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 executed only if there is no exception raised in the 'try' block. It allows you to specify code that should be executed when the 'try' block does not raise any exceptions.

try:

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

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

result = num1 / num2

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

else:

print("Result:", result)

In this example, if the user enters valid input for both numbers, the division operation in the 'try' block will succeed, and the 'else' block will be executed to print the result. If there is a ZeroDivisionError, the 'except' block will be executed instead.

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

example.

Yes, This is useful when you want to handle different exceptions at different levels of code execution.

try:

try:

num = int(input("Enter a number: "))

result = 10 / num

print("Result:", result)

except ValueError:

print("Error: Invalid input. Please enter a valid number.")

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

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

demonstrates its usage.

You can create a custom exception class by defining a new class that inherits from the Exception base class or any other built-in exception class.

class MyCustomException(Exception):

pass

try:

raise MyCustomException("This is a custom exception.")

except MyCustomException as e:

print("Custom Exception:", e)

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

Common built-in exceptions in Python:

ZeroDivisionError: Raised when division or modulo by zero occurs.

ValueError: Raised when an inappropriate value is provided for a built-in operation or function.

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

IndexError: Raised when a sequence subscript is out of range.

FileNotFoundError: Raised when a file or directory is requested but cannot be found.

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

Logging in Python is a mechanism to record events, messages, and errors during the execution of a program. It provides a more structured and controlled way of recording information than using print statements. Logging is important in software development because it helps in understanding and diagnosing the behavior of the application, especially in large-scale projects.

6. Explain the purpose of log levels in Python logging and provide examples of when

each log level would be appropriate.

Log levels are used to categorize log messages based on their severity and importance. Python logging supports various log levels, such as DEBUG, INFO, WARNING, ERROR, and CRITICAL.

Examples of log levels usage:

DEBUG: Used for detailed information useful for debugging purposes.

INFO: Used for general information about the application's execution.

WARNING: Used to indicate potential issues that don't prevent the application from running.

ERROR: Used for reporting errors that caused the application to fail or behave unexpectedly.

CRITICAL: Used for severe errors that may result in the application's termination.

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

message format using formatters?

Log formatters determine the format of log messages before they are written to the log handlers (e.g., file, console). Python logging provides a variety of built-in formatters to customize log message formatting.

import logging

logger = logging.getLogger('my\_logger')

logger.setLevel(logging.INFO)

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

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

file\_handler.setFormatter(formatter)

logger.addHandler(file\_handler)

logger.info('Hello, World!')

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

classes in a Python application?

You can set up logging in a central location (e.g., the main script) and then use the same logger instance in multiple modules or classes. This way, all log messages will be captured by the same logger and can be directed to the same log file or console.

# main.py

import logging

import my\_module

logging.basicConfig(filename='app.log', level=logging.INFO)

logger = logging.getLogger()

logger.info('Started application')

my\_module.do\_something()

# my\_module.py

import logging

def do\_something():

logger = logging.getLogger(\_\_name\_\_)

logger.info('Doing something')

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?

Logging is more flexible and customizable than print statements.

Logging supports different log levels, allowing you to control the amount of information recorded.

Logging messages can be directed to different log handlers, such as files, console, or network, making it easier to manage the logs in various environments.

Logging can include timestamps, log levels, and other additional information, making it easier to diagnose and analyze issues in the application.

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

# Configure logging

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

# Log the message

logging.info('Hello, World!')

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

# Configure logging to write to "errors.log" file with ERROR log level

logging.basicConfig(filename='errors.log', level=logging.ERROR)

try:

# Code that may raise an exception

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

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

result = num1 / num2

print("Result:", result)

except ZeroDivisionError as e:

# Log the error message with timestamp

logging.error(f"Error: {e}, Timestamp: {logging.Formatter().formatTime()}")

# Print the error message to the console

print(f"Error: {e}")

except ValueError as ve:

# Log the error message with timestamp

logging.error(f"Error: {ve}, Timestamp: {logging.Formatter().formatTime()}")

# Print the error message to the console

print(f"Error: {ve}")