execution flow and identify issues or errors, which simplifies debugging and troubleshooting.*
2. *Error Handling: Logging is vital for recording error messages and exceptions. It provides detailed information about the error context, such as the location, variables' values, and a stack trace, making it easier to diagnose and fix problems.*
3. *Monitoring and Reporting: In production environments, applications run autonomously. Logging allows system administrators to monitor the behavior of an application in real-time or to review logs after the fact. This is essential for detecting performance issues, security breaches, or other anomalies.*
4. *Performance Analysis: Logs can include timing information, which aids in profiling and performance analysis. Developers can identify performance bottlenecks and optimize code accordingly.*

**.**

**6. Explain the purpose of log levels in Python logging and provide examples of when each log level would be appropriate.**

**Ans:** *Log levels in Python logging are used to categorize log messages based on their severity or importance. They allow developers to control which log messages are recorded and which are ignored, based on the desired level of detail for debugging, monitoring, or analysis. Python's `logging` module defines several standard log levels, each serving a specific purpose:*

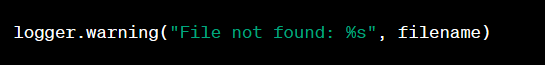
*1. \*\*DEBUG\*\*: The lowest log level, typically used for detailed debugging information. These messages are meant for developers during the development and debugging phase. For example:*

**

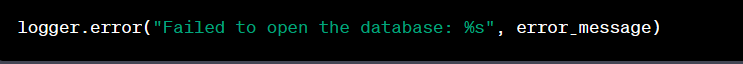
*2. \*\*INFO\*\*: Messages providing general information about the program's operation. These messages are typically used for high-level flow and state information. For example:*

**

*3. \*\*WARNING\*\*: Messages indicating that something unexpected or potentially problematic has occurred. These are not critical errors but should be reviewed. For example:*

**

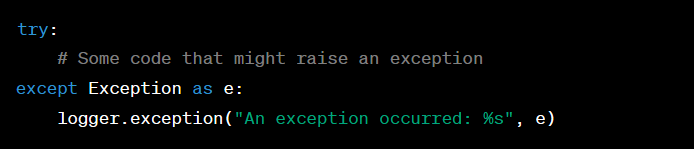
*4. \*\*ERROR\*\*: Messages indicating that a significant error or exception occurred but the program can continue. These errors are typically logged when a problem is detected, but the application can recover or continue functioning. For example:*

**

*5. \*\*CRITICAL\*\*: Messages indicating a critical error or exception that prevents the application from continuing. These are the most severe log messages. For example:*

**

*6. \*\*EXCEPTION\*\*: In some cases, custom log levels, such as "EXCEPTION," are used to capture exceptions and their stack traces explicitly. This allows you to handle exceptions with custom log level handling, separate from other log messages. For example:*

**

*The choice of log level depends on the specific needs of your application and the context in which the logging is used. Here are some examples of when each log level would be appropriate:*

* + *\*\*DEBUG\*\*: When you want to capture detailed information for debugging, such as variable values, function entry/exit points, and execution flow during development.*
  + *\*\*INFO\*\*: When you want to provide high-level information about the application's state, startup, and shutdown, or to indicate key application milestones.*
  + *\*\*WARNING\*\*: When you encounter non-critical issues or events that should be noted but don't necessarily disrupt the application's functionality.*
  + *\*\*ERROR\*\*: When you encounter errors that are significant but not catastrophic, and the application can continue functioning with some degradation.*
  + *\*\*CRITICAL\*\*: When a critical error occurs that prevents the application from continuing and requires immediate attention.*
  + *\*\*EXCEPTION\*\*: When you want to explicitly capture exceptions and their stack traces for detailed error analysis.*

*By using log levels effectively, you can control the volume of log output and tailor it to your application's specific requirements, making it easier to debug issues, monitor application behavior, and maintain software health.*

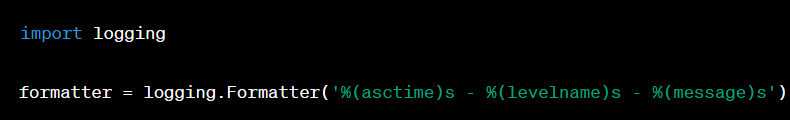
**7. What are log formatters in Python logging, and how can you customise the log message format using formatters?**

