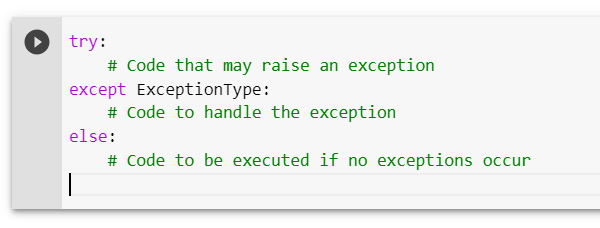
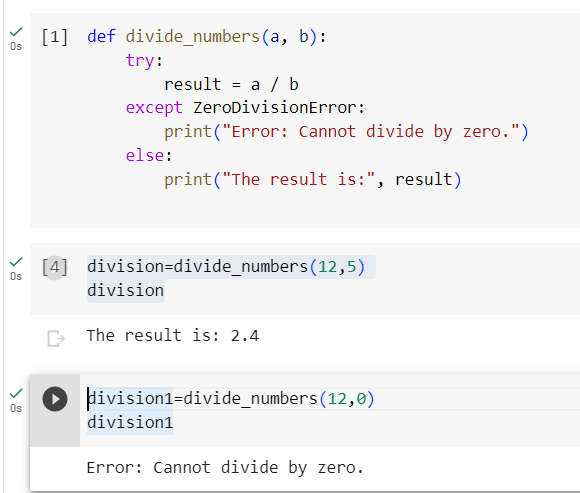
**1. What is the role of the 'else' block in a try-except statement? Provide an example scenario where it would be useful.**

Ans.) In a try-except statement, the 'else' block is optional and is executed only if no exceptions are raised in the corresponding 'try' block. It allows you to specify code that should run when the 'try' block completes successfully without any exceptions being raised.

**Syntax for try-except statement:**



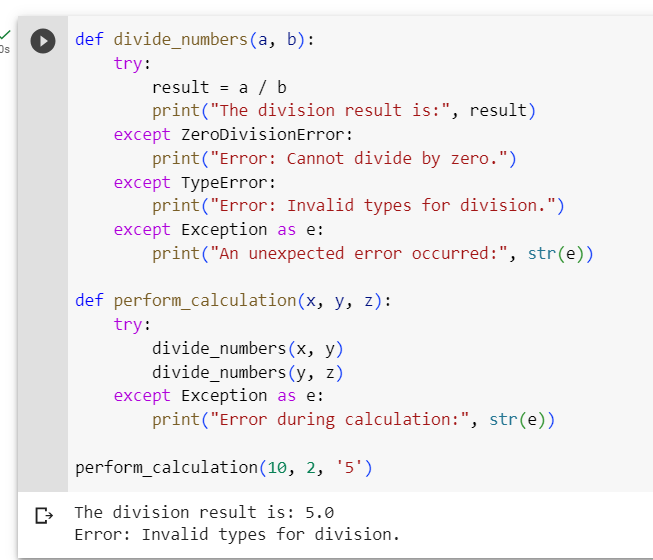
Ex.)



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

Ans.) Yes, a try-except block can be nested inside another try-except block. This is known as nested exception handling or nested try-except blocks. It allows you to handle different levels of exceptions in a hierarchical manner, providing more fine-grained error handling.

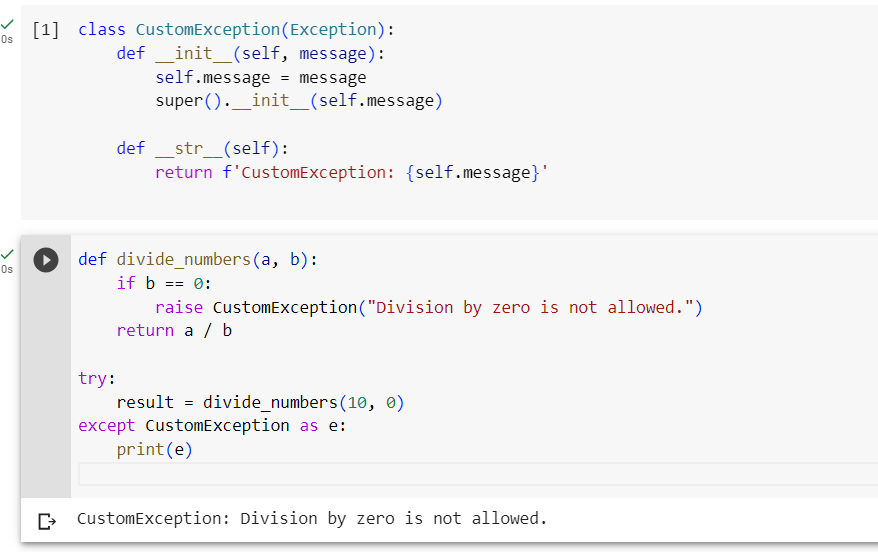
Example of nested exception handling:



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

Ans.) To create a custom exception class in Python, you can define a new class that inherits from the built-in **Exception** class or any of its subclasses.

