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 is used to define a block of code that should be executed if no exceptions occur within the corresponding try block. It provides a way to separate the code that can potentially raise an exception from the code that should be executed when no exceptions are raised.

Here's an example to illustrate the role of the 'else' block:

try:

num1 = int(input("Enter the first number: "))

num2 = int(input("Enter the second number: "))

result = num1 / num2

except ZeroDivisionError:

print("Error: Division by zero")

else:

print("The division was successful.")

print("Result:", result)

In this example, the program attempts to divide two numbers entered by the user. If a ZeroDivisionError occurs (i.e., if the user enters zero as the second number), the except block is executed, and an error message is printed.

However, if no exceptions occur during the division, the 'else' block is executed. In this case, it prints a message indicating that the division was successful, followed by the result of the division.

The 'else' block is only executed if no exceptions are raised within the corresponding try block. It allows you to specify code that should be executed as a result of the successful execution of the try block, separate from the exception handling code in the except block.

Using the 'else' block can help improve the clarity of your code and make it more readable by explicitly distinguishing between the normal code path and the exception handling code path.

**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. This allows for more granular exception handling and provides the ability to handle different levels of exceptions separately.

Here's an example of nested try-except blocks

try:

# Outer try block

num1 = int(input("Enter the numerator: "))

num2 = int(input("Enter the denominator: "))

try:

# Inner try block

result = num1 / num2

print("Result:", result)

except ZeroDivisionError:

print("Error: Division by zero in the inner try block")

except ValueError:

print("Error: Invalid input in the outer try block")

In this example, there are two levels of try-except blocks. The outer try block handles ValueError, which may occur if the user enters a non-numeric value for num1 or num2. If such an exception occurs, the outer except block is executed, and an error message is printed.

The inner try block handles ZeroDivisionError, which may occur if the user enters zero as the value for num2. If a ZeroDivisionError occurs within the inner try block, the inner except block is executed, and another error message is printed.

By nesting try-except blocks, you can handle exceptions at different levels of code, providing more specific exception handling based on the context. This allows for a more fine-grained approach to handle exceptions in a way that suits the specific needs of the program.

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

**demonstrates its usage.**

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. By creating a custom exception class, you can define your own exception types with specific behaviors and error messages.

Here's an example that demonstrates the creation and usage of a custom exception class:

class CustomException(Exception):

pass

def divide(num1, num2):

if num2 == 0:

raise CustomException("Cannot divide by zero")

return num1 / num2

try:

result = divide(10, 0)

print("Result:", result)

except CustomException as e:

print("An error occurred:", e)

In this example, we define a custom exception class called CustomException that inherits from the Exception class. The CustomException class doesn't have any additional functionality, but you can add custom attributes and methods as needed.

The divide() function is used to divide two numbers. If the second number is zero, we raise an instance of the CustomException class with a specific error message.

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

By creating and using custom exception classes, you can add more specific error handling to your code, allowing for better categorization and differentiation of exceptions based on their context or specific error conditions.

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

Python provides several built-in exceptions that cover a wide range of common error scenarios. Some of the most commonly used built-in exceptions include:

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

IndentationError: Raised when there is an indentation-related error, such as incorrect or inconsistent indentation.

TypeError: Raised when an operation or function is performed on an object of an inappropriate type.

ValueError: Raised when a built-in operation or function receives an argument of the correct type but with an invalid value.

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

IndexError: Raised when trying to access an index of a sequence (such as a list or tuple) that is out of range.

KeyError: Raised when trying to access a dictionary key that does not exist.

FileNotFoundError: Raised when a file or directory is not found.

ZeroDivisionError: Raised when division or modulo operation is performed with zero as the divisor.

AttributeError: Raised when an attribute reference or assignment fails.

These are just a few examples of the many built-in exceptions provided by Python. Each exception serves a specific purpose and can help identify and handle different types of errors that may occur during program execution.

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

Logging in Python is a built-in module that provides a flexible and powerful mechanism for recording events, messages, and errors that occur during the execution of a program. It allows developers to collect and store valuable information about the program's behavior, making it easier to understand and debug issues.

Logging is important in software development for several reasons:

Debugging and Troubleshooting: Logging provides a way to capture relevant information about the program's execution, including error messages, stack traces, variable values, and other diagnostic details. It helps developers identify and diagnose issues more effectively, speeding up the debugging and troubleshooting process.

Error and Exception Tracking: By logging errors and exceptions, developers can track and monitor the occurrence of specific issues in a program. This allows them to identify recurring problems, prioritize fixes, and ensure that critical errors are not missed.

Application Monitoring: Logging can be used to monitor the behavior and performance of an application in production environments. By logging important events and metrics, developers can gain insights into how the software is functioning, identify bottlenecks, and optimize performance.

Audit Trail and Compliance: Logging can serve as an audit trail by recording important events and actions within a system. This can be valuable for compliance purposes, as it helps track user activities, system changes, and other relevant information.

Communication and Collaboration: Logging provides a shared context and history of the program's execution. It allows developers to communicate and collaborate effectively by sharing log files or extracting relevant information from logs.