**Ans***: Log formatters in Python logging are used to control the appearance and structure of log messages. They allow you to customize how log messages are presented in the log output, including the log message content, timestamp, log level, and more. Formatters are especially useful for creating log messages that are well-organized and easy to read. Python's `logging` module provides a built-in `Formatter` class for this purpose.*

*Here's how you can create and customize log message formats using formatters:*

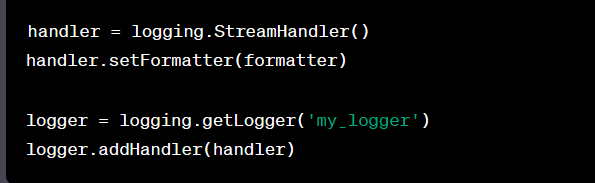
*1. Create a Formatter:*

* *You can create a `Formatter` instance by specifying the format string you want to use. The format string can include placeholders that represent various attributes of a log record, such as the timestamp, log level, message, and more. Here's an example of creating a basic formatter:*

**

*2. Attach Formatter to a Handler:*

* *Formatters are typically associated with logging handlers (e.g., `FileHandler`, `StreamHandler`) to control the format of log messages produced by that handler. You can attach the formatter to a handler like this:*

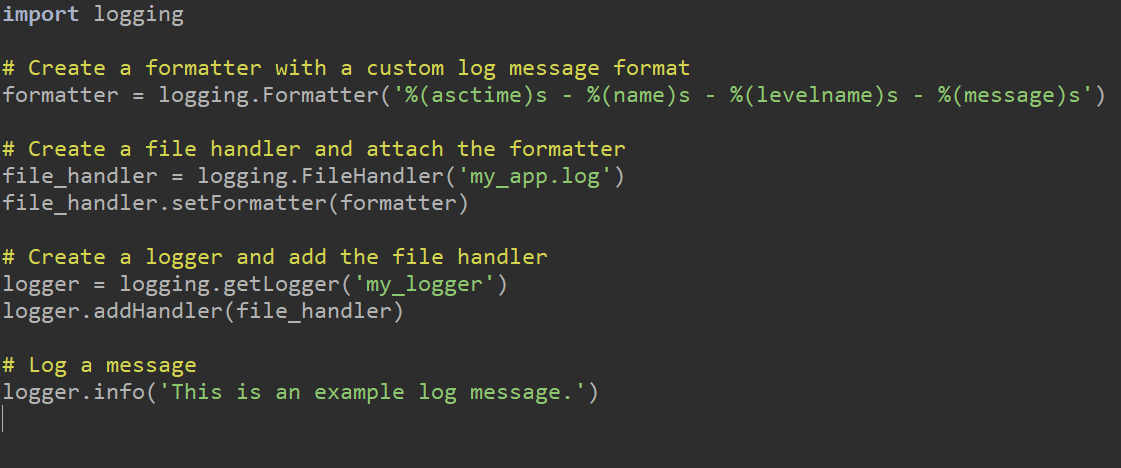
**

*3. Format String Placeholders:*

* *In the format string, you can use various placeholders enclosed in `%(` and `)s` to include attributes from log records.*
* *Common placeholders include:*
* *`asctime`: Timestamp when the log record was created.*
* *`levelname`: Log level name (e.g., 'DEBUG', 'INFO').*
* *`message`: The actual log message.*
* *`name`: The name of the logger.*
* *`module`: The name of the module where the log message originated.*
* *`lineno`: The line number in the source code where the log message originated.*
* *For example, `'%(asctime)s - %(levelname)s - %(message)s'` includes the timestamp, log level, and log message in the log format.*

*4. Advanced Customization:*

* *You can customize the format string to include other attributes or adjust the appearance of log messages. For example, you can include the name of the logger (`'%(name)s'`) or the filename and line number (`'%(module)s:%(lineno)d'`) where the log message was generated.*
* *Here's a more complete example that demonstrates creating a custom log message format with a formatter and associating it with a file handler:*

