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

The 'else' block in a try-except statement in Python is used to define a set of statements that should be executed when no exceptions are raised in the corresponding 'try' block. It comes after the 'try' block and before the 'finally' block (if present).

```
def calculate_sum_from_file(filename):
  try:
    with open(filename, 'r') as file:
      data = [int(line) for line in file]
  except FileNotFoundError:
    print(f"Error: File '{filename}' not found.")
  except ValueError:
    print("Error: Unable to convert data to integers.")
  else:
    # The 'else' block will only run if no exceptions were raised
    total_sum = sum(data)
    print(f"Sum of numbers in '{filename}': {total_sum}")

# Example usage
calculate_sum_from_file("data.txt")
```

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

Yes, a try-except block can be nested inside another try-except block in Python. This is known as nested exception handling, and it allows for handling different levels of exceptions in a more fine-grained manner.

```
def divide_and_read_data(filename):
  try:
    # Perform division
    num1 = int(input("Enter a numerator: "))
    num2 = int(input("Enter a denominator: "))
    result = num1 / num2
    print(f"Result of division: {result}")

    try:
      # Read data from the file
      with open(filename, 'r') as file:
        data = [int(line) for line in file]
      print(f"Data from '{filename}': {data}")
    except FileNotFoundError:
      print(f"Error: File '{filename}' not found.")
    except ValueError:
```

```
        print("Error: Unable to convert data to integers.")
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")

# Example usage
divide_and_read_data("data.txt")
```

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

In Python, you can create a custom exception class by subclassing the built-in Exception class or any other existing exception class. This allows you to define your own exception type with customized behavior and error messages.

```
class CustomError(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return f"CustomError: {self.message}"


def perform_operation_with_positive_number(number):
    if number < 0:
        raise CustomError("Negative numbers are not allowed.")
    # Perform the desired operation with the positive number
    return number * 2


# Example usage
try:
    num = int(input("Enter a positive number: "))
    result = perform_operation_with_positive_number(num)
    print(f"Result: {result}")
except CustomError as e:
    print(e)
except ValueError:
    print("Error: Please enter a valid integer.")
```

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

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

IndentationError: Raised when there is an indentation-related syntax error.

NameError: Raised when a name or variable is not found in the current scope.

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

ValueError: Raised when a function receives an argument of the correct type but an inappropriate value.

KeyError: Raised when a dictionary key is not found.

IndexError: Raised when a sequence (list, tuple, string, etc.) index is out of range.

ZeroDivisionError: Raised when attempting to divide by zero.

FileNotFoundError: Raised when trying to open or access a file that does not exist.

IOError: Raised for various input/output-related errors.

ImportError: Raised when importing a module or package fails.

AttributeError: Raised when an attribute or method is not found on an object.

StopIteration: Raised to signal the end of an iterator.

KeyboardInterrupt: Raised when the user interrupts the program with Ctrl+C.

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

Logging in Python refers to the process of recording events, messages, and diagnostic information during the execution of a program. It is accomplished through the Python logging module, which provides a flexible and powerful logging framework to handle various log messages.

Importance of logging in software development:

- Debugging and Troubleshooting
- Error Reporting
- Monitoring and Performance Optimization
- Auditing and Compliance
- Reproducibility
- Non-Intrusive and Configurable
- Communication and Collaboration

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

Log levels in Python logging are used to categorize log messages based on their severity and importance. The logging module provides several predefined log levels that allow developers to control which messages get displayed or recorded in the log output.
Some examples of when each log level would be appropriate:

DEBUG:
Example: Printing variable values or the flow of program execution for detailed debugging purposes.

INFO:
Example: Logging when the application starts, stops, or reaches a significant milestone.

WARNING:
Example: Logging when a deprecated function or API is used, indicating a potential issue in the future.

ERROR:
Example: Logging when an operation fails due to an unexpected condition.

CRITICAL:
Example: Logging when a database connection fails, causing the application to halt.