Python's logging module offers various features such as log levels, log formatting, log handlers, and log filtering, allowing developers to customize and control the logging behavior according to their specific needs. It promotes good software development practices by encouraging structured and consistent logging, enabling better understanding, maintenance, and improvement of the software over time.

**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 allow developers to categorize and prioritize log messages based on their importance or severity. Each log level represents a specific level of detail or criticality of an event. The purpose of log levels is to control the verbosity of log output and provide flexibility in filtering and handling log messages.

Python's logging module defines the following log levels (in ascending order of severity):

DEBUG: The lowest log level. Used for detailed debugging information. Typically, these messages are only useful for developers during development or troubleshooting.

Example usage: Logging variable values or detailed function call information for debugging purposes.

INFO: Used to convey information about the progress or general operation of the program. These messages are usually intended to be informative and may include high-level status updates.

Example usage: Logging application startup information or significant milestones during program execution.

WARNING: Indicates potentially harmful situations or unexpected behavior that may cause issues but does not prevent the program from continuing. Warnings can serve as early indicators of problems that need attention.

Example usage: Logging deprecated function usage, non-critical configuration issues, or recoverable errors.

ERROR: Indicates errors that caused the program to fail to perform a specific task or operation. These messages highlight issues that need attention and may affect the program's functionality or expected behavior.

Example usage: Logging unhandled exceptions, database connection failures, or critical errors that prevent the program from continuing.

CRITICAL: The highest log level. Indicates severe errors or failures that may result in the program's termination or inability to continue execution. Critical messages require immediate attention.

Example usage: Logging unrecoverable errors, system failures, or security breaches.

By setting the log level appropriately, developers can control which log messages are emitted and how much detail is included in the log output. This allows them to tailor the logging behavior to suit different stages of development, deployment environments, or specific requirements.

For example, during development and testing, a lower log level like DEBUG can provide detailed insights into the program's behavior. In production environments, it's common to use a higher log level like INFO to capture important events and status updates, while only logging warnings, errors, or critical issues that require immediate attention.

It's important to strike a balance in log levels to avoid flooding log files with unnecessary or redundant information while ensuring that crucial events and errors are properly captured for analysis and troubleshooting.

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

**message format using formatters?**

Log formatters in Python logging are responsible for defining the structure and content of log messages. They allow developers to customize the format of log output, including the timestamp, log level, message, and any additional contextual information.

Python's logging module provides the Formatter class, which serves as the base class for creating log formatters. The Formatter class provides various attributes and methods that can be overridden to define the desired log message format.

To customize the log message format using formatters, you can follow these steps:

Create an instance of the Formatter class or its subclass.

formatter = logging.Formatter('[%(asctime)s] [%(levelname)s] %(message)s')

Assign the formatter to the desired log handler.

handler = logging.StreamHandler()

handler.setFormatter(formatter)

logger.addHandler(handler)

In this example, we assign the formatter to a StreamHandler, which will output log messages to the console. However, formatters can be used with other types of log handlers as well, such as file handlers or network handlers.

Customize the format by specifying the desired format string using formatter attributes and placeholders.

In the example above, the format string '[%(asctime)s] [%(levelname)s] %(message)s' is used. It includes the following placeholders:

%(asctime)s: The timestamp of the log message in the format specified by the formatter.

%(levelname)s: The log level of the message (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL).

%(message)s: The actual log message.

Additionally, there are other attributes and placeholders available to include additional information like logger name, module name, line number, process ID, etc.

By customizing the format string and using different placeholders, you can tailor the log message format to suit your specific requirements. You can add or remove placeholders, specify their order, and format them as needed.

Here's an example of a more elaborate log message format:

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

**8. How can you set up logging to capture log messages from multiple modules or**

**classes in a Python application?**

To capture log messages from multiple modules or classes in a Python application, you can set up a logger hierarchy using Python's logging module. The logger hierarchy allows you to define loggers for different modules or classes and configure their behavior individually.

Here's an approach to set up logging to capture log messages from multiple modules or classes:

Import the logging module:

import logging

Configure the root logger or a custom logger:

logging.basicConfig(level=logging.DEBUG) # Set the desired root logger level

logger = logging.getLogger("my\_application") # Create a custom logger

logger.setLevel(logging.DEBUG) # Set the desired logger level

Add log handlers to the logger:

handler = logging.StreamHandler() # Create a log handler (e.g., StreamHandler, FileHandler)

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

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

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

Create a log handler, such as StreamHandler for console output or FileHandler for file-based logging. Set a formatter for the handler to specify the desired log message format. Finally, add the handler to the logger.

Use the # In module1.py

import logging

logger = logging.getLogger("my\_application.module1")

# In module2.py

import logging

logger = logging.getLogger("my\_application.module2")logger in different modules or classes:

# In module1.py

import logging

logger = logging.getLogger("my\_application.module1")

# In module2.py

import logging

logger = logging.getLogger("my\_application.module2")

In each module or class where you want to capture log messages, retrieve the logger using getLogger() and provide a unique name for the logger. The logger name should follow the hierarchy structure, such as "my\_application.module1" or "my\_application.module2".