**

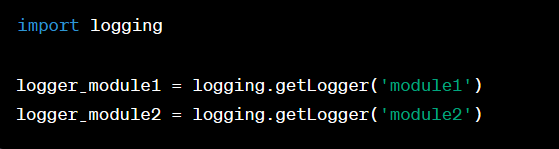
* *In this example, log messages produced by the logger will have a custom format that includes the timestamp, logger name, log level, and message. The messages will be written to a file specified by the file handler.*
* *Customizing log message formats with formatters allows you to tailor the presentation of log information to meet your application's specific needs, making logs more readable and valuable for debugging and monitoring.*

**8. How can you set up logging to capture log messages from multiple modules or classes in a Python application?**

**Ans:** *Setting up logging to capture log messages from multiple modules or classes in a Python application involves creating and configuring loggers, handlers, and formatters for each module or class as needed. Here's how you can do it:*

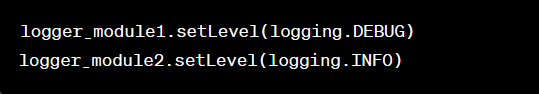
*1. Create a Logger for Each Module/Class:*

* *Start by creating a separate logger for each module or class that you want to log messages from. Typically, you create a logger using the module's name or a unique identifier. Use the `getLogger` method from the `logging` module:*

**

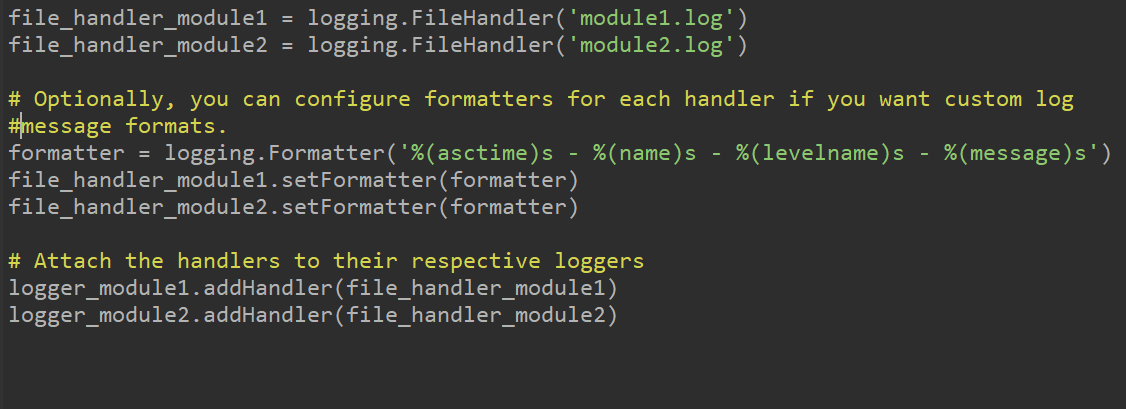
*2. Configure Log Levels:*

* *Set the desired log level for each logger to control which log messages are recorded. By default, loggers are set to the `WARNING` log level. You can adjust the log level using the `setLevel` method:*

**

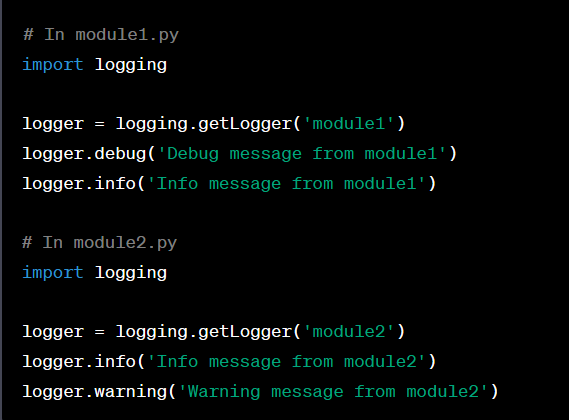
*3. Create Handlers for Each Logger:*

* *Create one or more handlers for each logger to specify where log messages should be sent. You can use different handlers for different log destinations, such as files, console, or remote servers. Here's an example of creating a file handler for each logger:*

**

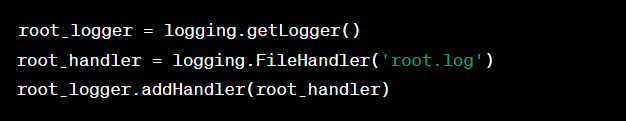
*4. Log Messages from Modules/Classes:*

* *In each module or class, use the logger created for that module or class to log messages. For example:*