Example that demonstrates how to create a custom exception class:



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

Ans.) Python provides a wide range of built-in exceptions that cover various types of errors and exceptional conditions. Here are some common exceptions that are frequently encountered in Python:

1. **SyntaxError**: Raised when there is a syntax error in the code.
2. **TypeError:** Raised when an operation or function is applied to an object of inappropriate type.
3. **NameError**: Raised when a local or global name is not found.
4. **ValueError**: Raised when a function receives an argument of the correct type but an inappropriate value.
5. **IndexError**: Raised when a sequence subscript is out of range.
6. **KeyError**: Raised when a dictionary key is not found.
7. **FileNotFoundError**: Raised when a file or directory is requested but cannot be found.
8. **IOError**: Raised when an input/output operation fails, such as if a file cannot be opened.
9. **ZeroDivisionError**: Raised when division or modulo operation is performed with zero as the divisor.
10. **AttributeError**: Raised when an attribute reference or assignment fails.
11. **ImportError**: Raised when an imported module or attribute cannot be found.
12. **OverflowError**: Raised when the result of an arithmetic operation exceeds the maximum representable value.
13. **MemoryError**: Raised when an operation fails to allocate necessary memory.
14. **RuntimeError**: Raised when an error occurs that doesn't belong to any specific category.
15. **KeyboardInterrupt**: Raised when the user interrupts the execution of the program by pressing Ctrl+C.

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

Ans.) Logging in Python refers to the process of recording events, messages, and other relevant information during the execution of a program. It involves the use of the built-in logging module in Python, which provides a flexible and powerful framework for generating log records.

Importance of Logging in Python:

**Debugging and troubleshooting**: Logging helps identify and fix issues by providing insights into the program's execution flow and error messages.

**Monitoring and performance optimization:** Logging allows real-time monitoring of performance metrics, enabling developers to identify and optimize code for better performance.

**Error tracking and analysis**: Logs provide valuable information about errors, aiding in reproducing and diagnosing issues for faster resolution.

**Audit trails and compliance**: Logging records important events and actions, serving as an audit trail for compliance purposes and forensic analysis.

**Understanding user behavior**: Logging user actions and interactions helps gather insights into user behavior and usage patterns.

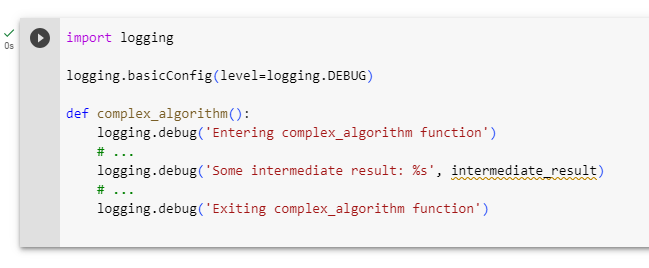
**Communication and collaboration:** Logging facilitate effective communication among team members by conveying important information and aiding in collaborative debugging and troubleshooting efforts.

**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 provide a way to categorize log messages based on their severity or importance. They help filter and control which log messages are emitted based on the configured log level. Python's logging module provides several predefined log levels, each serving a specific purpose.

Following are the commonly used log levels and examples of when each level would be appropriate:

**DEBUG:** The **DEBUG** level is used for detailed information helpful for debugging and development. It is typically used for low-level diagnostic information.

Example usage: 

**INFO**: The **INFO** level is used for informational messages that highlight the progress or important events in the program. It provides more significant information than DEBUG but is not as severe as warnings or errors.

Example usage:

A screenshot of a computer program

Description automatically generatedSS

**WARNING**: The **WARNING** level is used for non-critical issues or warnings that may indicate potential problems but don't necessarily require immediate attention.

Example usage:

A screenshot of a computer error

Description automatically generated

**ERROR**: The **ERROR** level is used to indicate error conditions that prevent a specific operation from completing successfully. These errors should be addressed as they may impact the program's functionality.

Example usage:

A screenshot of a computer program

Description automatically generated

**CRITICAL**: The **CRITICAL** level is used to indicate critical errors or failures that may lead to program termination or serious consequences. It represents the highest level of severity. Example usage:

A screen shot of a computer code

Description automatically generated

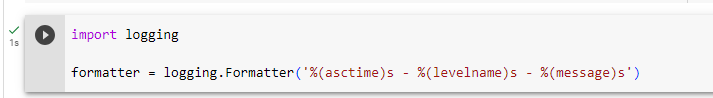
**7. What are log formatters in Python logging, and how can you customise the log message format using formatters?**

Ans.) In Python logging, log formatters define the structure and format of log messages. They allow you to customize how the log messages are presented, including the inclusion of timestamps, log levels, module names, and custom message formatting. Log formatters work in conjunction with handlers, which are responsible for determining where the log messages are sent.

