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 an optional block that follows all the except blocks. It is executed only if no exceptions occur in the try block. The purpose of the else block is to specify code that should be executed when the try block completes successfully, without any exceptions being raised.

The else block is useful when you want to perform certain actions that should only be executed when no exceptions occur. It provides a way to separate the code that might raise exceptions from the code that should execute only if everything goes smoothly.

Here's an example scenario where the else block would be useful:

python

```
try:
    file = open("data.txt", "r")
    data = file.read()
except FileNotFoundError:
    print("Error: File not found.")
except IOError:
    print("Error: Unable to read the file.")
else:
    file.close()
    print("File successfully read.")
    # Additional code that depends on the successful file read
```

In this example, the try block attempts to open a file named "data.txt" for reading and reads its contents. There are two except blocks specified to handle different types of exceptions that may occur: FileNotFoundError and IOError. If either of these exceptions occurs, the corresponding except block is executed, displaying an error message.

However, if no exceptions occur and the file is successfully read, the code in the else block is executed. In this case, the file is closed, and a success message is printed. Additionally, you can include any additional code or operations that depend on the successful file read within the else block.

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 for more fine-grained exception handling, where the outer try-except block handles exceptions at a higher level, while the inner try-except block handles exceptions at a more specific level.

Here's an example to illustrate nested try-except blocks:

```
python
try:
    # Outer try block
    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))

    try:
        # Inner try block
        result = num1 / num2
        print("Result:", result)
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")

except ValueError:
    print("Error: Invalid input. Please enter valid integers.")
```

In this example, there are two levels of try-except blocks. The outer try-except block handles the possibility of a ValueError when converting the user's input to integers. If a ValueError occurs, the outer except block is executed and an error message is displayed.

Inside the outer try block, there is an inner try-except block. This block attempts to perform a division operation with the user's input. If the user enters zero as the second number, a ZeroDivisionError occurs, and the inner except block is executed, printing an appropriate error message.

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

Ans:

n Python, you can create a custom exception class by defining a new class that inherits from the built-in Exception class or any of its subclasses. By creating a custom exception class, you can define your own exception types with specific behaviors and attributes.

Here's an example that demonstrates how to create a custom exception class and use it:

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

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

# Usage example
def divide_numbers(num1, num2):
    if num2 == 0:
        raise CustomException("Division by zero is not allowed.")
    return num1 / num2

try:
```

```
    result = divide_numbers(10, 0)
    print("Result:", result)
except CustomException as e:
    print("An error occurred:", str(e))
```

In this example, we define a custom exception class called CustomException that inherits from the base Exception class. The custom exception class has an initializer (__init__) method that accepts a message parameter. We also define a string representation (__str__) method to provide a formatted string when the exception is raised and printed.

We then use the custom exception class in a function called divide_numbers(). If the second number passed to the function is zero, we raise a CustomException with a specific error message.

In the try-except block, we attempt to call the divide_numbers() function with arguments 10 and 0, which triggers the custom exception to be raised. The except block catches the CustomException and prints the error message.

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

Ans: Python provides several built-in exceptions that are commonly used to handle various exceptional situations. Here are some of the most commonly used built-in exceptions in Python:

ZeroDivisionError: Raised when division or modulo operation is performed with zero as the divisor.
ValueError: Raised when a function receives an argument of the correct type but with an invalid value.
TypeError: Raised when an operation or function is applied to an object of an inappropriate type.
IndexError: Raised when attempting to access an index that is out of range for a sequence (e.g., list, string).
KeyError: Raised when attempting to access a dictionary key that doesn't exist.
FileNotFoundError: Raised when a file or directory is requested, but it cannot be found.
IOError: Raised when an I/O operation fails, such as opening or reading a file.
NameError: Raised when a local or global name is not found.
AttributeError: Raised when an attribute reference or assignment fails.
ImportError: Raised when an import statement fails to import a module.
TypeError: Raised when an operation or function is applied to an object of an inappropriate type.
OverflowError: Raised when the result of an arithmetic operation is too large to be represented.
MemoryError: Raised when an operation fails due to insufficient memory.
RuntimeError: Raised when a generic runtime error occurs that doesn't belong to any specific built-in exception.

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. The logging module in Python provides a flexible and configurable logging system that allows developers to track and record valuable information about the application's behavior, performance, errors, and other events.

Logging is crucial in software development for several reasons:

Debugging and Troubleshooting: Logging helps in identifying and diagnosing issues in software applications. By logging relevant information, such as error messages, stack traces, variable values, and program flow, developers can trace the execution and understand the state of the application during runtime. This assists in locating and

fixing bugs and provides insights into unexpected behaviors.

Monitoring and Auditing: Logging enables monitoring and auditing of application events. By logging important events, such as user actions, system events, or security-related activities, developers and system administrators can track the application's behavior, detect anomalies, and gain insights into its usage and performance.

Performance Analysis: Logging performance-related information, such as execution times, resource utilization, or database query durations, allows developers to analyze and optimize the performance of their applications. It helps in identifying bottlenecks, optimizing code, and improving overall system efficiency.

Security and Compliance: Logging plays a crucial role in security and compliance by recording security-related events, such as login attempts, access control violations, or system changes. It aids in monitoring for suspicious activities, identifying potential security breaches, and meeting compliance requirements.

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

Ans: In Python logging, log levels are used to categorize log messages based on their importance or severity. The logging module in Python provides several predefined log levels that allow developers to control the verbosity and granularity of the logged information. These log levels help in filtering and managing log messages based on their significance.

Here are the commonly used log levels in Python logging, along with examples of when each log level would be appropriate:

DEBUG: This is the lowest log level and is typically used for detailed information useful for debugging. It is suitable for messages that provide fine-grained details about the program's execution, variable values, or intermediate results. Example usage:

python