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

Log formatters are used to control the format of log messages before they are output to the specified logging destination, such as a file, console, or network stream. Formatters allow you to customize the appearance of log messages, including the date, time, log level, logger name, and the actual message itself.
The logging module provides a variety of built-in formatters, but you can also create your custom formatters to suit specific logging requirements.
The key components of a log formatter are placeholders enclosed in curly braces {}. These placeholders are replaced with relevant information when the log message is generated.

```
# Create a formatter with a custom log message format
formatter = logging.Formatter('{asctime} - {name} - {levelname} - {message}', style='{')

# Set the formatter for the file handler
file_handler.setFormatter(formatter)

# Add the file handler to the logger
logger.addHandler(file_handler)

# Now, log messages with different log levels
```

```
logger.debug('This is a debug message.')
logger.info('This is an info message.')
logger.warning('This is a warning message.')
logger.error('This is an error message.')
logger.critical('This is a critical message.')
```

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

To set up logging to capture log messages from multiple modules or classes in a Python application, you can follow these steps:

- Create a Logger Object: First, create a logger object using the logging.getLogger() method. You can provide a unique name to the logger to distinguish it from other loggers.
- Set Logger Level: Set the logging level for the logger using the setLevel() method. This level determines which log messages will be captured by the logger. For example, if the logger level is set to logging.DEBUG, all log messages (DEBUG, INFO, WARNING, ERROR, CRITICAL) will be captured.
- Create and Configure Handlers: Create log handlers to determine where the log messages will be sent. You can use handlers like FileHandler, StreamHandler, etc., depending on where you want to direct the log output.
- Create and Set Formatter: Create a log formatter using the Formatter class to specify the format of the log messages. Then, set this formatter on each handler to control how log messages will appear in the log output.
- Add Handlers to the Logger: Add the created handlers to the logger using the addHandler() method. This step allows the logger to send log messages to the specified destinations.
- Use the Logger in Modules/Classes: In each module or class that needs logging, retrieve the logger using logging.getLogger() with the same name you used while creating the logger. You can then use this logger to generate log messages with different log levels (e.g., logger.debug(), logger.info(), etc.).

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?**

print: The print statement is primarily used for basic output to the console or standard output. It is a quick way to display messages or values during the development and debugging process.

logging: The logging module is designed for more advanced and structured logging. It allows you to record various log messages with different log levels and direct them to different destinations, such as files, the console, or remote servers. Logging is meant for comprehensive logging in both development and production environments.

print: The print statement doesn't provide different log levels. It always prints the output to the console with no distinction between informational messages and debugging messages.

logging: The logging module offers various log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) that allow you to control the verbosity of the log output. You can specify the desired log level to limit or filter the log messages based on their importance.
print: The print statement is simple and straightforward but lacks the flexibility to manage log output in complex scenarios.
logging: The logging module provides extensive features and customization options. It allows you to configure multiple log handlers, set log levels per logger, define custom formatters, and send logs to different destinations.
print: Using print statements for debugging purposes is acceptable during the development phase. However, in production environments, excessive print statements can be problematic, as they can clutter the output, slow down the application, and potentially expose sensitive information.
logging: Logging is the recommended approach for production use. It allows you to capture critical information about the application's behavior while avoiding the downsides of using print statements in production.
In a real-world application:
For debugging and development purposes, using print statements can be convenient and straightforward. It helps quickly inspect values and program flow during the early stages of development.
However, for real-world applications and especially in production environments, it is recommended to use the logging module. Logging offers better control over log output, allows the separation of log levels, and provides the ability to direct logs to different destinations, making it easier to manage and analyze application behavior. Additionally, logging supports comprehensive logging with log levels that enable you to filter messages based on their severity, which is crucial for monitoring, troubleshooting, and maintaining production systems.

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 setup_logger():
    # Create a logger
    logger = logging.getLogger('my_logger')
    logger.setLevel(logging.INFO)

    # Create a file handler and set it to append mode
    file_handler = logging.FileHandler('app.log', mode='a')
    file_handler.setLevel(logging.INFO)
```

```
    # Create a formatter with a custom log message format
    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

    # Set the formatter for the file handler
    file_handler.setFormatter(formatter)

    # Add the file handler to the logger
    logger.addHandler(file_handler)

    return logger

def main():
    logger = setup_logger()

    # Log the "Hello, World!" message
    logger.info('Hello, World!')

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

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.**

```
import logging
import traceback
import datetime

def setup_logger():
    # Create a logger
    logger = logging.getLogger('my_logger')
    logger.setLevel(logging.ERROR)

    # Create a console handler
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.ERROR)

    # Create a file handler and set it to append mode
    file_handler = logging.FileHandler('errors.log', mode='a')
```

```
    file_handler.setLevel(logging.ERROR)

    # Create a formatter with a custom log message format
    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

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

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

    return logger

def main():
    logger = setup_logger()

    try:
        # Your program logic here
        result = 10 / 0  # This will raise a ZeroDivisionError
    except Exception as e:
        # Log the error message with exception type and timestamp
        error_message = f"Exception: {type(e).__name__}, Time: {datetime.datetime.now()}"
        logger.error(error_message)
        logger.error(traceback.format_exc())  # Also log the full traceback

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