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 ‘try-except’ statement runs if no exception occurs in try block. It is useful for code that should run only if try block runs successfully.

e.g.

try:

result= 10/2

except ZeroDivisionError:

print(‘Zero division error!’)

else:

print(“Division successful: ”, result)

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

example.

Ans: Yes, try-except block can be nested inside another try-except block

e.g.

try:

num= int(input(“Please enter a number: ”))

try:

result= 10/ num

except ZeroDivisionError:

print(‘Zero division error!’)

else:

print(“Result is: ”, result)

except ValueError:

print(“Please provide valid input!”)

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

demonstrates its usage.

Ans: Yes, you can create a custom exception class in python by subclassing a built-in ‘exception’ class.

e.g.

class CustomError(Exception):

pass

def check\_value(value):

if value < 0:

raise CustomError("Value cannot be negative!")

try:

check\_value(-5)

except CustomError as e:

print("Caught a custom error: ", e)

# Output: Caught a custom error: Value cannot be negative!

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

Ans:

Common built-in exceptions in python include:

1. ValueError: Raised when a function receives an argument of right type but inappropriate value.
2. TyepeError: Raised when an operation or a function is applied to an object of inappropriate type.
3. IndexError: Raised when you try to access the index that is out of range.
4. KeyError: Raised when a dictionary key is not found.
5. ZerorDivisionError: Raised when you try to divide number by zero.

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

each log level would be appropriate.

Ans:

1. Debug: Detailed information, typically used for debugging purpose.
2. Info: General information about the programs progress, typically used for informational purpose.
3. Warning: Indicates a potential problem, this is something not critical but could cause issue.
4. Error: Indicates more serious problem, that caused the programme to fail to perform some function.
5. Critical: Indicates a critical error that led to terminate the programme.

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

message format using formatters?

Ans: log formatters in python logging specific the layout of the messages, they define how log records are presented. Including format of the date, time and message.

Import logging

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

logger.setLevel(logging.DEBUG)

handler = logging.StreamHandler()

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

handler.setFormatter(formatter)

logger.addHandler(handler)

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 follow below steps:

# Import logging method

import logging

# Create logger

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

logger.setLevel(logging.DEBUG)

# Create a handler

console\_handler = logging.StreamHandler()

# Create a formater

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

# Set a formatter to the handler

console\_handler.setFormatter(formatter)

# Add handler to the logger

logger.addHandler(console\_handler)

# In each module or class where you want to log message import logger and log messages

import logging

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

logger.debug('Debug message')

logger.info('Info message')

logger.warning('Warning message')

logger.error('Error message')

logger.critical('Critical 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:

* Difference between logging and print statement are:

Logging is for recording events and messages and systematically, providing control over where and how they are stored.

Print statements are for immediate output to the console during development or debugging.

* When to use logging over print statement:

Use logging in real world apps for organised event recordings, error handling and controlling output levels.

Use print mainly during development for quick output and debugging, specially for temporary checks.

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

# Create a logger

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

logger.setLevel(logging.INFO)

# Create a file handler that appends to the log file

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

# Create a formatter

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

# Set the formatter for the handler

file\_handler.setFormatter(formatter)

# Add the handler to the logger

logger.addHandler(file\_handler)

# Log the message

logger.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 traceback

# Set up logger

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

logger.setLevel(logging.ERROR)

# Create handlers

console\_handler = logging.StreamHandler()

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

# Create a formatter

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

# Set the formatter for the handlers

console\_handler.setFormatter(formatter)

file\_handler.setFormatter(formatter)

# Add handlers to the logger

logger.addHandler(console\_handler)

logger.addHandler(file\_handler)

try:

# Example code that may raise an exception

x = 1 / 0

except Exception as e:

# Log the error message with exception type and timestamp

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

logger.error(error\_message)

logger.error(traceback.format\_exc())