The logging module provides the **Formatter** class to create log formatters.

customizing log message format using formatters:

**Create a Formatter instance**: Instantiate a **Formatter** object, specifying the desired log message format. The format is defined using placeholders that will be replaced with actual values at runtime.



**Configure a handler with the formatter**: Assign the formatter to a handler using the **setFormatter** method. This associates the formatter with the handler and ensures that log messages emitted through that handler will follow the specified format.

A screenshot of a computer program

Description automatically generated

**Assign the handler to a logger**: Attach the handler to a logger object using the **addHandler** method. This connects the handler to the logger, and any log messages generated by that logger will be processed by the handler with the associated formatter.



example of a more detailed log message format:



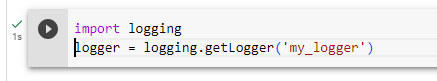
In the example above, we added the %(name)s placeholder to include the logger name in the log message format.

By customizing the log message format using formatters, you can structure log messages to include relevant information and make them more readable and informative for debugging, analysis, and monitoring purposes.

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

Ans.) To capture log messages from multiple modules or classes in a Python application, you can follow these steps:

**Create a Logger**: Create a logger object using the **logging.getLogger()** method. This logger will be used by different modules or classes to emit log messages.



**Configure Logging Handlers**: Configure one or more logging handlers to determine where the log messages will be sent. You can use different handlers for different purposes, such as writing to a file, displaying on the console, or sending to a remote server.

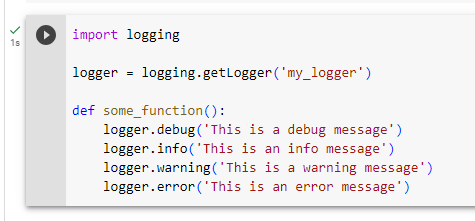


In the above example, we added both a **FileHandler** and a **StreamHandler** to the logger. This way, log messages will be written to the file specified by **file\_handler** and also displayed on the console.

**Set Logging Levels**: Set the desired logging levels for the logger and handlers. This determines the minimum severity level at which the log messages will be emitted.



**Import and Use the Logger:** Import the logger object in the modules or classes where you want to emit log messages. Use the logger to emit log messages at appropriate points in the code.



In the above example, the logger is imported and used in the **some\_function()** to emit log messages of different levels.

**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.)

|  |  |
| --- | --- |
| **Logging** | **Print Statements** |
| Logging is a dedicated mechanism for generating log messages during the execution of a program. | Print statements are a simple way to display information or debug output during program execution. |
| Logging provides more control and flexibility in capturing and managing log messages. | Print statements are typically used for quick and temporary debugging purposes. |
| Logging supports advanced features like log formatting, log rotation, and logging to multiple destinations simultaneously. | Print statements are easy to use but lack the advanced features and configurability provided by logging. |
| Logging supports advanced features like log formatting, log rotation, and logging to multiple destinations simultaneously. | Print statements provide less control over log messages compared to logging. |

While both logging and print statements can be used to display information during program execution, logging is the preferred choice in real-world applications for the following reasons:

1. **Flexibility and Control**: Logging provides more control and flexibility in capturing, categorizing, and managing log messages based on severity levels and destinations.
2. **Configurability**: Logging allows you to configure different log levels, filters, formatting, and output targets, making it suitable for various scenarios and environments.
3. **Long-term Logging and Analysis:** Logging is designed for long-term logging and is crucial in production environments for troubleshooting, monitoring, auditing, and analysis of logs.
4. **Advanced Features**: Logging supports advanced features like log rotation, log formatting, logging to multiple destinations, and integration with external logging systems.
5. **Granular Debugging**: Logging allows you to log messages at different levels of severity, enabling granular debugging and the ability to filter log messages based on severity levels.

**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.**

Ans.) 

In this program, we use the **basicConfig** method from the logging module to configure the logging. Here's a breakdown of the parameters:

**filename**: Specifies the name of the log file to which the log entries will be written. In this case, it is set to "app.log".

**level**: Sets the log level to "INFO". This means that only log messages with the level "INFO" and above will be emitted.

**filemode**: Sets the file mode to "a" (append). This ensures that new log entries are appended to the existing file without overwriting the previous ones.

Finally, we log the message "Hello, World!" using the **info** method of the **logging** module. This message will be written to the "app.log" file in the specified log format.

**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.) 