**

*5.Configure the Root Logger (Optional):*

* *If you want to capture log messages that do not belong to a specific logger, configure the root logger using similar steps as above. This will capture log messages from modules or classes that haven't been explicitly configured.*

**

*6. Customize as Needed:*

* *You can further customize the logging setup by specifying different log levels, handlers, or formatters for each module or class. This flexibility allows you to capture log messages in different ways, depending on your application's requirements.*
* *By following these steps, you can set up logging to capture log messages from multiple modules or classes in your Python application while maintaining control over log levels, destinations, and formats for each logger. This enables fine-grained control over logging behavior and helps you manage and analyze log data effectively.*

**9. What is the difference between the logging and print statements in Python? When should you use logging over print statements in a real-world application?**

**Ans:** **T***he main differences between the `logging` module and `print` statements in Python are in their purpose, features, and flexibility. Here's a comparison of the two and when to use one over the other in a real-world application:*

*1. Purpose:*

* + *print` Statement: `print` statements are primarily used for debugging and providing quick output during development. They display information on the console, but they are not well-suited for long-term monitoring, error tracking, or capturing logs in production.*
  + *`logging` Module: The `logging` module is designed for structured and systematic logging. It is intended for recording information, events, errors, and other messages for various purposes, including debugging, monitoring, auditing, and performance analysis.*

*2. Features:*

* + *`print` Statements:*
  + *Simple and straightforward.*
  + *Output goes to the console (stdout).*
  + *No log levels or log formatting.*
  + *Not suitable for filtering or capturing logs in different ways.*
  + *Usually used for temporary debugging.*
  + *`logging` Module:*
  + *Offers log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) for categorizing messages.*
  + *Supports customizable log formatting.*
  + *Allows redirection of logs to various destinations (files, console, network, etc.) using handlers.*
  + *Enables filtering and configuring log output.*
  + *Provides log rotation, log rolling, and log retention features for managing log files.*

*3. Flexibility:*

* + *`print` Statements: Limited in terms of customization, filtering, and organization. Output is unstructured text, and it's challenging to distinguish different log messages or levels.*
  + *`logging` Module: Offers fine-grained control over log messages, including the ability to specify log levels, customize log formats, redirect logs to various locations, and filter logs based on criteria. This makes it more suitable for real-world application logging needs.*

*4. When to Use Each:*

* + *\*\*`print` Statements\*\*:*
  + *Use `print` statements during development for quick, temporary debugging and examination of variables and program flow.*
  + *They are handy for small scripts and one-off tasks.*
  + *\*\*`logging` Module\*\*:*
  + *Use the `logging` module in real-world applications, especially in production environments.*
  + *It's essential for long-term monitoring, error tracking, security auditing, performance analysis, and maintaining application health.*
  + *Log messages generated with `logging` can be configured to provide the right level of detail based on the context (e.g., debugging during development, high-level information in production).*
  + *Logging allows you to capture, filter, and analyze logs systematically, which is invaluable for diagnosing issues and improving software reliability.*

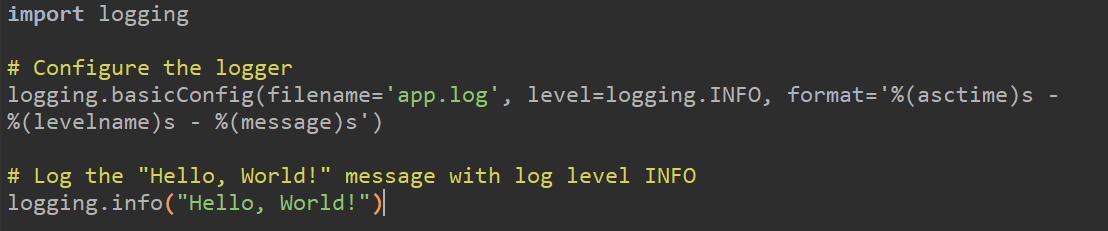
**10. Write a Python program that logs a message to a file named "app.log" with the following requirements:**