```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug("Entering function foo()")  # Debugging information
logging.debug("Variable x = %d", x)  # Debugging information with variable value
```

INFO: The INFO log level is used to provide general information about the program's execution. It is suitable for messages that convey important progress or significant events that can help in understanding the program's behavior. Example usage:

python

```
import logging

logging.basicConfig(level=logging.INFO)
logging.info("Processing started")  # Information about the start of a process
logging.info("Data loaded successfully")  # Information about successful data loading
```

WARNING: The WARNING level indicates a potential issue or a condition that might lead to an error. It is used to highlight abnormal or unexpected situations that are not critical but require attention. Example usage:

python

Copy code

```
import logging

logging.basicConfig(level=logging.WARNING)
logging.warning("Disk space running low")  # Warning about low disk space
logging.warning("Deprecated function called")  # Warning about deprecated function
usage
```

ERROR: The ERROR level signifies errors that cause a significant failure or unexpected behavior but do not terminate the program. It is appropriate for messages related to errors that need to be investigated or handled. Example usage:

python

```
import logging

logging.basicConfig(level=logging.ERROR)
logging.error("Failed to open file")  # Error message indicating file opening failure
logging.error("Invalid input data")  # Error message indicating invalid input
```

CRITICAL: The CRITICAL level represents the highest severity level. It indicates critical errors that may lead to program termination or complete system failure. It is used for exceptional and severe conditions that require immediate attention. Example usage:

python

```
import logging

logging.basicConfig(level=logging.CRITICAL)
logging.critical("Application crashed")  # Critical error indicating application crash
logging.critical("Database connection lost")  # Critical error indicating loss of databa
```

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 are used to specify the format of log messages. Formatters determine how log records are transformed into a final log message that is displayed or written to the desired output stream, such as the console or a log file. They allow developers to customize the structure and content of log messages to meet their specific requirements.

The logging module in Python provides the Formatter class, which can be used to define log message formats. A log formatter is associated with a handler (e.g., StreamHandler, FileHandler) and is responsible for formatting the log records produced by the logger.

Here's an example that demonstrates how to customize log message format using formatters:

python

```
import logging

# Create a logger and set its level
logger = logging.getLogger("my_logger")
logger.setLevel(logging.DEBUG)

# Create a formatter
```

```
formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")

# Create a handler and set the formatter
stream_handler = logging.StreamHandler()
stream_handler.setFormatter(formatter)

# Add the handler to the logger
logger.addHandler(stream_handler)

# Log messages
logger.debug("This is a debug message")
logger.info("This is an info message")
logger.warning("This is a warning message")
```

In this example, we create a logger named "my_logger" and set its level to logging.DEBUG. We then create a formatter using logging.Formatter and specify the desired format by passing a format string as an argument. The format string can contain placeholders such as %(asctime)s for the timestamp, %(levelname)s for the log level, and %(message)s for the log message itself.

Next, we create a StreamHandler to send the log messages to the console. We associate the formatter with the handler using setFormatter().

Finally, we add the handler to the logger using addHandler(). The log messages logged using the logger will be formatted according to the specified format string.

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 set up a logging configuration that allows logging to be centralized and shared across the entire application. This can be achieved by following these steps:

Create a logger: In each module or class where you want to log messages, create an instance of the logging.Logger class. You can use the module or class name as the logger name to easily identify the source of the log messages.

python

```
import logging

logger = logging.getLogger(__name__)
```

Set the logging level: Set the desired logging level for each logger. The logging level determines which log messages will be recorded. For example, you can set the level to logging.DEBUG to capture all log messages, or set it to logging.INFO to capture only informational messages and above.

python

```
logger.setLevel(logging.DEBUG)
```

Configure a common logging handler: Create a logging handler that will handle the log messages. This can be a StreamHandler for console output or a FileHandler for writing to a file. You can set the desired log level and formatter for the handler.

python