Log messages from different modules or classes:

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")

In each module or class, use the logger to emit log messages at the desired log levels (e.g., DEBUG, INFO, WARNING, ERROR). The log messages will be captured and processed based on the logger's level and the configured log handlers.

**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?**

The logging and print statements in Python serve different purposes and have distinct characteristics. Understanding the differences between them is important for choosing the appropriate one for different scenarios in a real-world application.

Here are the key differences between logging and print statements:

Output Destination:

Print statements output directly to the standard output (usually the console or terminal) by default.

Logging statements can be configured to output to various destinations, such as the console, log files, or external logging services.

Verbosity and Flexibility:

Print statements are typically used for immediate output of values or messages during program execution. They are straightforward and simple to use but lack flexibility and control over the formatting and filtering of output.

Logging statements offer greater flexibility and control over the log messages. You can specify log levels, format log messages, and direct them to different log handlers. This allows you to control the verbosity of the log output and filter log messages based on their severity or other criteria.

Levels and Filtering:

Print statements don't provide built-in levels or filters. All print statements are executed and displayed.

Logging statements support log levels (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL), allowing you to specify the importance or severity of log messages. You can configure the logging system to display only messages above a certain log level or to filter out messages based on specific criteria. This is especially useful in real-world applications where you want to differentiate between different levels of importance and focus on critical information while ignoring less important details.

Maintenance and Debugging:

Print statements are often used for quick debugging and temporary output during development. However, they can become hard to maintain or remove once the codebase grows, leading to cluttered code and reducing the maintainability of the application.

Logging statements, on the other hand, can be left in the code even in production environments. They provide a systematic way of capturing important information, errors, and warnings. Logging statements can be easily enabled or disabled based on the desired log level, making it easier to maintain and debug the application over time.

In a real-world application, it is generally recommended to use logging over print statements for several reasons:

Logging offers more control and flexibility in managing log messages, allowing you to adjust the log level, format, and destination as needed.

Logging statements can be left in the codebase without cluttering the code, making it easier to maintain and debug the application in various environments.

Logging provides a standardized approach to capturing and managing log messages, facilitating troubleshooting and error analysis.

Logging allows for better separation of concerns by providing a dedicated mechanism for capturing and handling log events, making it easier to integrate with existing logging infrastructure or external logging services.

However, there may be situations where print statements can be useful for quick and temporary debugging purposes, especially during development or when working with small scripts. But in larger applications or projects, using a comprehensive logging approach is preferred for better control, maintainability, and long-term debugging and analysis capabilities.

**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.**

To log a message to a file named "app.log" with the specified requirements, you can use the Python logging module. Here's an example program that accomplishes this:

import logging

# Configure the logger

logging.basicConfig(filename="app.log", level=logging.INFO, filemode="a")

# Log the message

logging.info("Hello, World!")

The logging.basicConfig() function is used to configure the logging system. We provide the following arguments:

filename: Specifies the name of the log file ("app.log").

level: Sets the log level to "INFO" so that only messages at or above this level will be logged.

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

After configuring the logger, we can use the logging.info() method to log the desired message ("Hello, World!"). This will write the log message to the "app.log" file with an "INFO" log level.

Each time you run the program, the log message will be appended to the existing "app.log" file without overwriting previous entries. You can open the log file to view the logged message.

**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.**

To log an error message to both the console and a file named "errors.log" when an exception occurs during the program's execution, you can use the Python logging module. Here's an example program that fulfills these requirements:

import logging

import datetime

# Configure the logger

logging.basicConfig(level=logging.ERROR, format='%(asctime)s - %(levelname)s - %(message)s')

# Create a file handler to log errors to the file

file\_handler = logging.FileHandler('errors.log')

file\_handler.setLevel(logging.ERROR)

file\_handler.setFormatter(logging.Formatter('%(asctime)s - %(levelname)s - %(message)s'))

# Add the file handler to the logger

logger = logging.getLogger('')

logger.addHandler(file\_handler)

try:

# Your code that may raise an exception

# For demonstration, let's divide by zero to force an exception

result = 10 / 0

except Exception as e:

# Log the exception with timestamp

error\_message = f"{type(e).\_\_name\_\_}: {str(e)}"

logging.error(error\_message)

# Print the exception to the console

print(error\_message)

Explanation:

We configure the logger using logging.basicConfig(). The level is set to logging.ERROR to only log error messages or higher severity. The format specifies the desired log message format, including the timestamp, log level, and message.

We create a FileHandler named file\_handler to log errors to the file "errors.log". We set its level to logging.ERROR to capture only error messages. The setFormatter() method is used to set the desired log message format for the file handler.

We add the file\_handler to the logger using logger.addHandler() to ensure that error messages are logged to both the console and the "errors.log" file.

The code inside the try block represents your actual program logic. In this example, we intentionally raise an exception by dividing by zero (10 / 0).

If an exception occurs, we catch it using the except block. The exception is logged using logging.error() to capture the error message with the timestamp.

The error message is also printed to the console using print() for immediate visibility during program execution.

Now, when an exception occurs, the error message containing the exception type, the exception message, and the timestamp will be logged to both the console and the "errors.log" file.