**● The log message should be "Hello, World!" ● The log level should be set to "INFO."**

**● The log file should append new log entries without overwriting previous ones. 11. Create a Python program that logs an error message to the console and a file named**

**"errors.log" if an exception occurs during the program's execution. The error message should include the exception type and a timestamp.**

**Ans:** *To achieve the requirements you've described; you can use the Python `logging` module. Here's a Python program that logs a message to a file named "app.log" with the specified requirements:*

**

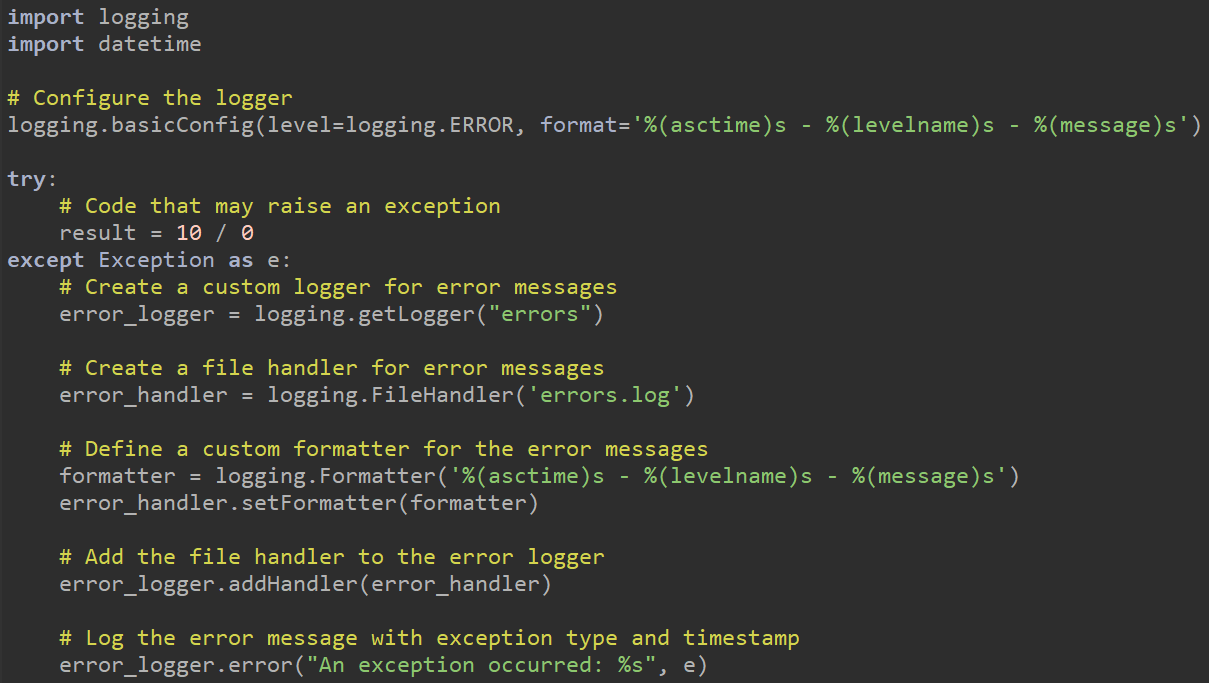
*In this program, we do the following:*

*1. We import the `logging` module.*

*2. We configure the logger using `basicConfig`. We specify the log file name as "app.log," set the log level to `INFO`, and define the log message format.*

*3. We log the "Hello, World!" message with an `INFO` log level. The message will be written to the "app.log" file.*

*Here's a Python program that logs an error message to both the console and a file named "errors.log" if an exception occurs during the program's execution. The error message includes the exception type and a timestamp:*

**

*In this program:*

*1. We configure the main logger to capture log messages with a log level of `ERROR`.*

*2. We wrap the code that may raise an exception in a `try...except` block. In this case, we divide 10 by 0, which will trigger a `ZeroDivisionError`.*

*3. If an exception occurs, we create a custom logger named "errors."*

*4. We create a file handler named `error\_handler` for logging error messages.*

*5. We define a custom formatter that includes the timestamp, log level, and the error message.*

*6. We attach the file handler and formatter to the error logger.*

*7. Finally, we log the error message with the exception type and a timestamp.*