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 execute code if no exception occurs in the try block. The else block is optional, and it will only be executed if the try block does not raise an exception.

For example, the following code tries to divide two numbers. If the division is successful, the else block will be executed and the message "The division was successful" will be printed

try:

x = 10

y = 2

z = x / y

except ZeroDivisionError:

print("Division by zero")

else:

print("The division was successful")

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 called nested try-except. Nested try-except blocks can be used to handle multiple levels of exceptions. For example, you could use nested try-except blocks to handle exceptions that occur when opening a file, reading from a file, and writing to a file.

def factorial(n):

try:

try:

return 1 if n == 0 else n \* factorial(n - 1)

except ZeroDivisionError:

print("n cannot be 0")

except TypeError:

print("n must be an integer")

print(factorial(5))

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. Here's an example of how you can create a custom exception class:

class CustomException(Exception):

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

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

self.message = message

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

return f"CustomException: {self.message}"

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

ArithmeticError: This exception is raised when an arithmetic operation fails. For example, if you try to divide a number by zero, an ArithmeticError exception will be raised.

AssertionError: This exception is raised when an assertion fails. An assertion is a statement that checks if a condition is true. If the condition is not true, an AssertionError exception will be raised.

AttributeError: This exception is raised when an attribute is not found. For example, if you try to access the attribute name of a variable that does not have a name attribute, an AttributeError exception will be raised.

EOFError: This exception is raised when the end of a file is reached unexpectedly. This can happen if the file is empty, or if the user presses Ctrl+D to end the input.

FloatingPointError: This exception is raised when a floating-point operation fails. This can happen if the operation results in a number that is too large or too small to be represented by a floating-point number.

IndexError: This exception is raised when an index is used to access an element of a sequence that is out of bounds. For example, if you try to access the 10th element of a list that only has 5 elements, an IndexError will be raised.

KeyError: This exception is raised when a key is not found in a dictionary. For example, if you try to access the key "name" in a dictionary that does not have a "name" key, an KeyError exception will be raised.

KeyboardInterrupt: This exception is raised when the user presses Ctrl+C to interrupt a program.

LookupError: This exception is raised when a key or index is not found in a mapping or sequence.

NameError: This exception is raised when a variable is not found. For example, if you try to use the variable name but the variable name has not been defined, a NameError exception will be raised.

OSError: This exception is raised when an operating system error occurs. For example, if you try to open a file that does not exist, an OSError exception will be raised.

OverflowError: This exception is raised when the result of an arithmetic operation is too large to be represented by the data type of the operands. For example, if you try to add two very large integers, an OverflowError will be raised.

ReferenceError: This exception is raised when a reference to an object is not found. For example, if you try to delete a variable that has not been defined, a ReferenceError exception will be raised.

RuntimeError: This exception is raised when an error occurs that does not fall under any other category.

StopIteration: This exception is raised when the end of an iterator is reached. For example, if you try to iterate over a list that has no elements, a StopIteration exception will be raised.

SyntaxError: This exception is raised when a syntax error is found in a Python program. For example, if you forget to close a parenthesis, a SyntaxError exception will be raised.

SystemError: This exception is raised when a fatal error occurs in the Python interpreter.

TypeError: This exception is raised when an operation is performed on an object of an incorrect type. For example, if you try to add a string and an integer, a TypeError exception will be raised.

ValueError: This exception is raised when a function or operation is passed an argument of an incorrect type or value. For example, if you try to call the int() function with a string argument, a ValueError exception will be raised.

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

Logging is important in software development because it can help you to:

Debug your program: If your program is not working as expected, you can use logging to track down the source of the problem.

Track your program's progress: You can use logging to record the steps that your program takes as it executes. This can be helpful for understanding how your program works and for troubleshooting problems.

Understand what is happening in your program: You can use logging to record events that occur in your program, such as user input, database queries, and file operations. This can be helpful for understanding how your program works and for troubleshooting problems.

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

each log level would be appropriate.

In Python, log levels are used to control the verbosity of logging output. There are five standard log levels:

DEBUG: This level is used for debugging purposes. It should only be used when you are trying to track down a specific problem in your code.

INFO: This level is used for general information. It is a good level to use for logging the steps that your program takes as it executes.

WARNING: This level is used for warning messages. It is a good level to use for logging potential problems that may not cause your program to crash.

ERROR: This level is used for error messages. It is a good level to use for logging errors that will cause your program to crash.

CRITICAL: This level is used for critical errors. It is a good level to use for logging errors that will prevent your program from continuing to run.

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

message format using formatters?

In Python, log formatters are used to customize the format of log messages. You can use formatters to control the following aspects of log messages:

The message format: You can use formatters to control the format of the message itself, including the level, the timestamp, the source location, and the message content.

The output destination: You can use formatters to control where the log messages are sent, such as to a file, a console, or an email address.

The log level: You can use formatters to control which log levels are displayed.

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

classes in a Python application?

Here are the steps on how to set up logging to capture log messages from multiple modules or classes in a Python application:

Import the logging module.

Create a logger object for each module or class that you want to capture log messages from.

Configure the logger objects with the desired log level and format.

Use the logger objects to log messages.

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?

Here are the main differences between the logging and print statements in Python:

Logging is more flexible: Logging allows you to customize the format of your log messages, and you can also specify where your log messages are sent. Print statements, on the other hand, only allow you to print messages to the console.

Logging is more versatile: Logging can be used to log errors, warnings, and other events. Print statements can only be used to print messages.

Logging is more scalable: Logging can be used to log messages from multiple modules or classes in a Python application. Print statements can only be used to log messages from the current module or class.

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 log\_message(message, level):

"""Logs a message to a file named "app.log".

Args:

message (str): The message to log.

level (str): The log level.

"""

logger = logging.getLogger("app")

logger.setLevel(level)

file\_handler = logging.FileHandler("app.log", mode="a")

file\_handler.setFormatter(logging.Formatter("%(asctime)s: %(levelname)s: %(message)s"))

logger.addHandler(file\_handler)

logger.info(message)

if \_\_name\_\_ == "\_\_main\_\_":

log\_message("Hello, World!", level="INFO")

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 datetime

# Configure logging

logging.basicConfig(

level=logging.ERROR,

format="%(asctime)s [%(levelname)s] %(message)s",

handlers=[

logging.FileHandler("errors.log"),

logging.StreamHandler()

]

)

def log\_error(exception\_type, message):

timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

error\_message = f"{timestamp} [{exception\_type.\_\_name\_\_}] {message}"

logging.error(error\_message)

try:

# Your program code here

# ...

# Simulate an exception

raise ValueError("Something went wrong!")

except Exception as e:

log\_error(type(e), str(e))