In [ ]:
```
Que.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 in Python, the else block is optional and is executed only if
       no exceptions are raised in the corresponding try block.
EXAMPLES
try:
    # Code that might raise an exception
except ExceptionType1:
    # Code to handle ExceptionType1
except ExceptionType2:
    # Code to handle ExceptionType2
else:
    # Code to execute if no exceptions are raised in the try block
```

In [ ]:
```
Que.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, and it allows for more granular control over handling different
     types of exceptions in different parts of the code. Each nested try-except block can handle
     specific exceptions that might occur in its corresponding code block.
    def nested_exception_example():
EXAMPLE:
    try:
        # Outer try block
        numerator = int(input("Enter the numerator: "))
        denominator = int(input("Enter the denominator: "))

        try:
            # Inner try block
            result = numerator / denominator
        except ZeroDivisionError:
            print("Inner Except Block: Cannot divide by zero!")
        else:
            print("Inner Else Block: The result is:", result)

    except ValueError:
        print("Outer Except Block: Please enter valid integers for numerator and denominator.")
    except Exception as e:
        print("Outer Except Block: An error occurred:", e)

# Example usage
nested_exception_example()
```

In [ ]:
```
Que.3:  How can you create a custom exception class in Python? Provide an example that
        demonstrates its usage.
Ans:  we can create a custom exception class by inheriting from the built-in Exception class or
      one of its subclasses
class CustomError(Exception):
    """Custom exception class."""
    def __init__(self, message="A custom error occurred"):
        self.message = message
        super().__init__(self.message)

# Example of using the custom exception
def example_function(value):
    if value < 0:
        raise CustomError("Input value cannot be negative.")

try:
    user_input = int(input("Enter a number: "))
    example_function(user_input)
except CustomError as ce:
    print(f"Caught an exception: {ce}")
except ValueError:
    print("Invalid input. Please enter a valid number.")
else:
    print("No exception occurred.")
finally:
    print("This block will be executed no matter what.")
```

In [ ]:
```

```

In [ ]:
```
Que, 4: . What are some common exceptions that are built-in to Python?
Ans: There are some common built-in excepton in Python:
ValueError, TypeError, FileNotFoundError, IOError, IndexError, KeyError, AttributeError, ZeroDivisionError,
ImportError, NameError
```

In [ ]:
```
Q.5 :  What is logging in Python, and why is it important in software development?
Ans:
Logging in Python refers to the process of recording messages, events, or information
generated during the execution of a program. The logging module in Python provides a
flexible and powerful framework for emitting log messages from applications.
The logging module in Python supports various logging levels (DEBUG, INFO, WARNING,
ERROR, CRITICAL), handlers (to define where log messages go), formatters
(to control the log message format), and filters (to selectively filter log records).
Properly implemented logging enhances the maintainability, reliability, and overall quality
of software applications.
```

In [ ]:
```
Que.6: Explain the purpose of log levels in Python logging and provide examples of when
each log level would be appropriate.
Ans: log levels are used to categorize log messages based on their severity or importance.
    The purpose of log levels is to allow developers and system administrators to control the
    amount and type of information that is logged.
1: DEBUG:
Purpose: Detailed information, typically useful only for diagnosing problems or debugging.
Example Use Cases:
Printing variable values during development.
Detailed function call traces.
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug("This is a debug message.")

2:INFO:
Purpose: General information about the program's operation. Used to confirm that things are working as expected.
Example Use Cases:
Displaying startup information.

import logging

logging.basicConfig(level=logging.INFO)
logging.info("This is an info message.")
```

In [ ]:
```
Q.7: What are log formatters in Python logging, and how can you customise the log
message format using formatters?
Ans: log formatters are used to define the layout and structure of log messages.
    They allow developers to customize the way log records are formatted before they are
    output to different destinations such as the console, files, or external services
Formatters help in creating a consistent and readable log output by specifying the format of
timestamps, log levels, messages, and other relevant information.

To customize the log message format using formatters, you need to create a formatter object and
associate it with a Handler or a Logger.
```

In [ ]:
```
Que.8: How can you set up logging to capture log messages from multiple modules or
classes in a Python application?
 setting up logging to capture log messages from multiple modules or classes involves creating
    a central logging configuration and then using that configuration across various modules and classes
```

In [ ]:
```
Que 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?


While print statements are quick and handy for debugging during development, logging provides a
more sophisticated and scalable solution for managing information in real-world applications, especially
in production environments. The use of logging becomes particularly crucial as the complexity of the application
and the need for detailed information and error tracking increase.
```

In [ ]:
```
Que.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.

import logging

def configure_logging():
    logging.basicConfig(
        filename='app.log',
        level=logging.INFO,
        format='%(asctime)s - %(levelname)s - %(message)s',
        datefmt='%Y-%m-%d %H:%M:%S',
    )

def main():
    # Configure logging
    configure_logging()

    # Log "Hello, World!" with INFO level
    logging.info("Hello, World!")

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

In [ ]:
```
Que. 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: import logging
import sys

def configure_logging():
    # Create a logger
    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)  # Set the root logger's level to the lowest level

    # Create a console handler and set the level to ERROR
    console_handler = logging.StreamHandler(sys.stdout)
    console_handler.setLevel(logging.ERROR)

    # Create a file handler and set the level to ERROR
    file_handler = logging.FileHandler('errors.log')
    file_handler.setLevel(logging.ERROR)

    # Create a formatter
    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s', datefmt='%Y-%m-%d %H:%M:%S')

    # Set the formatter for both handlers
    console_handler.setFormatter(formatter)
    file_handler.setFormatter(formatter)

    # Add the handlers to the logger
    logger.addHandler(console_handler)
    logger.addHandler(file_handler)

def main():
    # Configure logging
    configure_logging()

    try:
        # Your program logic here
        result = 10 / 0  # This will raise a ZeroDivisionError
    except Exception as e:
        # Log the exception with details to both console and file
        logging.exception(f"An exception occurred: {type(e).__name__}")

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