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 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 handle the code that should run when the 'try' block executes successfully without any exceptions being raised.

try:

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

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

result = num1 / num2

except ValueError:

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

except ZeroDivisionError:

print("Error: Division by zero is not allowed.")

else:

print("The division result is:", result)

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 is known as nested exception handling. It allows you to handle different levels of exceptions in a hierarchical manner, where the inner try-except block handles specific exceptions and the outer try-except block handles more general exceptions.

try:

# Outer try-except block

try:

# Inner try-except block

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

result = 10 / x

print("Result:", result)

except ValueError:

print("Invalid input. Please enter a valid integer.")

except ZeroDivisionError:

print("Cannot divide by zero.")

except Exception as e:

print("An error occurred:", str(e))

except Exception as e:

print("An error occurred at the outer level:", str(e))

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 inheriting from the built-in Exception class or any of its subclasses. By defining your own exception class, you can add custom attributes, methods, and behaviors to handle specific types of errors in your code.

class CustomException(Exception):

def \_\_init\_\_(self, message, error\_code):

self.message = message

self.error\_code = error\_code

super().\_\_init\_\_(message)

def \_\_str\_\_(self):

return f"CustomException: {self.message} (Error Code: {self.error\_code})"

# Usage example

def divide\_numbers(a, b):

if b == 0:

raise CustomException("Division by zero is not allowed.", 1001)

return a / b

try:

result = divide\_numbers(10, 0)

except CustomException as e:

print(e)

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

Ans: Python provides several built-in exceptions that are commonly used for handling various types of errors and exceptional situations. Here are some of the common exceptions in Python:

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 invalid value.

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

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

KeyError: Raised when a dictionary key is not found.

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

Ans:

In Python, logging refers to the process of recording events, messages, or information generated by a program during its execution. It involves capturing relevant data and storing it in log files or other output destinations. The logging module in Python provides a flexible framework for implementing logging functionality in applications.

Logging is important in software development for several reasons:

Debugging and troubleshooting: Logging allows developers to track the flow of execution and identify issues within their code. By logging relevant information, such as variable values, function calls, and error messages, developers can gain insights into the behavior of their program and pinpoint the source of problems.

Error and exception tracking: When an error or exception occurs during program execution, logging can help capture details about the error, such as the stack trace, error message, and context information. These logs can be invaluable in diagnosing and fixing issues, especially in production environments where direct access to the system may be limited.

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 and prioritize log messages based on their severity or importance. They provide a way to control the verbosity of log output and allow developers to filter and handle log messages based on their desired level of detail. Python's logging module provides several standard log levels, each serving a specific purpose. Here are the commonly used log levels and examples of when each level would be appropriate:

DEBUG:

This is the lowest level of logging used for detailed diagnostic information.

It is typically used during development or debugging phases to provide fine-grained details about the program's execution.

Example: Logging variable values or function call traces for troubleshooting a specific issue.

INFO:

This level is used to confirm that things are working as expected.

It provides information about the program's progress or significant milestones.

Example: Logging startup messages, successful connections, or major configuration changes.

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

message format using formatters?

Ans: In Python's logging module, log formatters are responsible for defining the format of log messages. They determine how log records are rendered into text, specifying the structure and content of each log message.

The logging module provides a default log formatter called Formatter, which defines a basic log message format that includes information such as the log level, timestamp, and log message. However, you can customize the log message format by creating your own formatter and specifying it for your logger.

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

classes in a Python application?

Ans: In Python, you can set up logging to capture log messages from multiple modules or classes using the built-in logging module. The logging module provides a flexible framework for emitting log messages from different parts of your application.

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?

In Python, logging and print statements serve different purposes and have different features. Here are the key differences between logging and print statements:

Output location: Print statements send their output to the standard output (usually the console or terminal) by default. Logging, on the other hand, allows you to direct the output to various destinations such as files, network sockets, or even external services like email or databases.

Granularity and control: Print statements are generally used for debugging or temporary information display. They provide a simple way to quickly inspect variables or trace program flow. On the contrary, logging provides more fine-grained control over what information gets logged and at what level of severity. You can configure different log levels (e.g., debug, info, warning, error, etc.) to filter and control the amount of information logged based on your needs.

Configurability: The logging module in Python offers extensive configuration options. You can define different loggers, handlers, formatters, and filters to customize the logging behavior. It allows you to control log message formatting, log levels, log propagation, and more. Print statements, on the other hand, provide limited configurability out of the box.

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:

import logging

# Set up the logging configuration

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.

Ans:

import logging

import datetime

def log\_exception():

# Create a logger

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

logger.setLevel(logging.ERROR)

# Create a file handler for the log file

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

file\_handler.setLevel(logging.ERROR)

# Create a console handler

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.ERROR)

# Create a formatter for the log message

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

file\_handler.setFormatter(formatter)

console\_handler.setFormatter(formatter)

# Add the handlers to the logger

logger.addHandler(file\_handler)

logger.addHandler(console\_handler)

try:

# Code that may raise an exception

# ...

raise ValueError("This is a sample exception.")

except Exception as e:

# Log the exception

logger.exception(str(e))

# Call the function to test the logging

log\_exception()