```
stream_handler = logging.StreamHandler()
stream_handler.setLevel(logging.DEBUG)

formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
```

stream_handler.setFormatter(formatter)

Add the handler to the logger: Add the logging handler to each logger created in Step 1 using the addHandler() method. This ensures that the log messages from different modules or classes are captured by the same handler.

python

logger.addHandler(stream_handler)

Start logging: Start logging messages using the loggers created in Step 1. Use the appropriate logging methods (debug(), info(), warning(), error(), critical()) to log messages at the desired log levels.

python

Copy code

logger.debug("This is a debug message")

logger.info("This is an info message")

logger.warning("This is a warning message")

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: he logging and print statements serve different purposes in Python, and their usage depends on the specific requirements and context of the application.

Purpose and Output: The primary purpose of the print statement is to display information to the console or standard output. It is typically used for debugging or quick information display during development. On the other hand, the logging module is designed for flexible and configurable logging of events, messages, and other information during the execution of a program. Logging provides a more structured and centralized approach to capturing and managing log messages.

Flexibility and Control: The logging module offers more flexibility and control over log messages compared to print. With logging, you can define different log levels (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL), set logging levels for different parts of the application, and configure log handlers to control where log messages are directed (e.g., console, file, network). You can also define custom log formats and have timestamps, log levels, and other contextual information automatically included in the log messages.

Granularity and Filtering: With logging, you can control the verbosity and granularity of log messages by setting different log levels for different parts of the application. This allows you to log only the necessary information based on the severity or significance of events. Additionally, the logging module provides options for filtering and routing log messages to different outputs or storage systems, which is essential for large-scale or distributed applications.

Maintenance and Production Use: logging is particularly useful for long-term maintenance and production use of an application. It allows you to have a consistent logging approach throughout the codebase, making it easier to track down issues, troubleshoot errors, and monitor the application's behavior. Logging also enables logging to be disabled or redirected in production environments without modifying the code, which can be helpful for security, performance, or privacy concerns.

In a real-world application, it is generally recommended to use logging over print statements for logging purposes. Logging provides a more sophisticated and controlled approach to capturing and managing log messages, allowing for flexibility, filtering, and customization. It facilitates debugging, troubleshooting, monitoring, and

maintenance of the application, making it easier to identify issues and track the application's behavior in different environments. print statements, on the other hand, are more suitable for quick debugging or ad hoc output during development but lack the extensive capabilities and configurability of the logging module.

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: To meet the requirements of logging a message to a file named "app.log" with the log message "Hello, World!" and a log level of "INFO," you can use the logging module in Python. Here's an example program that demonstrates this:

    python

    ```
    import logging

    # Configure the logging
    logging.basicConfig(
        level=logging.INFO,
        filename="app.log",
        filemode="a",
        format="%(asctime)s - %(levelname)s - %(message)s"
    )

    # Log the message
    logging.info("Hello, World!")
    ```

    In this example:

    We import the logging module.

    We use the basicConfig() function to configure the logging. Here's a breakdown of the parameters used:

    level=logging.INFO: Sets the logging level to INFO, so only INFO level messages and above will be recorded.

    filename="app.log": Specifies the name of the log file as "app.log".

    filemode="a": Sets the file mode to "append" ("a"), so new log entries are appended to the existing file without overwriting it.

    format="%(asctime)s - %(levelname)s - %(message)s": Specifies the log message format, including the timestamp (asctime), log level (levelname), and actual log message (message).

    We use the logging.info() function to log the message "Hello, World!" with the log level set to INFO. This message will be recorded in the "app.log" file.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.

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 log an error message to both the console and a file named "errors.log" in case of an exception during program execution, you can use the logging module in Python. Here's an example program that demonstrates this:

python

```
import logging

# Configure the logging
logging.basicConfig(
    level=logging.ERROR,
    format="%(asctime)s - %(levelname)s - %(message)s",
    handlers=[
        logging.FileHandler("errors.log", mode="a"),
        logging.StreamHandler()
    ]
)

try:
    # Code that may raise an exception
    num1 = 10
    num2 = 0
    result = num1 / num2

except Exception as e:
    # Log the exception
    logging.error(f"An exception occurred: {str(e)}")
```

In this example:

We import the logging module.
We use the basicConfig() function to configure the logging. Here's a breakdown of the parameters used:
level=logging.ERROR: Sets the logging level to ERROR, so only ERROR level messages and above will be recorded.
format="%(asctime)s - %(levelname)s - %(message)s": Specifies the log message format, including the timestamp (asctime), log level (levelname), and actual log message (message).
handlers=[logging.FileHandler("errors.log", mode="a"), logging.StreamHandler()]: Sets up two log handlers - FileHandler to write to the "errors.log" file in "append" mode (mode="a"), and StreamHandler to log to the console.
Within the try-except block:

We have the code that may raise an exception. In this example, we intentionally divide a number by zero to trigger an exception.
In the except block, we log the exception using the logging.error() function. The error message includes the exception type and its string representation (str(e)).